Elements of Statistical Computation

Longhai Li

2026-11-27

Table of contents

Preface 3
Table of Contents . . . 3

1 Introduction to Statistical Computation 4
1.1 Review of Basic Concepts . . . 4
1.1.1 Statistical Inference: The Setup . . . 4
1.1.2 Point Estimation . . . 4
1.1.3 Evaluating an Estimator . . . 5
1.1.4 Maximum Likelihood Estimation . . . 6
1.1.5 Hypothesis Testing . . . 6
1.1.6 p-value, Type I Error, and Power . . . 7
1.1.7 Bayesian Inference . . . 8
1.1.8 Prior, Likelihood, and Posterior . . . 8
1.1.9 Summary of the Three Tasks . . . 8
1.2 The Use of Computers in Statistics . . . 9
1.2.1 1. Manipulating Numbers . . . 9
1.2.2 2. Simulation with Random Numbers . . . 10
1.2.3 3. Optimization . . . 11
1.2.4 4. Advanced Simulation and Integration . . . 11
1.2.5 Course Roadmap . . . 12

2 Introduction to R and Quarto 14
2.1 A toy example . . . 14
2.2 Generating an artificial score dataset . . . 14
2.2.1 Vectors: recording quiz scores . . . 14
2.2.2 Strings: naming the categories and students . . . 15
2.2.3 Simulating the tests and the final using vectorized math . . . 15
2.2.4 Matrices: combining all the scores (Complete Data) . . . 17
2.2.5 Adding and handling missing values . . . 18
2.2.6 Data frames: adding names and program . . . 19
2.3 Calculating a Single Student's Score . . . 19
2.3.1 Aggregating into a Reusable Function . . . 20
2.4 Applying the Function to Every Student . . . 22
2.4.1 Using `apply` function . . . 22
2.4.2 Looping the rows with `for` . . . 23
2.4.3 Saving and reloading the results . . . 24
2.5 Writing a Function for the Whole Process with List Output . . . 24
2.6 The use of pipe `|>` . . . 26

2.7 Data Manipulation with `dplyr` . . . 27
2.7.1 Generating a roster with `mutate()` . . . 27
2.7.2 Recomputing course scores with `mutate()` and `rowwise()` . . . 28
2.7.3 Summarizing by group with `group_by()` and `summarize()` . . . 29
2.7.4 Filtering and sorting with `filter()` and `arrange()` . . . 30
2.8 Presenting the Results . . . 30
2.8.1 Presenting the Processed Results with `gt` . . . 30
2.8.2 Base R graphics . . . 32
2.8.3 Plotting with ggplot2 . . . 33

3 Computer Arithmetic **36**
3.1 Representations of Numbers in Computer . . . 36
3.1.1 Integers . . . 36
3.1.2 Floating point numbers . . . 38
3.1.3 Zoom Models for Representable Floating Numbers . . . 38
3.2 Methods for Mitigating Rounding Errors . . . 46
3.2.1 Roundoff errors . . . 46
3.2.2 Computing sums: avoiding large + small . . . 46
3.2.3 Computing e^x: avoiding large − large . . . 49
3.2.4 Computing variance: avoiding large − large . . . 52
3.3 Methods for Mitigating Overflow and Underflow Errors . . . 55
3.3.1 Where overflow and underflow arise in statistics . . . 55
3.3.2 Changing the algebraic expression . . . 55
3.3.3 Representing numbers in logarithmic form . . . 57
3.3.4 The log-sum-exp trick . . . 58
3.3.5 Differences on the log scale . . . 60
3.3.6 Computing $\log(1 - e^{-u})$: large + small and large − large together . . . 61
3.3.7 A combined example: $\log\{1 - \Phi(x)\}$ at both tails . . . 63
3.4 Summary . . . 65
3.5 Exercises . . . 65

4 Random Numbers and Monte Carlo Methods **67**
4.1 Monte Carlo Methods . . . 67
4.1.1 Pseudo Random Numbers . . . 67
4.1.2 Seeds and Reproducibility . . . 69
4.1.3 Generating Unif(a, b) . . . 69
4.1.4 Inverting the CDF . . . 70
4.1.5 Example: The Exponential Distribution . . . 71
4.1.6 Shinylive App for Inverse CDF Sampling . . . 72
4.1.7 A Special Transformation for Generating Normal Samples . . . 72
4.1.8 Two Standard Normals in Polar Coordinates . . . 72
4.1.9 The Box–Muller Transformation . . . 73
4.1.10 Law of Large Numbers and Central Limit Theorem . . . 75
4.1.11 Monte Carlo Error Bound from the CLT . . . 75
4.1.12 Demonstrations . . . 76
4.1.13 An Example of Monte Carlo for Estimating π . . . 78

4.2 Point Estimation . . . 82
4.2.1 Monte Carlo Evaluation of an Estimator . . . 83
4.2.2 Example 1: Two Estimators of Proportions . . . 83
4.2.3 Example 2: Mean, Median and Trimmed Mean . . . 85
4.3 Hypothesis Testing . . . 86
4.3.1 The Two Types of Error . . . 86
4.3.2 Estimating False and True Positive Rates by Monte Carlo . . . 87
4.3.3 The p-value: A Probability Integral Transform of T under H_0 . . . 87
4.3.4 Uniformity of the p-value under H_0 . . . 88
4.3.5 Monte Carlo Estimates of Size and Power . . . 88
4.3.6 Example 1: The t-test under Different Data Settings . . . 89
4.3.7 Monte Carlo for Calculating (Not Evaluating) a p-value . . . 95
4.3.8 Example 2: A Permutation Test for the Correlation Coefficient . . . 95
4.3.9 Example 3: Permuation Correction for Post-Selection Linear Regression . . . 98
4.3.10 Advanced Computation-Intensive Tests . . . 102

5 Optimization for Maximum Likelihood Estimation 108
5.1 Maximum Likelihood: Theory . . . 108
5.1.1 Likelihood, Negative Log-Likelihood, and Information . . . 108
5.1.2 Example: iid Bernoulli . . . 109
5.1.3 Closed Form: the Normal Mean . . . 110
5.1.4 Consistency, Asymptotic Normality, and Fisher Information . . . 111
5.1.5 Demonstration: the Cauchy Location Model . . . 112
5.2 Univariate Optimization Algorithms . . . 113
5.2.1 The Running Example: Zero-Truncated Poisson Counts . . . 114
5.2.2 Three Equivalent Formulations . . . 115
5.2.3 Fixed-Point (Simple) Iteration . . . 116
5.2.4 Bisection . . . 117
5.2.5 Golden Section Search . . . 118
5.2.6 Newton–Raphson . . . 121
5.2.7 Advantage: quadratic convergence . . . 122
5.2.8 A Classic Illustration: Newton's Method for $\sqrt{a}$ . . . 123
5.2.9 Problem: divergence . . . 124
5.2.10 Univariate Fisher Scoring . . . 126
5.2.11 Safeguards . . . 127
5.2.12 Numerical Derivatives . . . 127
5.2.13 Summary of the Methods . . . 128
5.2.14 Putting the Methods Together on the Truncated Poisson . . . 128
5.3 Nelder–Mead Method for Multivariate Optimization . . . 134
5.3.1 One iteration . . . 134
5.3.2 Implementation . . . 136
5.3.3 Example: the Rosenbrock banana . . . 136
5.3.4 Strengths and limits . . . 138
5.3.5 Shinylive App for the Nelder–Mead Algorithm . . . 138
5.4 Multivariate Newton-Raphson Optimzation Methods . . . 138
5.4.1 Gradient and Hessian: A Geometric Illustration . . . 138

5.4.2 Naive Newton-Raphson Iterations . . . 140
5.4.3 Convergence properties . . . 142
5.4.4 Standard errors as a by-product . . . 142
5.4.5 Enhancements to Multivariate Newton-Raphson Methods . . . 142
5.5 Example of Fitting Logistic Regression with Safeguarded Multivariate Newton-Raphson . . . 144
5.5.1 The model and its derivatives . . . 144
5.5.2 The gradient by the chain rule . . . 145
5.5.3 Implementation in R . . . 148
5.5.4 Comparison with `nlm` and `glm` . . . 149
5.6 Backpropagation Algorithm for Neural Networks . . . 150
5.6.1 Example of Poisson Neural Network . . . 150
5.6.2 What the model can represent . . . 151
5.6.3 The gradient by backpropagation . . . 152
5.6.4 Two kinds of arrow . . . 153
5.6.5 A General Description of Gradient of Neural Networks . . . 161
5.7 Gradient-Based Methods . . . 163
5.7.1 Classification by derivative information . . . 163
5.7.2 Steepest Descent with Line Search . . . 164
5.7.3 Why Gradient Descent Zig-Zags . . . 164
5.7.4 Conjugate Gradient . . . 165
5.7.5 Quasi-Newton: BFGS . . . 166
5.7.6 Stochastic Gradient Descent . . . 167
5.7.7 Shinylive App for Comparing the Optimizers . . . 173
5.8 Examples of Fitting Neural Networks (Shinylive App Illustration) . . . 173
5.8.1 Gaussian Neural Network . . . 173
5.8.2 Poisson Neural Network . . . 174
5.9 Examples of Fitting NN with Tensorflow . . . 174
5.10 Remarks on Implementation of Optimization Algorithms . . . 177
5.10.1 One Form for All the Methods . . . 177
5.10.2 Comparison . . . 178
5.10.3 Software Packages for Optimization . . . 179
5.10.4 Optimization in Practice for Deep Learning . . . 181

6 EM Algorithms **182**
6.1 Overview of EM . . . 182
6.1.1 Setup and motivation . . . 182
6.1.2 The algorithm . . . 182
6.1.3 Q as an average of complete log-likelihoods . . . 183
6.2 Example 1: Two-Component Normal Mixture . . . 183
6.2.1 Model and likelihoods . . . 183
6.2.2 Complete log-likelihood . . . 185
6.2.3 E-step: soft classification . . . 185
6.2.4 M-step: weighted proportions and means . . . 186
6.2.5 Implementation . . . 187
6.2.6 Shinylive App for the EM Algorithm for a Normal Mixture . . . 189
6.2.7 Comparison with general-purpose optimizers . . . 189

6.3 Example 2: Censored Poisson Data . . . 191
6.3.1 Truncated distributions in brief . . . 191
6.3.2 Data and likelihoods . . . 191
6.3.3 E-step . . . 192
6.3.4 M-step and the EM update . . . 193
6.3.5 Implementation . . . 193
6.3.6 Comparison with general-purpose optimizers . . . 195
6.4 Example 3: Right-Censored Weibull Data . . . 195
6.4.1 Data and likelihoods . . . 196
6.4.2 E-step . . . 197
6.4.3 M-step . . . 199
6.4.4 Special case: known shape . . . 199
6.4.5 Implementation . . . 199
6.4.6 Simulation 1: one data set . . . 201
6.4.7 Simulation 2: the censoring rate sets the speed . . . 203
6.4.8 Simulation 3: what ignoring the censoring costs . . . 205
6.5 Example 4: Logistic Regression as a Latent-Variable Problem . . . 207
6.5.1 The latent-variable model . . . 207
6.5.2 E-step . . . 208
6.5.3 M-step . . . 209
6.5.4 The fixed point is the logistic MLE . . . 210
6.5.5 Implementation . . . 210
6.5.6 Comparison with `glm` . . . 211
6.5.7 Convergence rate and the missing information . . . 212
6.5.8 Remarks . . . 213
6.6 More Complex Applications . . . 214
6.6.1 Model-based clustering . . . 214
6.6.2 Hidden Markov models . . . 215
6.6.3 Factor analysis and the common pattern . . . 216
6.7 Justification of the EM Algorithm . . . 216
6.7.1 Cross-entropy and Kullback-Leibler distance . . . 217
6.7.2 The key identity . . . 217
6.7.3 The ascending property . . . 218
6.7.4 The function $F(\tilde{P}, \theta)$ . . . 220
6.7.5 EM as alternating optimization . . . 220
6.8 Summary . . . 221
6.9 References . . . 222

7 Introduction to Bayesian Inference 223
7.1 The Bayesian Framework . . . 223
7.2 Summarizing the Posterior . . . 224
7.2.1 Point estimates from loss functions . . . 224
7.2.2 Posterior prediction . . . 225
7.3 Example: Bernoulli Data with a Beta Prior . . . 226
7.3.1 Prior . . . 226
7.3.2 Likelihood and posterior . . . 227

7.3.3 Posterior summaries . . . 228
7.3.4 The marginal likelihood in closed form . . . 228
7.3.5 Posterior predictive, in closed form . . . 228
7.3.6 Worked numbers: the complete inference output . . . 229
7.4 Example: Poisson Data with a Gamma Prior . . . 230
7.4.1 Prior . . . 230
7.4.2 Likelihood and posterior . . . 230
7.4.3 The marginal likelihood in closed form . . . 231
7.4.4 Posterior predictive, in closed form . . . 231
7.4.5 Worked numbers: the complete inference output . . . 232
7.4.6 Overdispersion, visualized . . . 233
7.5 Example: Normal Data with a Normal Prior (Known Variance) . . . 234
7.5.1 Prior . . . 234
7.5.2 Likelihood and posterior . . . 234
7.5.3 The marginal likelihood in closed form . . . 235
7.5.4 Posterior predictive, in closed form . . . 235
7.5.5 Worked numbers: the complete inference output . . . 235
7.5.6 Overdispersion, visualized . . . 237
7.6 Shinylive App for Conjugate Bayesian Updating . . . 237
7.7 Where Computation Enters . . . 237
7.8 Summary . . . 238

8 Numerical Quadrature 239
8.1 The Problem and Transformations . . . 239
8.1.1 The general setup . . . 239
8.1.2 Infinite ranges: the logistic transformation . . . 239
8.1.3 Probabilities as a special case . . . 240
8.2 Quadrature Rules on a Finite Interval . . . 241
8.2.1 Midpoint rule . . . 241
8.2.2 Trapezoidal rule . . . 241
8.2.3 Simpson's rule . . . 241
8.2.4 Implementation and accuracy . . . 243
8.3 Application: The Marginal Likelihood . . . 245
8.3.1 Setup . . . 245
8.3.2 A two-parameter example: mean and variance unknown . . . 245
8.3.3 Computing on the log scale . . . 246
8.3.4 An efficient implementation . . . 246
8.3.5 Visualizing the grid and the integrand . . . 248
8.3.6 Checking the quadrature against Monte Carlo . . . 248
8.3.7 Convergence as the grid is refined . . . 249
8.3.8 Comparing priors . . . 249
8.3.9 Comparing likelihoods . . . 250
8.4 Difficulties of Numerical Quadrature . . . 251
8.4.1 When the grid misses the posterior's mode . . . 251
8.4.2 The same failure in the full two-parameter model . . . 252
8.4.3 The curse of dimensionality . . . 253

8.5 Summary . . . 254

9 Laplace Approximation **255**
9.1 Laplace Approximation Formula . . . 255
9.1.1 Example: A Poisson Rate with a Gamma Prior . . . 257
9.1.2 Linear functionals: posterior mean and credible intervals . . . 259
9.1.3 Example: Computing ML of Normal Data with an Unknown Mean . . . 260
9.2 Quadrature Adapted by the Laplace Approximation . . . 264
9.2.1 Univraite Laplace-adapted midpoint rule . . . 264
9.2.2 Example: Computing ML of Normal Data with an Unknown Mean . . . 266
9.2.3 Multivariate Laplace-adapted midpoint rule . . . 266
9.2.4 Example: Computing ML of Normal Data with Unknown Mean and Variance . . . 269
9.2.5 Example: Choosing the Number of Predictors in Logistic Regression . . . 269
9.3 Remarks on the Laplace Approximation . . . 273
9.3.1 BIC as a large-n simplification . . . 273
9.3.2 When Laplace Works Well, and When It Does Not . . . 274
9.4 Summary . . . 274

10 Rejection Sampling **275**
10.1 The Envelope and the Algorithm . . . 275
10.1.1 The envelope condition . . . 275
10.1.2 Uniform points under a curve . . . 275
10.1.3 The algorithm in two forms . . . 277
10.1.4 Why the retained points have density f . . . 278
10.2 Example 1: Sampling a Normal from a Laplace Envelope . . . 279
10.3 Example 2: Sampling a Gamma Distribution . . . 280
10.3.1 Checking the envelope numerically . . . 281
10.3.2 Sampling function . . . 283
10.3.3 Test cases and efficiency . . . 283
10.3.4 Shinylive App for Rejection Sampling from a Gamma Density with a Cauchy Envelope . . . 284
10.4 Adaptive Rejection Sampling for Log-Concave Targets . . . 284
10.4.1 Log-concavity and the tangent hull . . . 284
10.4.2 The adaptive algorithm . . . 285
10.5 Example 3: Truncated Normal by Adaptive Rejection Sampling . . . 285
10.6 Example 4: Gamma Distribution by ARS on $\log X$ . . . 288
10.6.1 Shinylive App for Sampling a Gamma Distribution by ARS on $\log X$ . . . 293
10.7 Summary . . . 293

11 Importance Sampling **295**
11.1 Goal and Idea . . . 295
11.1.1 The Goal . . . 295
11.1.2 The Idea: Reweighting . . . 295
11.1.3 Picture: Areas and Weights . . . 296
11.2 The Importance Reweighting Formula . . . 297
11.2.1 Derivation . . . 297
11.2.2 The Estimators . . . 298

11.2.3 Effective Sample Size . . . 298
11.2.4 Derivation of ESS . . . 299
11.2.5 Limitations of the ESS Diagnostic . . . 300
11.3 Example 1: The Volume of a Set . . . 300
11.3.1 Indicator Functions as f and g . . . 300
11.3.2 Variance . . . 302
11.3.3 Effective Sample Size . . . 302
11.3.4 What n_{eff} Means . . . 302
11.3.5 Numerical Example: Area of a Crescent . . . 303
11.4 Example 2: Tail Probability of a Student-t Distribution . . . 306
11.4.1 The Problem and the Proposal . . . 306
11.4.2 The Target, the Proposal and the Estimator . . . 307
11.4.3 Implementation and Results . . . 309
11.4.4 Shinylive App for Estimating a t Probability with a Normal Sample . . . 310
11.5 Example 3: Same Support, but a Mode Is Missed . . . 310
11.5.1 The Setup . . . 310
11.5.2 The Figure . . . 311
11.5.3 Shinylive App for the Failure of Importance Sampling . . . 313
11.6 Example 4: Marginal Likelihood of Normal Data via Importance Sampling . . . 314
11.6.1 The Target and the Proposal . . . 314
11.6.2 The Contours of f and g . . . 314
11.6.3 Implementation . . . 316
11.6.4 Results Across Sample Sizes . . . 317
11.7 Example 5: Choosing the Number of Predictors in Logistic Regression . . . 319
11.8 Advanced Applications of Importance Sampling . . . 323
11.8.1 Free Energy and the Marginal Likelihood . . . 323
11.8.2 Stepping Stones: Many Small Jumps Instead of One Big One . . . 323
11.8.3 Bayesian Leave-One-Out Cross-Validation . . . 325
11.9 Summary . . . 326

12 Markov Chain Monte Carlo **327**
12.1 Markov Chains . . . 327
12.1.1 Definition and the convergence question . . . 327
12.1.2 Invariance, detailed balance, and convergence . . . 328
12.1.3 Example 1: Two Random Walks on a Cycle . . . 328
12.2 Gibbs Sampling . . . 333
12.2.1 The idea and why it works . . . 333
12.2.2 Data augmentation . . . 333
12.2.3 Example 2: Normal Data with Unknown Mean and Variance . . . 333
12.2.4 Example 3: Interval-Censored Normal Data . . . 339
12.2.5 Example 4: Cauchy Data with Unknown Location and Scale . . . 343
12.3 Metropolis–Hastings . . . 348
12.3.1 The algorithm . . . 348
12.3.2 Random-walk Metropolis . . . 348
12.3.3 Choosing the step size . . . 349
12.3.4 Metropolis-within-Gibbs . . . 350

12.3.5 Example 5: A Normal Truncated to Its Tail . . . 350
12.3.6 Example 6: Normal Data with Unknown Mean and Variance, Revisited . . . 352
12.4 Hamiltonian Monte Carlo and Stan . . . 356
12.4.1 Why random walks struggle . . . 356
12.4.2 The algorithm . . . 356
12.4.3 Example 7: Bivariate Normal with Correlation 0.98 . . . 357
12.4.4 Stan: automating HMC . . . 358
12.4.5 Where Stan earns its keep . . . 358
12.4.6 Example 8: Normal Data with Unknown Mean and Variance, in Stan . . . 359
12.4.7 Example 9: Cauchy Data with Unknown Location and Scale, in Stan . . . 360
12.5 A Final Practical Example: Sparse Logistic Regression . . . 361
12.5.1 The problem . . . 362
12.5.2 Why maximum likelihood fails . . . 362
12.5.3 A sparse prior: the horseshoe . . . 362
12.5.4 Fitting in Stan . . . 362
12.5.5 The horseshoe posterior . . . 364
12.5.6 Prediction on test data . . . 365
12.5.7 What `glm` and the LASSO estimate . . . 368
12.6 Summary . . . 371

Appendices **372**

A List of Figures, Tables, and Apps **372**
A.1 List of Shinylive Apps . . . 372
A.2 List of Figures . . . 372
A.2.1 Introduction to Statistical Computation . . . 372
A.2.2 Introduction to R and Quarto . . . 372
A.2.3 Computer Arithmetic . . . 373
A.2.4 Random Numbers and Monte Carlo Methods . . . 373
A.2.5 Optimization for Maximum Likelihood Estimation . . . 374
A.2.6 EM Algorithms . . . 375
A.2.7 Introduction to Bayesian Inference . . . 376
A.2.8 Numerical Quadrature . . . 376
A.2.9 Laplace Approximation . . . 377
A.2.10 Rejection Sampling . . . 377
A.2.11 Importance Sampling . . . 377
A.2.12 Markov Chain Monte Carlo . . . 378
A.3 List of Tables . . . 378
A.3.1 Introduction to R and Quarto . . . 378
A.3.2 Computer Arithmetic . . . 378
A.3.3 Random Numbers and Monte Carlo Methods . . . 379
A.3.4 EM Algorithms . . . 379
A.3.5 Introduction to Bayesian Inference . . . 379
A.3.6 Numerical Quadrature . . . 379
A.3.7 Laplace Approximation . . . 380

A.3.8 Rejection Sampling 380
A.3.9 Importance Sampling 380
A.3.10 Markov Chain Monte Carlo 380

Preface

This is a concise course about statistical inference, which was developed for the course STAT 420/812 at University of Saskatchewan.

Table of Contents

1. Introduction to Statistical Computation
2. Introduction to R and Quarto
3. Computer Arithmetic
4. Random Numbers and Monte Carlo Methods
5. Optimization for Maximum Likelihood Estimation
6. EM Algorithms
7. Introduction to Bayesian Inference
8. Numerical Quadrature
9. Laplace Approximation
10. Rejection Sampling
11. Importance Sampling
12. Markov Chain Monte Carlo

Appendices

A. List of Figures, Tables, and Apps

1 Introduction to Statistical Computation

Statistics is a science of inference from data, and every inference eventually becomes a calculation. This opening chapter reviews the three basic tasks of statistical inference — point estimation, hypothesis testing, and Bayesian inference — and shows that each of them, stated honestly, reduces to a computational problem: calculating a quantity reliably, simulating from a distribution, optimizing a function, or integrating over a parameter. The rest of the book is organized around these four computational primitives, and this chapter is a map of where each one is needed and why closed-form answers run out so quickly outside of textbook examples.

1.1 Review of Basic Concepts

1.1.1 Statistical Inference: The Setup

A statistical model starts from data $x_1, \dots, x_n$, assumed to be observed values of independent and identically distributed random variables with density or probability mass function $f(x \mid \theta)$, where θ is an unknown parameter (possibly a vector). Almost everything a statistician does with such a model falls into one of three tasks:

1. **Point estimation**: produce a single value $\hat{\theta}$ that is close to the unknown θ;
2. **Hypothesis testing**: decide whether the data are compatible with a specific hypothesis H_0 about θ;
3. **Bayesian inference**: describe the full state of uncertainty about θ after seeing the data, as a probability distribution rather than a single number.

The three tasks ask different questions of the same model, and, as this chapter will show, they differ mainly in *what has to be computed* to answer them.

1.1.2 Point Estimation

An **estimator** is simply a function of the data,

$$\hat{\theta} = \hat{\theta}(x_1, \dots, x_n), \tag{1.1}$$

computed before any particular sample is observed; once the data are in hand, evaluating the function gives an **estimate**. Three estimators are familiar from an introductory course:

- the sample mean $\hat{\mu} = \bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$, estimating a population mean;
- the sample variance $\hat{\sigma}^2 = S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$, estimating a population variance;
- the least-squares estimator in linear regression, $\hat{\beta} = (X'X)^{-1}X'y$, estimating a vector of regression coefficients.

Each is an explicit formula that a computer evaluates in one pass over the data. The interesting statistical question is not how to compute these three particular estimators — that part is trivial — but how to *evaluate* an estimator once it has been proposed, and how to construct one when no such tidy formula exists.

1.1.3 Evaluating an Estimator

Because $\hat{\theta}$ is a function of random data, it is itself a random variable, with its own **sampling distribution**: the distribution of the values $\hat{\theta}$ would take if the whole data-collection process were repeated over and over. Two estimators for the same θ can have sampling distributions that differ in location and in spread, as in the schematic below.

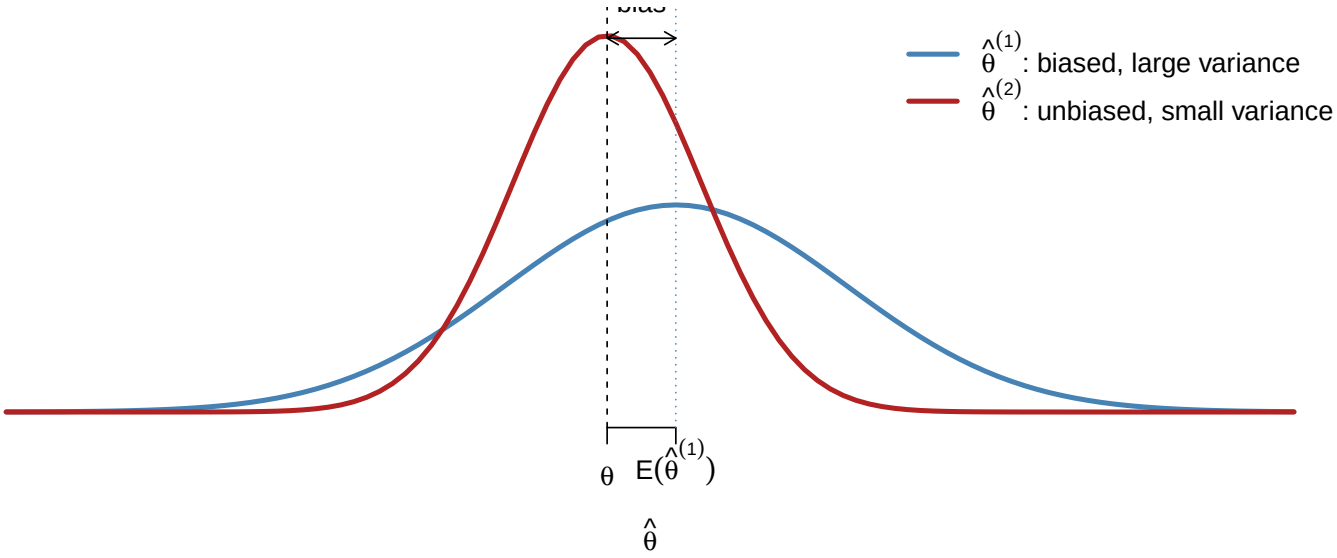


Figure 1.1: Illustrates the two components of estimator quality, bias and variance, by contrasting the sampling distributions of two estimators of the same parameter. Two sampling distributions for the same target θ: $\hat{\theta}^{(1)}$ is biased with large variance, $\hat{\theta}^{(2)}$ is unbiased with small variance.

Three numbers summarize a sampling distribution for the purpose of comparing estimators:

- the **bias**, $E(\hat{\theta}) - \theta$, how far the sampling distribution's center sits from the truth;
- the **variance**, $V(\hat{\theta})$, how spread out the sampling distribution is around its own center;
- the **mean squared error**, $\text{MSE}(\hat{\theta}) = E(\hat{\theta} - \theta)^2 = \text{Bias}(\hat{\theta})^2 + V(\hat{\theta})$, which combines both into a single criterion.

Deciding whether $\hat{\theta}^{(1)}$ or $\hat{\theta}^{(2)}$ is the better estimator therefore requires knowing — or, when no closed form is available, approximating — both sampling distributions. This is the first place computation enters: when $\hat{\theta}$ is a complicated function of the data, its exact sampling distribution is rarely available in closed form, and the only way to see it is to simulate many datasets from the assumed model, compute $\hat{\theta}$ on each one, and look at the resulting histogram. That technique, Monte Carlo simulation, is the subject of the next unit.

1.1.4 Maximum Likelihood Estimation

Among all conceivable estimators, one construction recurs throughout applied statistics: the **maximum likelihood estimator**. Writing the joint density of the data as a function of θ with $x_1, \dots, x_n$ held fixed gives the **likelihood**, and its logarithm the **log-likelihood**,

$$\ell(\theta) = \log f(x_1, \dots, x_n \mid \theta) = \log \prod_{i=1}^{n} f(x_i \mid \theta) = \sum_{i=1}^{n} \log f(x_i \mid \theta). \tag{1.2}$$

The maximum likelihood estimator (MLE) is the value of θ that makes the observed data most probable under the model,

$$\hat{\theta}_{\text{MLE}} = \arg\max_{\theta} \ell(\theta). \tag{1.3}$$

A handful of textbook models — the normal mean with known variance, the binomial proportion — admit a closed-form maximizer found by setting a derivative to zero and solving algebraically. The moment a model becomes even mildly realistic — a generalized linear model, a mixed-effects model, a survival model with censoring — the score equation $\ell'(\theta) = 0$ no longer has an algebraic solution, and $\hat{\theta}_{\text{MLE}}$ must be located by a numerical search. Locating the maximum of a function without a formula for where it is, is **optimization**, the third computational primitive of the course.

1.1.5 Hypothesis Testing

Testing $H_0 : \mu = \mu_0$ against $H_1 : \mu \neq \mu_0$ proceeds by forming a test statistic, such as

$$T = \frac{\bar{x} - \mu_0}{S/\sqrt{n}}, \tag{1.4}$$

and asking how extreme the observed value of T is if H_0 were true. That question can only be answered by knowing the **null distribution** of T, the distribution T would follow under repeated sampling if H_0 held exactly. For data that really are normal, T follows a t_{n-1} distribution exactly, a fact discovered algebraically a century ago. For almost any other kind of data — counts, proportions, skewed measurements, a statistic more complicated than a studentized mean — the null distribution of T is known only approximately, through large-sample theory, or not at all in closed form.

1.1.6 p-value, Type I Error, and Power

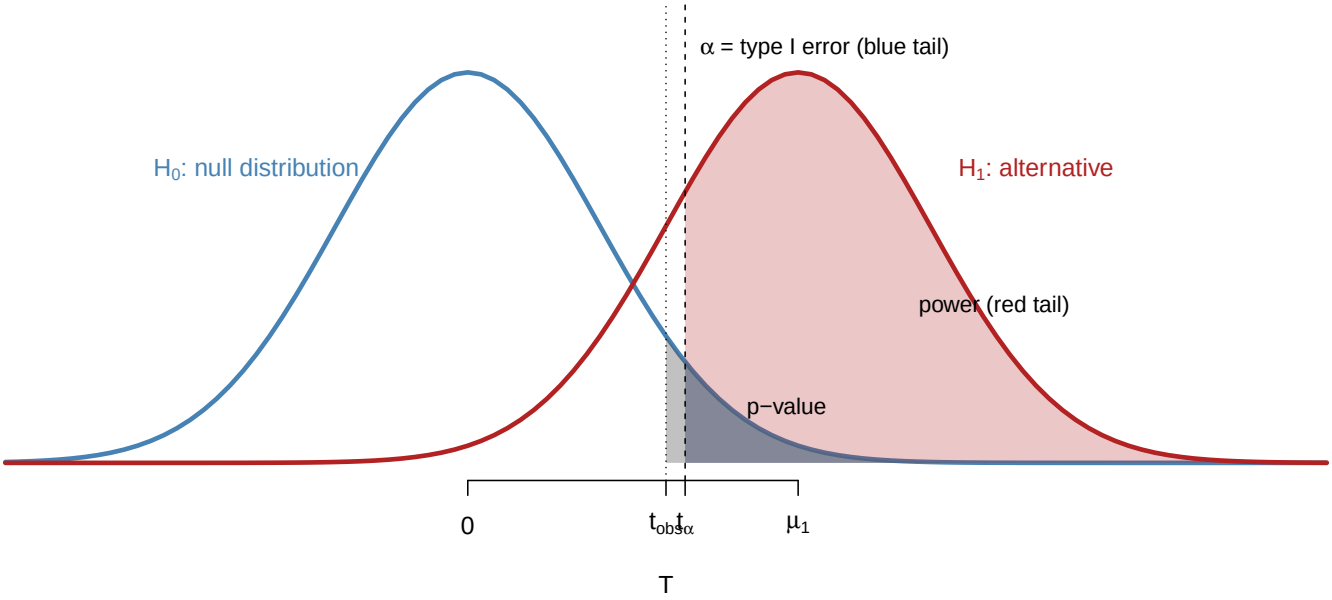


Figure 1.2: Shows how the error rates and the p-value of a hypothesis test all arise from the null and alternative distributions of a single test statistic. Null (H_0) and alternative (H_1) distributions of a test statistic T, with the type I error, power, and p-value shown as tail areas.

Every quantity used to judge a hypothesis test is a tail probability computed from one of these two curves:

Quantity	Definition	Distribution used
p-value	$P_{H_0}(T \geq t_{\text{obs}})$: probability, under H_0, of a statistic at least as extreme as observed	null
Type I error α	$P_{H_0}(T \geq t_\alpha)$: probability of rejecting H_0 when it is true	null
Power	$P_{H_1}(T \geq t_\alpha)$: probability of correctly rejecting H_0 when H_1 holds	alternative

Computing any of these three numbers requires the same thing: the ability to evaluate a tail area of a distribution that, outside of the normal-theory special cases, is not available in closed form. Simulating the null and alternative distributions by repeated sampling — the same Monte Carlo idea used to study sampling distributions above — is often the most direct route when an exact formula does not exist.

1.1.7 Bayesian Inference

The Bayesian approach treats θ itself as random, assigns it a **prior** distribution $\pi(\theta)$ encoding what is known before the data arrive, and updates that belief with the likelihood via Bayes' theorem to obtain the **posterior**,

$$p(\theta \mid x) = \frac{f(x \mid \theta)\,\pi(\theta)}{\int f(x \mid \theta)\,\pi(\theta)\,d\theta}. \tag{1.5}$$

The numerator is easy: it is just the likelihood times the prior, both of which are specified. The denominator — the **marginal likelihood** or **normalizing constant** — is an integral over the entire parameter space, and for anything beyond a handful of conjugate textbook models, that integral has no closed form, especially once θ has more than one or two components.

1.1.8 Prior, Likelihood, and Posterior

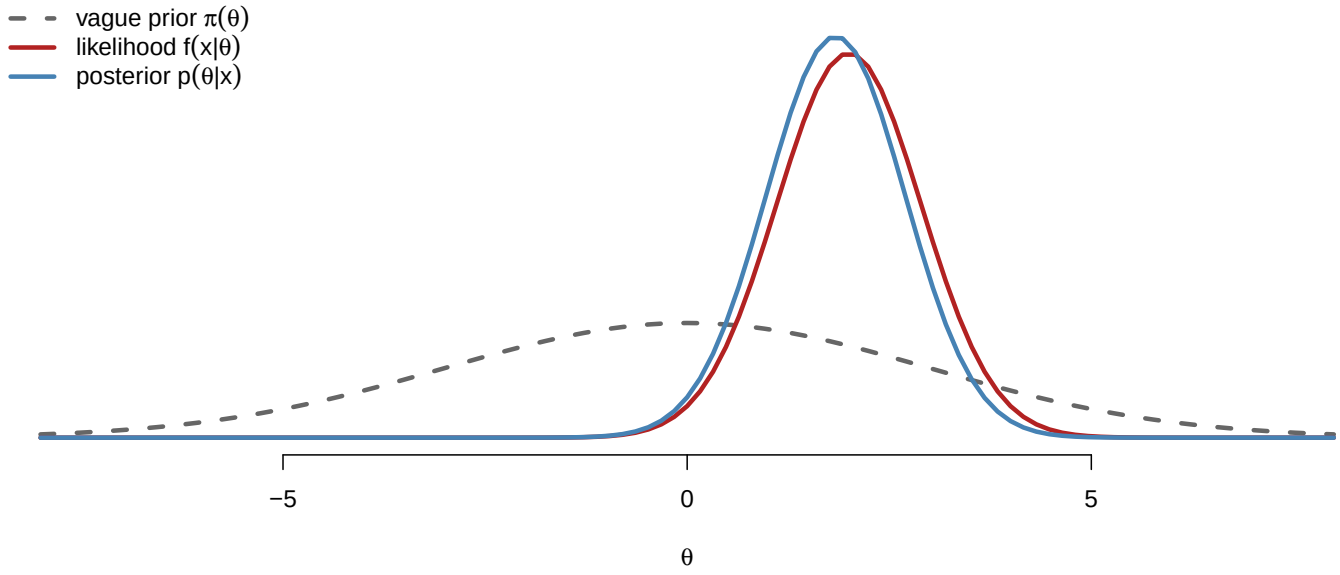


Figure 1.3: Previews Bayesian updating, in which the posterior combines the prior with the likelihood, here in a case where the data dominate the prior. A vague prior, the likelihood, and the resulting posterior for a normal mean: the posterior sits close to the likelihood, slightly narrower and pulled a little toward the prior mean.

In this normal-mean example the posterior can be written down exactly, and the picture makes the mechanism visible: a vague (wide, nearly flat) prior lets the likelihood dominate, so the posterior sits close to the likelihood, just slightly narrower and pulled a little toward the prior mean. Once the answer is available, inference reports posterior summaries — the posterior mean $E(\theta \mid x)$, the posterior variance $V(\theta \mid x)$, a credible interval, or a posterior probability $P(\theta \in H_0 \mid x)$ — and every one of these summaries is, again, an integral against $p(\theta \mid x)$. When the posterior itself has no closed form, none of its summaries do either.

1.1.9 Summary of the Three Tasks

Task	Object needed	Typical difficulty
Point estimation	Sampling distribution of $\hat{\theta}$; maximizer of $\ell(\theta)$	No closed form
Hypothesis testing	Null distribution of T	Unknown for general data
Bayesian inference	$\int f(x \mid \theta)\pi(\theta)\, d\theta$; $E(\theta \mid x)$	High-dimensional integral

Read across the rows, and a pattern emerges: each of the three classical tasks of statistical inference reduces to a computational problem — **calculating** a quantity accurately, **simulating** from a distribution that cannot be sampled directly, **optimizing** a function with no algebraic maximizer, or **integrating** over a parameter space with no closed-form antiderivative. None of these problems is solved by more statistical theory alone; all four are solved by numerical methods, which is what the rest of this course builds.

1.2 The Use of Computers in Statistics

The four computational tasks identified above — calculating, simulating, optimizing, integrating — organize the whole course. This section previews each one with a small example, and points ahead to the unit where it is developed properly.

1.2.1 1. Manipulating Numbers

Even simple calculations are not as trivial on a computer as they look on paper:

- $1.1 + 2.1 + \cdots + 100.1$
- $S^2 = \sum_{i=1}^{n} (x_i - \bar{x})^2/(n-1)$
- $\prod_{i=1}^{n} f(x_i \mid \theta)$ for large n

A computer stores numbers with finite precision, so arithmetic that is exact on paper can go wrong in three specific ways: **overflow**, when a result is too large to represent and becomes `Inf`; **underflow**, when a result is too small and is rounded to zero; and **rounding error**, the gradual loss of significant digits as many operations accumulate. All three are silent — R will not raise an error — so a computation can return a plausible-looking but wrong number without any warning.

```
x <- c(1e10 + 1, 1e10 + 2, 1e10 + 3)
var(x)                                  # correct answer: 1
```

```
[1] 1
```

```
sum(x^2)/2 - 3 * mean(x)^2 / 2          # textbook formula
```

```
[1] 0
```

The two lines above compute the same variance by two algebraically equivalent formulas; they disagree because the second formula first squares numbers near 10^{10} and then subtracts two similarly huge quantities, destroying most of the precision in the answer — a textbook illustration of **catastrophic cancellation**.

```
prod(dnorm(rnorm(1000)))                  # likelihood underflows to 0
```

```
[1] 0
```

```
sum(dnorm(rnorm(1000), log = TRUE))       # log-likelihood is fine
```

```
[1] -1459.311
```

Multiplying a thousand densities together underflows to exactly zero, even though the true likelihood is a small but perfectly well-defined positive number; summing the log-densities instead avoids the problem entirely, which is exactly why $\ell(\theta) = \sum_i \log f(x_i \mid \theta)$, and not $L(\theta) = \prod_i f(x_i \mid \theta)$, is the quantity every optimizer in this course actually works with. The unit on computer arithmetic develops these failure modes systematically — how floating-point numbers are stored, when each failure occurs, and the standard tricks (working on the log scale, the log-sum-exp identity, centering and scaling) that avoid them.

1.2.2 2. Simulation with Random Numbers

Monte Carlo methods replace a probability calculation that has no closed form with a large number of repeated random experiments, summarized numerically. The simplest possible illustration: if X_1, X_2 are independent Unif$(0, 1)$ random variables, what is the distribution of $X_1 + X_2$? It can be worked out by hand (it is triangular on $[0, 2]$), but it can also just be *seen*, by generating many pairs and plotting a histogram of their sums.

```
z <- runif(10000) + runif(10000)
hist(z, breaks = 40, freq = FALSE, main = "", xlab = "X1 + X2")
```

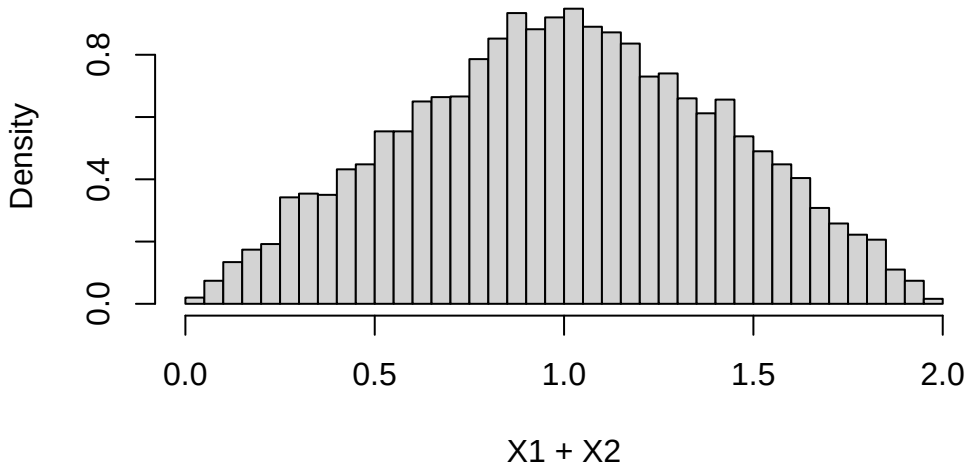


Figure 1.4: Demonstrates Monte Carlo simulation as a way of approximating a distribution, using an example whose exact answer is known so that the approximation can be checked. Simulated distribution of $X_1 + X_2$ for independent Unif$(0, 1)$ variables, from 10,000 draws: a Monte Carlo approximation to the (triangular) true density.

The same idea scales up directly to the sampling-distribution question from earlier in this chapter. Repeating "draw $x_1, \ldots, x_n$ from the assumed model, then compute the statistic" many times over gives an empirical approximation to the sampling distribution of any estimator or test statistic — $\hat{\mu}^{(1)}$ versus $\hat{\mu}^{(2)}$, the null distribution of T for non-normal data, the actual coverage of a nominal 95% confidence interval — without ever deriving a formula for it. The unit on random numbers and Monte Carlo develops this systematically: how pseudo-random numbers are generated, how draws from one distribution are transformed into draws from another, the Law of Large Numbers and Central Limit Theorem that justify treating a simulation average as an estimate, and how to use simulation to evaluate estimators and tests.

1.2.3 3. Optimization

The MLE problem above is one instance of a completely general task: find $x_0 = \arg\max_x f(x)$.

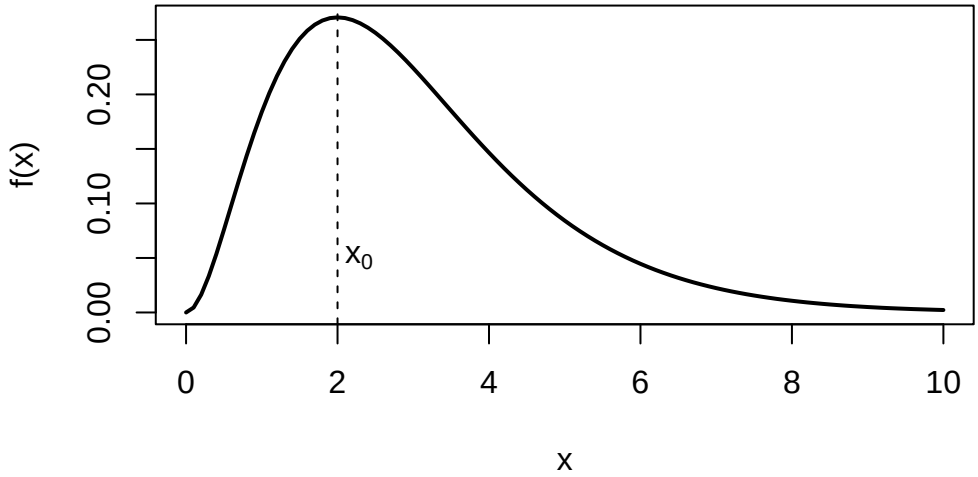


Figure 1.5: A one-dimensional warm-up for numerical optimization, showing what it means to maximize a function before moving to problems that cannot be plotted. A Gamma(3,1) density: its maximizer x_0 is obvious by eye, unlike a multi-parameter log-likelihood.

For a gamma density plotted over its whole domain, as above, the maximizer is obvious by eye. For a log-likelihood $\ell(\theta)$ with many parameters, plotting is impossible and "obvious by eye" is not an option: the maximum has to be found by an iterative numerical search that starts somewhere and moves uphill. The unit on MLE and optimization develops these searches in order of increasing generality — bisection and golden-section search in one dimension, Newton–Raphson and Fisher scoring using derivative information, and gradient-based methods (BFGS, stochastic gradient descent) for high-dimensional problems where forming a Hessian is too expensive.

1.2.4 4. Advanced Simulation and Integration

The quantities that summarize a distribution are integrals:

$$E(X) = \int x f(x)\, dx, \qquad V(X), \qquad \int f(x \mid \theta)\pi(\theta)\, d\theta. \tag{1.6}$$

Two broad strategies approximate an integral that has no closed form. **Numerical quadrature** approximates the area under the integrand by a sum of rectangles, trapezoids, or higher-order pieces; it is accurate and efficient in one or two dimensions, but its cost grows exponentially with the number of dimensions, which makes it impractical beyond a handful of parameters. **Monte Carlo integration** instead draws random points from (or related to) f and averages, trading the deterministic accuracy of quadrature for a convergence rate that does not depend on dimension — the only practical route once θ has more than a few components. For a Bayesian posterior $p(\theta \mid x)$, direct sampling is typically impossible because the normalizing constant is unknown, which is precisely the setting where **Markov chain Monte Carlo** (MCMC) is used: a Markov chain is constructed whose stationary distribution is the posterior itself, so that simulating the chain for long enough produces (dependent) draws from $p(\theta \mid x)$ without ever having to evaluate the integral in its denominator.

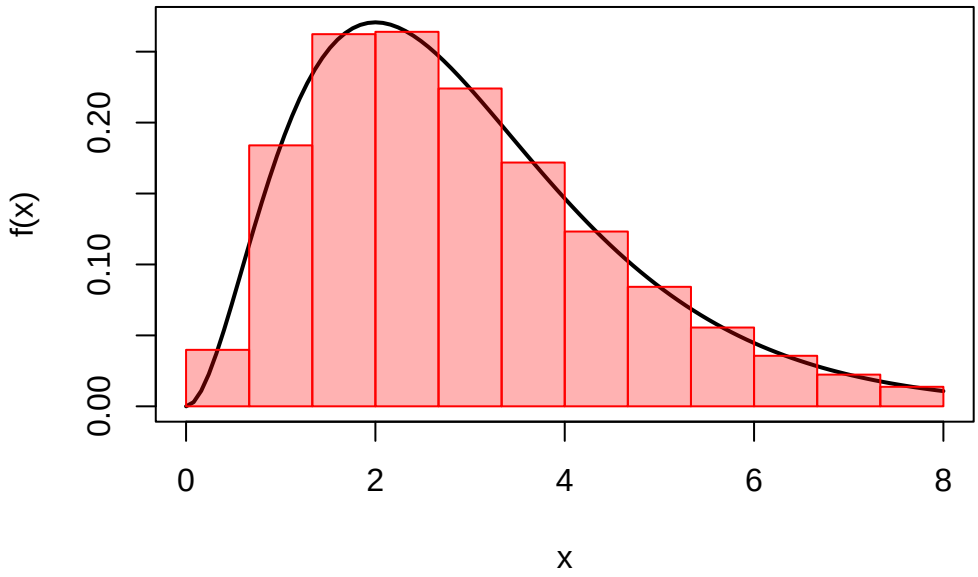


Figure 1.6: Introduces numerical quadrature, the deterministic alternative to Monte Carlo for computing integrals, by turning the area under a density into a sum of rectangle areas. Numerical quadrature by rectangle rule: approximating the area under a Gamma(3,1) density with 12 midpoint rectangles.

The rectangles above illustrate the quadrature idea directly:

$$\int_a^b f(x)\,dx \approx \sum_{j=1}^{k} f(m_j)\,\Delta x, \tag{1.7}$$

where m_j is the midpoint of the jth rectangle of width Δx. The later units on numerical quadrature, Laplace approximation, rejection and importance sampling, and Markov chain Monte Carlo develop this strategy and its Monte Carlo counterpart from the ground up, culminating in the algorithms that make modern Bayesian computation possible.

1.2.5 Course Roadmap

Topic	Statistical task
Computer arithmetic	Reliable calculation of likelihoods and variances
Random numbers and Monte Carlo	Sampling distributions, evaluating estimators and tests
MLE and optimization	Point estimation without closed forms
Numerical integration and MCMC	Bayesian inference

Throughout the course, three themes recur alongside the statistical content: programming in R, writing vectorized code instead of explicit loops, and paying attention to the efficiency and numerical stability of the code itself — since, as the arithmetic examples above already show, a correct formula implemented carelessly can still give a wrong answer.

2 Introduction to R and Quarto

2.1 A toy example

Throughout this unit we will use one running example to introduce the basic building blocks of R: vectors, strings, special values, matrices, data frames, lists, functions, file I/O, and graphics.

The mission: a class of 100 students has taken graded work in three *categories*: five quizzes, two tests, and one final exam, each scored out of 100 points. A student's **course score** is a weighted average of the category means – quizzes count 30%, tests count 30%, and the final counts 40%. To make the simulated data realistic, a student's test and final scores are generated from their quiz performance by a simple linear regression, with a different intercept for Program A and Program B students, and a few quizzes are missing entirely (a student who did not sit that quiz). We want to

1. work out the calculation step by step for one student,
2. package that calculation into a reusable function,
3. apply the function to every student to get a full set of results, and
4. present those results as a table and as plots.

This mirrors how real data-analysis code is usually built: explore on one case, generalize into a function, then apply and report.

2.2 Generating an artificial score dataset

2.2.1 Vectors: recording quiz scores

Each quiz, across all students, is naturally stored as a vector. We simulate a class of 100 students, five quizzes scored out of 100, using `rnorm` and clipping the result to a valid `0-100` range with `pmin`/`pmax`:

```
set.seed(123)
n <- 100

gen.quiz <- function(mean, sd)
{
    x <- round(rnorm(n, mean, sd))
    pmin(100, pmax(0, x))
}

Quiz1 <- gen.quiz(75, 20)
Quiz2 <- gen.quiz(78, 20)
```

```r
Quiz3 <- gen.quiz(72, 20)
Quiz4 <- gen.quiz(80, 20)
Quiz5 <- gen.quiz(74, 20)

Quiz1
```

```
  [1]  64  70 100  76  78 100  84  50  61  66  99  82  83  77  64 100  85  36
 [19]  89  66  54  71  54  60  62  41  92  78  52 100  84  69  93  93  91  89
 [37]  86  74  69  67  61  71  50 100  99  53  67  66  91  73  80  74  74 100
 [55]  70 100  44  87  77  79  83  65  68  55  54  81  84  76  93 100  65  29
 [73]  95  61  61  96  69  51  79  72  75  83  68  88  71  82  97  84  68  98
 [91]  95  86  80  62 100  63 100 100  70  54
```

2.2.2 Strings: naming the categories and students

We could type out `"Quiz1"`, `"Quiz2"`, … by hand, but `paste0` builds such labels programmatically – a habit that pays off the moment there are 40 quizzes instead of 5:

```r
quiz.names <- paste0("Quiz", 1:5)
test.names <- paste0("Test", 1:2)

quiz.names
```

```
[1] "Quiz1" "Quiz2" "Quiz3" "Quiz4" "Quiz5"
```

```r
test.names
```

```
[1] "Test1" "Test2"
```

With 100 students, typing out individual names is not realistic either. `sprintf` generates zero-padded IDs like `"S001"`, `"S002"`, … in one line – exactly the kind of job it is built for:

```r
students <- sprintf("S%03d", 1:n)

head(students)
```

```
[1] "S001" "S002" "S003" "S004" "S005" "S006"
```

2.2.3 Simulating the tests and the final using vectorized math

In a real class, a student's test and final scores are not independent of how they did on the quizzes. We simulate that by treating each student's **mean quiz score** as a predictor in a simple linear regression.

This is a great place to demonstrate **vectorized calculations** in R. Instead of writing a `for` loop to calculate each student's score one by one, we can simply add and multiply whole vectors of length 100 together. R aligns the elements automatically:

```r
program <- factor(sample(c("Program A", "Program B"), n, replace = TRUE))

quiz.mat <- cbind(Quiz1, Quiz2, Quiz3, Quiz4, Quiz5)

head(quiz.mat)
```

```
     Quiz1 Quiz2 Quiz3 Quiz4 Quiz5
[1,]    64    64   100    66    73
[2,]    70    83    98    65    51
[3,]   100    73    67    61    61
[4,]    76    71    83    59    73
[5,]    78    59    64    71    87
[6,]   100    77    62    87    41
```

```r
head(data.frame(program, quiz.mean.true))
```

```
    program quiz.mean.true
1 Program A           73.4
2 Program B           73.4
3 Program B           72.4
4 Program A           72.4
5 Program B           71.8
6 Program A           73.4
```

First six of the 100 rows; the HTML version of this book shows the full table.

The intercept depends on `program`, so we compute it with `ifelse`; the slope is the same for every student:

```r
head(data.frame(program, b0.test, b1.test))
```

```
    program b0.test b1.test
1 Program A      12     0.8
2 Program B       6     0.8
3 Program B       6     0.8
4 Program A      12     0.8
5 Program B       6     0.8
6 Program A      12     0.8
```

First six of the 100 rows; the HTML version of this book shows the full table.

The final exam uses its own intercept and slope, computed the same way:

```r
head(data.frame(program, b0.final, b1.final))
```

```
    program b0.final b1.final
1 Program A       15     0.85
2 Program B        8     0.85
3 Program B        8     0.85
4 Program A       15     0.85
5 Program B        8     0.85
6 Program A       15     0.85
```

First six of the 100 rows; the HTML version of this book shows the full table.

`clip.round` rounds a score and clips it to the valid `0-100` range – we'll reuse it for every test and the final:

```
clip.round <- function(x) round(pmin(100, pmax(0, x)))
```

Now we add random error to the linear predictor to get each student's raw `Test1` score, then clip and round it:

```
head(data.frame(program, b0.test, b1.test, quiz.mean.true, error, Test1.raw))
```

```
    program b0.test b1.test quiz.mean.true      error Test1.raw
1 Program A      12     0.8           73.4   6.759234  77.47923
2 Program B       6     0.8           73.4  -6.257615  58.46239
3 Program B       6     0.8           72.4   8.885691  72.80569
4 Program A      12     0.8           72.4   1.998598  71.91860
5 Program B       6     0.8           71.8  13.215323  76.65532
6 Program A      12     0.8           73.4 -11.671766  59.04823
```

First six of the 100 rows; the HTML version of this book shows the full table.

```
head(data.frame(program, Test1.raw, Test1))
```

```
    program Test1.raw Test1
1 Program A  77.47923    77
2 Program B  58.46239    58
3 Program B  72.80569    73
4 Program A  71.91860    72
5 Program B  76.65532    77
6 Program A  59.04823    59
```

First six of the 100 rows; the HTML version of this book shows the full table.

Using the above method, we generate two tests and a final:

```
# Vectorized math: b0.test + b1.test * quiz.mean.true + rnorm(...)
Test1 <- clip.round(b0.test  + b1.test*quiz.mean.true  + rnorm(n, 0, 8))
Test2 <- clip.round(b0.test  + b1.test*quiz.mean.true  + rnorm(n, 0, 8))
Final <- clip.round(b0.final + b1.final*quiz.mean.true + rnorm(n, 0, 6))
```

2.2.4 Matrices: combining all the scores (Complete Data)

Before we simulate absences (missing values), let's bind the quiz, test, and final columns together into one complete matrix. It will have one row per student and one column per graded item.

```
scores <- cbind(quiz.mat, Test1, Test2, Final)
colnames(scores) <- c(quiz.names, test.names, "Final")
rownames(scores) <- students

# Here is our complete dataset before any missing values
head(scores)
```

```
     Quiz1 Quiz2 Quiz3 Quiz4 Quiz5 Test1 Test2 Final
S001    64    64   100    66    73    55    83    81
S002    70    83    98    65    51    66    64    70
S003   100    73    67    61    61    69    68    76
S004    76    71    83    59    73    58    72    70
S005    78    59    64    71    87    67    62    83
S006   100    77    62    87    41    64    70    74
```

2.2.5 Adding and handling missing values

In reality, students occasionally miss a quiz. Let's randomly drop about 5% of the quiz scores by assigning them the special value `NA` (Not Available).

```
miss.rate <- 0.05
sample(n, round(n*miss.rate))
```

```
[1] 31 35 73 49 38
```

```
# We only want to introduce NAs into the first 5 columns (the quizzes)
for(j in 1:ncol(scores)) {
    scores[sample(n, round(n*miss.rate)), j] <- NA
}

head(scores, 10)
```

```
     Quiz1 Quiz2 Quiz3 Quiz4 Quiz5 Test1 Test2 Final
S001    64    64   100    66    73    55    83    81
S002    70    83    98    65    51    66    64    70
S003   100    73    67    61    61    69    68    76
S004    76    71    83    59    73    58    NA    70
S005    78    59    64    NA    87    67    62    83
S006   100    77    62    87    41    64    70    74
S007    84    62    56    40    67    70    70    59
S008    50    45    NA    84    89    63    57    62
S009    61    70   100    NA    63    75    53    76
S010    66    96    71   100    79    79    76    95
```

`is.na` locates missing values anywhere in an object. Only the quiz columns have any, since we generated `Test1`, `Test2`, and `Final` without missingness:

```
colSums(is.na(scores))
```

```
Quiz1 Quiz2 Quiz3 Quiz4 Quiz5 Test1 Test2 Final
    5     5     5     5     5     5     5     5
```

We will rely on `na.rm = TRUE` later so that a missing quiz simply drops out of a student's quiz average, rather than returning `NA` for their whole course score.

2.2.6 Data frames: adding names and program

A data frame lets us combine columns of different types – here, the students' names and program alongside their (numeric) scores – into a single table:

```
head(roster)
```

```
     name   program Quiz1 Quiz2 Quiz3 Quiz4 Quiz5 Test1 Test2 Final
S001 S001 Program A    64    64   100    66    73    55    83    81
S002 S002 Program B    70    83    98    65    51    66    64    70
S003 S003 Program B   100    73    67    61    61    69    68    76
S004 S004 Program A    76    71    83    59    73    58    NA    70
S005 S005 Program B    78    59    64    NA    87    67    62    83
S006 S006 Program A   100    77    62    87    41    64    70    74
```

First six of the 100 rows; the HTML version of this book shows the full table.

2.3 Calculating a Single Student's Score

Before writing a function, it often helps to work through the logic step by step on a single example. Let's find a student who missed a quiz so we can see how missing values are handled, and extract their record:

```
# Find the index of the first student who has at least one NA in columns 1 to 5
example.ix <- which(rowSums(is.na(scores[, 1:5])) > 0)[1]
student_record <- roster[example.ix, ]

student_record
```

```
     name   program Quiz1 Quiz2 Quiz3 Quiz4 Quiz5 Test1 Test2 Final
S005 S005 Program B    78    59    64    NA    87    67    62    83
```

To avoid hardcoding column positions, we can extract the column names from this record and use `grep` to dynamically search for columns starting with "Quiz", "Test", or "Final":

```
# Extract all the column names from the student record
col_names <- names(student_record)
col_names
```

```
 [1] "name"    "program" "Quiz1"   "Quiz2"   "Quiz3"   "Quiz4"   "Quiz5"
 [8] "Test1"   "Test2"   "Final"
```

```
# Use grep to find the numeric indices of specific categories
grep("^Quiz", col_names, value = TRUE)
```

```
[1] "Quiz1" "Quiz2" "Quiz3" "Quiz4" "Quiz5"
```

```
quiz.cols <- grep("^Quiz", col_names)
quiz.cols
```

```
[1] 3 4 5 6 7
```

```
test.cols <- grep("^Test", col_names)
final.col <- grep("^Final", col_names)
```

```
# Check which columns grep found for the quizzes
quiz.cols
```

```
[1] 3 4 5 6 7
```

Next, we extract the scores using those indices, convert them to numeric values, and calculate the mean for each category. Using `na.rm = TRUE` ensures that the missing quiz is simply ignored rather than turning the whole average into `NA`:

```
# Calculate the mean for each category, ignoring NAs
quiz.mean <- mean(as.numeric(student_record[quiz.cols]), na.rm = TRUE)
test.mean <- mean(as.numeric(student_record[test.cols]), na.rm = TRUE)
final.score <- as.numeric(student_record[final.col])

c(Quiz = quiz.mean, Test = test.mean, Final = final.score)
```

```
 Quiz  Test Final
 72.0  64.5  83.0
```

Finally, we apply the syllabus weights to calculate the overall course score:

```
# Calculate the weighted course score
course.score <- (0.3 * quiz.mean) + (0.3 * test.mean) + (0.4 * final.score)

course.score
```

```
[1] 74.15
```

2.3.1 Aggregating into a Reusable Function

Now that we know the logic works perfectly for one student, we can package those exact steps into a reusable function. This function will take any `student_record` as its input and return a named vector containing the category means and the final score.

To make the grading policy more forgiving, we are adding a conditional rule: if a student's final exam score is strictly higher than their test average, the final exam score will replace the test average. Because `compute.course.score` processes exactly one student at a time, we evaluate a single logical condition using a standard `if` statement rather than a vectorized alternative like `ifelse`. We also use `!is.na()` checks to ensure the `if` statement doesn't crash by attempting to evaluate a missing value (which could happen if a student had no test or final scores).

```r
compute.course.score <- function(student_record, w_quiz = 0.3, w_test = 0.3, w_final = 0.4) {

    col_names <- names(student_record)
    quiz.cols <- grep("^Quiz", col_names)
    test.cols <- grep("^Test", col_names)
    final.col <- grep("^Final", col_names)

    quiz.mean <- mean(as.numeric(student_record[quiz.cols]), na.rm = TRUE)
    test.mean <- mean(as.numeric(student_record[test.cols]), na.rm = TRUE)
    final.score <- as.numeric(student_record[final.col])

    # Explicit conditional check: replace test mean with final score if it is higher
    if (!is.na(final.score) && !is.na(test.mean) && final.score > test.mean) {
        test.mean <- final.score
    }

    # The weights are now dynamic variables rather than hardcoded numbers
    course.score <- (w_quiz * quiz.mean) + (w_test * test.mean) + (w_final * final.score)

    c(Quiz = quiz.mean, Test = test.mean, Final = final.score, course.score = course.score)
}
```

Let's test our updated function on two specific scenarios to verify the conditional logic. We will format these inputs as **named vectors** rather than data frames. This mimics exactly what the `apply()` function will do later when it passes rows of our dataset to this function.

Student A struggled on the tests (average: 70) but aced the final (95). The function should detect that `95 > 70` and replace their test mean with 95 before calculating the final course score. **Student B** did great on the tests (average: 90) but dropped on the final (75). The function should keep their original test mean.

```r
# Create the mock record for Student A as a named vector
student_A <- c(
    name = "Student_A", program = "Program A",
    Quiz1 = "80", Quiz2 = "80", Quiz3 = "80", Quiz4 = "80", Quiz5 = "80", # Quiz mean = 80
    Test1 = "65", Test2 = "75",                                           # Test mean = 70
    Final = "95"
)

# Create the mock record for Student B as a named vector
student_B <- c(
    name = "Student_B", program = "Program B",
    Quiz1 = "80", Quiz2 = "80", Quiz3 = "80", Quiz4 = "80", Quiz5 = "80", # Quiz mean = 80
    Test1 = "90", Test2 = "90",                                           # Test mean = 90
    Final = "75"
)

# Test Student A: The Test mean should be updated to 95 in the output
cat("Student A Results:\n")
```

```
Student A Results:
```

```
compute.course.score(student_A)
```

```
       Quiz        Test       Final course.score
       80.0        95.0        95.0         90.5
```

```
# Math check: (0.3 * 80) + (0.3 * 95) + (0.4 * 95) = 90.5

# Test Student B: The Test mean should remain 90 in the output
cat("\nStudent B Results:\n")
```

```
Student B Results:
```

```
compute.course.score(student_B)
```

```
       Quiz        Test       Final course.score
         80          90          75           81
```

```
# Math check: (0.3 * 80) + (0.3 * 90) + (0.4 * 75) = 81
```

2.4 Applying the Function to Every Student

2.4.1 Using `apply` function

Now that we have a function working for one student, we can apply it across the entire class. The `apply` function is perfect for this: it takes our `roster` data frame, loops over either rows (`1`) or columns (`2`), and executes our custom function.

```
head(results)
```

```
     name   program Quiz Test Final course.score
S001 S001 Program A 73.4   81    81        78.72
S002 S002 Program B 73.4   70    70        71.02
S003 S003 Program B 72.4   76    76        74.92
S004 S004 Program A 72.4   70    70        70.72
S005 S005 Program B 72.0   83    83        79.70
S006 S006 Program A 73.4   74    74        73.82
```

First six of the 100 rows; the HTML version of this book shows the full table.

2.4.2 Looping the rows with `for`

While `apply()` is idiomatic R and very concise, we can also use a traditional `for` loop to process each student. To do this efficiently, we first pre-allocate an empty matrix of the correct size to store our output. Then, we iterate through the row indices one by one.

Inside the loop, we use `unlist()` to convert the single row of our data frame into a named vector. This guarantees we are feeding our `compute.course.score` function the exact same data structure that `apply()` uses under the hood.

```
# 1. Pre-allocate an empty matrix to hold the results
num_students <- nrow(roster)
loop_results <- matrix(NA, nrow = num_students, ncol = 4)

# Set the column and row names to match our expected output
colnames(loop_results) <- c("Quiz", "Test", "Final", "course.score")
rownames(loop_results) <- roster$name

# 2. Iterate over each row index
for (i in 1:num_students) {
    # Extract the single row as a named character vector
    student_record <- unlist(roster[i, ])

    # Compute the score and store it in the corresponding row of our matrix
    loop_results[i, ] <- compute.course.score(student_record)
}

# Peek at the first few rows of the loop results
head(loop_results)
```

```
     Quiz Test Final course.score
S001 73.4   81    81        78.72
S002 73.4   70    70        71.02
S003 72.4   76    76        74.92
S004 72.4   70    70        70.72
S005 72.0   83    83        79.70
S006 73.4   74    74        73.82
```

Now, let's run the exact same calculation using `apply()` and use the `all.equal()` function to verify that both methods yield the exact same mathematical results.

```
# 3. Check if they are mathematically identical
# (We set check.attributes = FALSE to ignore minor metadata differences,
# like how apply() inherits row names vs. how we manually assigned them)
is_same <- all.equal(loop_results, results, check.attributes = FALSE)

print(paste("Do the for loop and apply() give the exact same results?", is_same))
```

```
[1] "Do the for loop and apply() give the exact same results? target is matrix, current is data.frame"
```

2.4.3 Saving and reloading the results

The `results` table is worth keeping. A data frame can be written as a plain csv file for spreadsheets, or saved as an `.RDS` file to preserve it exactly (including the `program` factor) for future R sessions:

```
head(read.csv("results.csv"))
```

```
  name   program Quiz Test Final course.score
1 S001 Program A 73.4   81    81        78.72
2 S002 Program B 73.4   70    70        71.02
3 S003 Program B 72.4   76    76        74.92
4 S004 Program A 72.4   70    70        70.72
5 S005 Program B 72.0   83    83        79.70
6 S006 Program A 73.4   74    74        73.82
```

```
identical(results, results.reloaded)
```

```
[1] TRUE
```

First six of the 100 rows; the HTML version of this book shows the full table.

2.5 Writing a Function for the Whole Process with List Output

Sometimes a function needs to return multiple distinct pieces of information that don't fit neatly into a single table. A **list** in R is a flexible container that can bundle objects of completely different types and sizes together.

Let's write a wrapper function that takes our raw `roster` data and our weighting scheme. It will calculate the scores, but instead of just returning the table, it will return a list containing four very different items:

1. The full processed data frame.
2. A numeric vector of just the final course scores.
3. A frequency table of scores binned into 10-point intervals.
4. A character vector of the names of students who failed (scored below 60).

```
analyze_class_performance <- function(roster_data, w_quiz = 0.3, w_test = 0.3, w_final = 0.4) {

    # 1. Pass the weights down into compute.course.score
    all.results <- t(apply(roster_data, 1, compute.course.score,
                           w_quiz = w_quiz, w_test = w_test, w_final = w_final))

    processed_table <- cbind(roster_data[, c("name", "program")], all.results)

    # 2. Extract final course scores
    final_scores <- processed_table$course.score

    # 3. Create bins (handling NAs)
    bins <- seq(0, 100, by = 10)
    score_bins <- table(cut(final_scores, breaks = bins, include.lowest = TRUE), useNA =
↪   "ifany")
```

```
    # 4. Extract names of failing students (safely ignoring NAs using which)
    failing_students <- processed_table$name[which(final_scores < 60)]

    # 5. Extract names of students with missing overall scores
    incomplete_students <- processed_table$name[is.na(final_scores)]

    # Bundle everything into a named list
    list(
        ProcessedTable = processed_table,
        CourseScores = final_scores,
        ScoreDistribution = score_bins,
        Failing_Students = failing_students,
        incomplete_students = incomplete_students
    )
}
```

When we run this function, it returns a single list object. We can use the `str()` function to quickly inspect the structure of our new list and see all the different data types nested inside it:

```
# Run the analysis
class_report <- analyze_class_performance(roster)

# Inspect the structure of the returned list
str(class_report, max.level = 1)
```

```
List of 5
 $ ProcessedTable     :'data.frame':    100 obs. of  6 variables:
 $ CourseScores       : num [1:100] 78.7 71 74.9 70.7 79.7 ...
 $ ScoreDistribution  : 'table' int [1:11(1d)] 0 0 0 0 0 1 21 42 28 3 ...
  ..- attr(*, "dimnames")=List of 1
 $ Failing_Students   : chr "S094"
 $ incomplete_students: chr [1:5] "S026" "S027" "S039" "S070" ...
```

To access individual elements inside a list, you use the `$` operator (just like with data frames) or double brackets `[[]]`. Let's look at the score distribution and the failing students:

```
# Access the 10-point bin distribution
class_report$ScoreDistribution
```

```
  [0,10]  (10,20]  (20,30]  (30,40]  (40,50]  (50,60]  (60,70]  (70,80]
       0        0        0        0        0        1       21       42
 (80,90] (90,100]     <NA>
      28        3        5
```

```
# Access the names of the failing students
class_report$Failing_Students
```

```
[1] "S094"
```

```
class_report$incomplete_students
```

```
[1] "S026" "S027" "S039" "S070" "S096"
```

2.6 The use of pipe |>

The pipe operator `x |> f(...)` is just shorthand for `f(x, ...)` – it takes whatever is on its left and inserts it as the *first* argument of the function call on its right. This lets you write a sequence of transformations in the order you think of them, instead of nesting calls inside out:

```
# Without the pipe: read from the inside out
round(sqrt(mean(c(4, 9, 16, 25))), 2)
```

```
[1] 3.67
```

```
# With the pipe: read top to bottom, left to right
c(4, 9, 16, 25) |>
    mean() |>
    sqrt() |>
    round(2)
```

```
[1] 3.67
```

Each step's output becomes the next step's first input, so `mean()` receives the vector, `sqrt()` receives the mean, and `round(2)` receives the square root (with 2 filling the *second* argument, `digits`). Extra arguments after the first are passed straight through exactly as they would be in a normal call, which matters once a pipeline mixes several multi-argument functions – much like the quiz-generating recipe used throughout this unit:

```
set.seed(123)

# Same idea as gen.quiz(), written as one pipeline: each step's output
# fills the *first* argument of the next call, while the remaining,
# named arguments are supplied just as in an ordinary function call
rnorm(20, mean = 75, sd = 10) |>
    round(digits = 0) |>       # round(x, digits = 0)
    pmin(100) |>                # pmin(x, 100)
    pmax(0) |>                  # pmax(x, 0)
    sort(decreasing = TRUE) |>  # sort(x, decreasing = TRUE)
    head(n = 5)                 # head(x, n = 5)
```

```
[1] 93 92 91 87 82
```

Sometimes the piped value needs to land in an argument *other than* the first. The placeholder _ marks exactly where it goes, as long as that argument is named:

```r
# seq(1, to = 10, by = 2) -- the piped 10 fills the "to" argument
10 |> seq(1, to = _, by = 2)
```

```
[1] 1 3 5 7 9
```

This same idea – chaining verbs with `|>` – is what makes the dplyr pipelines below read like a recipe.

2.7 Data Manipulation with `dplyr`

Base R's indexing, `apply()`, and `for` loops can do everything we've done so far, but **dplyr** offers a grammar of a handful of verbs – `mutate()`, `filter()`, `select()`, `arrange()`, `group_by()`, `summarize()` – that are chained together with the pipe `|>`, so a whole data-wrangling recipe reads top to bottom like a sentence instead of nesting function calls inside one another.

2.7.1 Generating a roster with `mutate()`

`mutate()` isn't limited to computing new columns from data that already exists in a data frame – it can build up a data frame from nothing at all, one column at a time, chaining together exactly the same generative steps we used in Generating an artificial score dataset. Within a single `mutate()` call, each new column is available to the expressions that follow it, so we can go straight from raw quiz scores to `quiz.mean.true` to `Test1`/`Test2`/`Final`, all in one pipeline, reusing the `gen.quiz()` and `clip.round()` helpers we already defined:

```r
library(dplyr)

set.seed(2026)   # a fresh cohort, just for this demonstration

n2 <- 100
students2 <- sprintf("S%03d", 1:n2)
program2  <- factor(sample(c("Program A", "Program B"), n2, replace = TRUE))

data.frame(name = students2, program = program2) |>
    mutate(
        Quiz1 = gen.quiz(75, 20),
        Quiz2 = gen.quiz(78, 20),
        Quiz3 = gen.quiz(72, 20),
        Quiz4 = gen.quiz(80, 20),
        Quiz5 = gen.quiz(74, 20),
        quiz.mean.true = rowMeans(cbind(Quiz1, Quiz2, Quiz3, Quiz4, Quiz5)),
        b0.test  = if_else(program == "Program A", 12, 6),
        b0.final = if_else(program == "Program A", 15, 8),
        Test1 = clip.round(b0.test  + 0.80 * quiz.mean.true + rnorm(n(), 0, 8)),
        Test2 = clip.round(b0.test  + 0.80 * quiz.mean.true + rnorm(n(), 0, 8)),
        Final = clip.round(b0.final + 0.85 * quiz.mean.true + rnorm(n(), 0, 6)),
        across(c(starts_with("Quiz"), starts_with("Test"), Final),
               ~ replace(.x, sample(n(), round(n() * miss.rate)), NA))
    ) -> roster_worktable
```

```
roster_worktable |> slice_head (n = 6)
```

```
  name   program Quiz1 Quiz2 Quiz3 Quiz4 Quiz5 quiz.mean.true b0.test b0.final
1 S001 Program A    52    78    66    NA    84           76.0      12       15
2 S002 Program A    71    NA    86    NA    97             NA      12       15
3 S003 Program A    56    98    64   100    80           79.6      12       15
4 S004 Program B    84    88    70    65    75           76.4       6        8
5 S005 Program A    62    90   100    47   100           79.8      12       15
6 S006 Program A    70   100    59    78    88           79.0      12       15
  Test1 Test2 Final
1    80    75    82
2    78    79    84
3    80    77    85
4    65    60    72
5    70    78    82
6    87    86    73
```

```
roster_mutate <- roster_worktable |> select(-quiz.mean.true, -b0.test, -b0.final)

roster_mutate |> slice_head (n= 6)
```

```
  name   program Quiz1 Quiz2 Quiz3 Quiz4 Quiz5 Test1 Test2 Final
1 S001 Program A    52    78    66    NA    84    80    75    82
2 S002 Program A    71    NA    86    NA    97    78    79    84
3 S003 Program A    56    98    64   100    80    80    77    85
4 S004 Program B    84    88    70    65    75    65    60    72
5 S005 Program A    62    90   100    47   100    70    78    82
6 S006 Program A    70   100    59    78    88    87    86    73
```

A few things are worth noticing here. `b0.test` and `b0.final` are computed with `if_else()` – the vectorized, type-checked dplyr analogue of `ifelse()` – directly from the `program` column, and then used (and finally dropped again with `select(-...)`) just like the helper vectors from the original script. `n()`, dplyr's built-in count of the current number of rows, replaces the standalone `n <- 100` we relied on earlier, so this pipeline needs nothing from the outside except `students2`/`program2` to start from. The tidyselect helper `starts_with()` – the dplyr analogue of the `grep("^Quiz", ...)` trick from earlier – picks out groups of columns by name, and wrapping it in `across()` applies the *same* NA-injection rule to all eight score columns at once, in place of the explicit `for` loop over `1:ncol(scores)`.

Because this pipeline draws its own random numbers with a fresh seed, `roster_mutate` is a different simulated class from `roster` – not a reproduction of it – but it is built by *exactly* the same recipe, entirely inside one `mutate()` chain. The rest of this section goes back to analyzing the original `roster`.

2.7.2 Recomputing course scores with `mutate()` and `rowwise()`

Our course-score calculation is inherently a **row-wise** operation: for each student we need to look across several quiz columns and average just that student's values. `rowwise()` tells dplyr to treat each row as its own little group, and `c_across()` lets `mean()` reach across a *selection* of columns (again using `starts_with()`) within that row.

Inside a single `mutate()` call, later expressions can refer to columns just created earlier in the *same* call – so we can reproduce the "replace the test mean with the final score if it's higher" rule from `compute.course.score()` using `if_else()` in place of the base-R `if` statement:

```
head(dplyr_results)
```

```
# A tibble: 6 x 6
  name  program    Quiz  Test Final course.score
  <chr> <fct>     <dbl> <dbl> <dbl>        <dbl>
1 S001  Program A  73.4    81    81         78.7
2 S002  Program B  73.4    70    70         71.0
3 S003  Program B  72.4    76    76         74.9
4 S004  Program A  72.4    70    70         70.7
5 S005  Program B  72      83    83         79.7
6 S006  Program A  73.4    74    74         73.8
```

First six of the 100 rows; the HTML version of this book shows the full table.

`rowwise()` groups by row, so we `ungroup()` again immediately afterward – otherwise every later verb would keep operating one row at a time. As a sanity check, this dplyr pipeline should produce *exactly* the same numbers as `results`, our earlier `apply()`-based table, just as the `for`-loop version did:

```
all.equal(as.data.frame(dplyr_results), results, check.attributes = FALSE)
```

```
[1] TRUE
```

2.7.3 Summarizing by group with `group_by()` and `summarize()`

This is where dplyr's advantage over base R really shows. Producing a by-program summary table with `tapply()` or nested `sapply()` calls in base R takes some fiddling; with dplyr it's a direct continuation of the same pipeline. `group_by()` splits the data by `program`, and every subsequent `summarize()` argument is computed separately within each group:

```
dplyr_results |>
    group_by(program) |>
    summarize(
        n                = n(),
        mean_quiz        = mean(Quiz, na.rm = TRUE),
        mean_test        = mean(Test, na.rm = TRUE),
        mean_final       = mean(Final, na.rm = TRUE),
        mean_course_score = mean(course.score, na.rm = TRUE),
        .groups = "drop"
    )
```

```
# A tibble: 2 x 6
  program       n mean_quiz mean_test mean_final mean_course_score
  <fct>     <int>     <dbl>     <dbl>      <dbl>             <dbl>
1 Program A    50      76.3      80.2       79.4              78.7
2 Program B    50      74.1      74.1       72.9              73.8
```

Program A's mean course score comes out noticeably higher – exactly the intercept difference (`b0.test`, `b0.final`) we built into the simulation, and the same pattern we'll see again in the by-program boxplot below.

2.7.4 Filtering and sorting with `filter()` and `arrange()`

`filter()` keeps rows matching a logical condition (replacing the `which()`-based indexing we used for `Failing_Students`), and `arrange()` sorts rows by one or more columns – `desc()` reverses the order. Chaining them together gives the failing students, worst first, and separately the top five performers in the class:

```
# Failing students, worst first
dplyr_results |>
    filter(course.score < 60) |>
    arrange(course.score) |>
    select(name, program, course.score)
```

```
# A tibble: 1 x 3
  name  program   course.score
  <chr> <fct>            <dbl>
1 S094  Program B         56.0
```

```
# Top 5 students in the class
dplyr_results |>
    arrange(desc(course.score)) |>
    select(name, program, course.score) |>
    slice_head(n = 5)
```

```
# A tibble: 5 x 3
  name  program   course.score
  <chr> <fct>            <dbl>
1 S079  Program A         93.8
2 S016  Program A         92.2
3 S010  Program A         91.2
4 S011  Program A         89.8
5 S061  Program B         89.8
```

We'll keep using `class_report$ProcessedTable` – our original `apply()`-based results – for the reports and plots below, but from here on in the course, dplyr's pipe-and-verb style will be our default tool for wrangling real datasets.

2.8 Presenting the Results

2.8.1 Presenting the Processed Results with `gt`

Finally, **gt** turns the raw data frame of results into a polished, presentation-ready report. We can extract the `ProcessedTable` directly from our `class_report` list and enhance it.

Here we use `tab_spanner()` to group the category scores and the final score under distinct visual headers, and `fmt_number()` to ensure all grades display exactly two decimal places. The column headers below spell out the underlying formulas: the quiz average $\sum_i^5 q_i/5$, the test score T, the final exam F_1, and the overall course score F.

```r
library(gt)

# Extract the full processed table from our list output
# (We use head() to display just the first 10 rows in the report)
full_results <- head(class_report$ProcessedTable, 10)

gt(full_results) |>
    tab_header(
        title = "Class Performance Report",
        subtitle = "Individual category means and final calculated course scores"
    ) |>
    # Student Information
    tab_spanner(
        label = "Student Information",
        columns = c(name, program)
    )|>
    # Group the individual graded categories
    tab_spanner(
        label = "Category Breakdown",
        columns = c(Quiz, Test, Final)
    ) |>
    # Group the final processed output
    tab_spanner(
        label = "Submitted Score",
        columns = c(course.score)
    ) |>
    # Enforce two decimal places for all numeric score columns
    fmt_number(
        columns = c(Quiz, Test, Final),
        decimals = 2
    ) |>
    fmt_number(
        columns = c(course.score),
        decimals = 0
    ) |>
    cols_label(
        name = "Name",
        program = "Program",
        Quiz = md("Quizzes ($\\sum_{i}^5{q_i}/5$)"),
        Test = md("Tests ($T$)"),
        Final = md("Final Exam ($F_1$)"),
        course.score = md("Course Score ($F$)")
    ) |>
    # Shrink the name/program columns so the score columns have more room
    cols_width(
        name ~ px(100),
        program ~ px(90)
    ) |>
    tab_stubhead(label = "Student ID")
```

Table 2.1: Shows the kind of tidy data frame used to practise data handling in R, with one row per student and one column per assessment category. Class performance report: first 10 students, with category scores and the final course score.

Class Performance Report
Individual category means and final calculated course scores

Student Information		Category Breakdown			Submitted Score
Name	Program	Quizzes ($\sum_i^5 q_i/5$)	Tests (T)	Final Exam (F_1)	Course Score (F)
S001	Program A	73.40	81.00	81.00	79
S002	Program B	73.40	70.00	70.00	71
S003	Program B	72.40	76.00	76.00	75
S004	Program A	72.40	70.00	70.00	71
S005	Program B	72.00	83.00	83.00	80
S006	Program A	73.40	74.00	74.00	74
S007	Program A	61.80	70.00	59.00	63
S008	Program B	67.00	62.00	62.00	64
S009	Program B	73.50	76.00	76.00	75
S010	Program A	82.40	95.00	95.00	91

2.8.2 Base R graphics

A quick histogram shows the distribution of course scores, with a vertical line marking the class average:

```
hist(class_report$ProcessedTable$course.score, breaks = 15, col = "lightblue",
     main = "Distribution of Course Scores", xlab = "Course Score")
abline(v = mean(class_report$ProcessedTable$course.score, na.rm = TRUE), col = "red", lwd = 2)
```

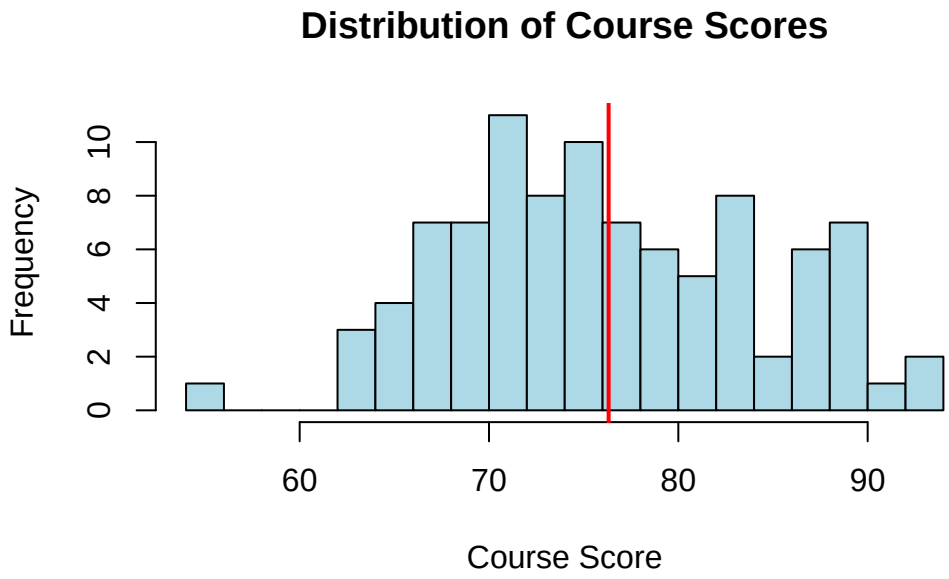


Figure 2.1: Demonstrates how to summarize a single numeric variable in R with a histogram, which shows the shape, centre, and spread of the class scores. Distribution of course scores across the class, with the class average marked in red.

2.8.3 Plotting with ggplot2

ggplot2 builds a comparison across programs declaratively, by mapping columns of `class_report$ProcessedTable` to visual properties and layering geometries with `+`. Here the boxplot is enhanced with the individual course scores jittered on top:

```
library(ggplot2)

ggplot(class_report$ProcessedTable, aes(x = program, y = course.score, fill = program)) +
    geom_boxplot(show.legend = FALSE, alpha = 0.6) +
    geom_jitter(width = 0.05, size = 1.5, alpha = 0.6) +
    labs(
        title = "Course Score by Academic Program",
        x = "Program", y = "Course Score"
    ) +
    theme_minimal()
```

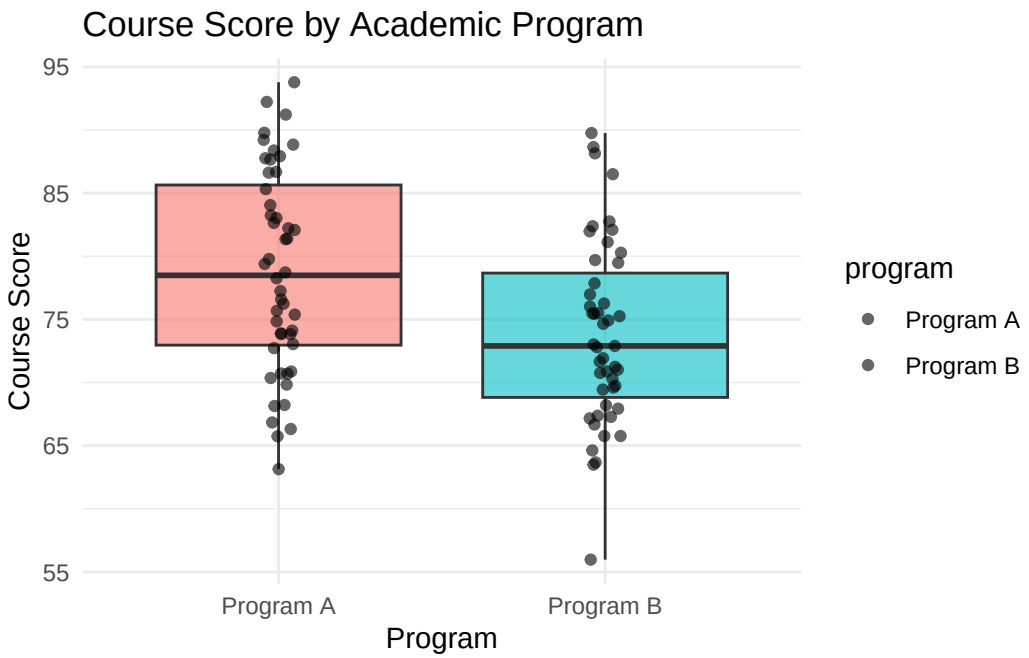


Figure 2.2: Demonstrates how to compare a numeric variable across groups in R, using boxplots with jittered points so that both the group summaries and the individual scores are visible. Course score by academic program, with individual scores jittered over the boxplots.

We built a program-specific intercept into the simulation when generating the final exam scores; a scatter plot of the final versus each student's mean quiz score, with a fitted line per program, should reveal it:

```
ggplot(class_report$ProcessedTable, aes(x = Quiz, y = Final, color = program)) +
    geom_point(size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    labs(
        title = "Final Exam vs. Mean Quiz Score, by Program",
        x = "Mean Quiz Score", y = "Final Exam Score", color = "Program"
    ) +
    theme_minimal()
```

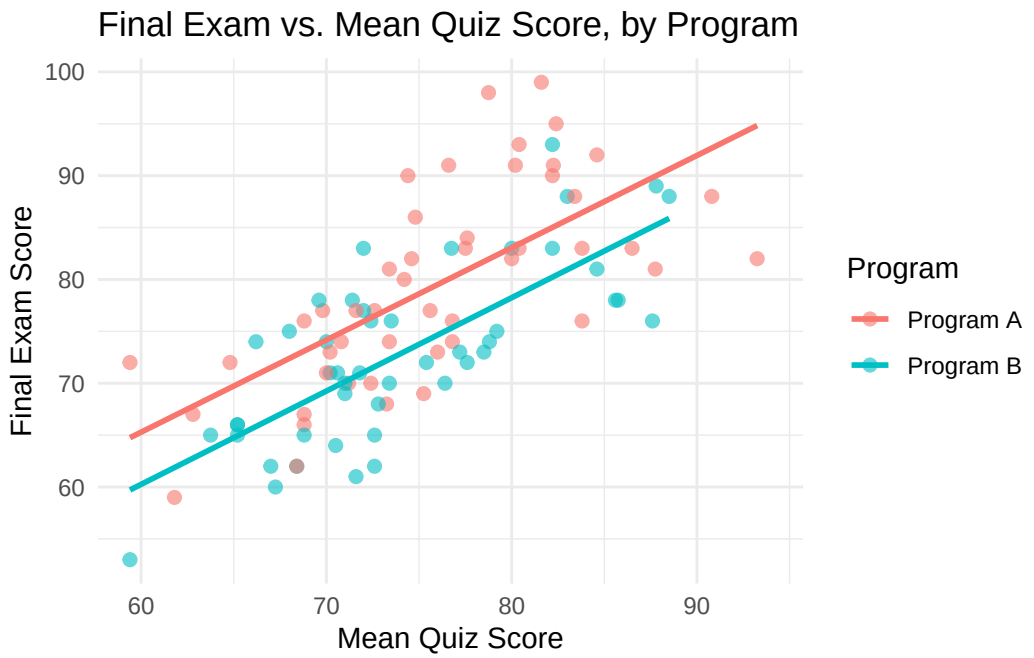


Figure 2.3: Demonstrates how to visualize a relationship separately within groups in R, by fitting one regression line per program to a scatterplot. Final exam score versus mean quiz score, by program, with a fitted regression line per program revealing the built-in intercept offset.

The two fitted lines are roughly parallel but vertically offset – exactly the intercept difference we built into `b0.final` when simulating the data.

3 Computer Arithmetic

Statistical computation is arithmetic on a machine that stores every number in a fixed number of binary digits. This chapter describes how integers and real numbers are represented, what goes wrong when a computation exceeds the capacity of that representation (overflow, underflow, and roundoff error), and the standard techniques for writing statistical code that avoids these failures: reordering sums, rewriting formulas, and working on the logarithmic scale.

3.1 Representations of Numbers in Computer

3.1.1 Integers

Every number is a string of digits in boxes, and each box has a **place value**. In base 10,

$$\begin{array}{cccc} 10^3 & 10^2 & 10^1 & 10^0 \\ \boxed{1} & \boxed{3} & \boxed{4} & \boxed{8} \end{array}$$
$$1348 = 1 \times 10^3 + 3 \times 10^2 + 4 \times 10^1 + 8 \times 10^0$$

A computer uses base 2, so each box, called a **bit**, holds 0 or 1:

$$\begin{array}{cccc} 2^3 & 2^2 & 2^1 & 2^0 \\ \boxed{1} & \boxed{1} & \boxed{0} & \boxed{1} \end{array}$$
$$1101_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13$$

Everything that follows is a consequence of one fact: the number of boxes is fixed in advance.

An integer is stored in a fixed number of bits, typically 32 or 64. With p bits the largest unsigned value is obtained by filling every box with 1,

$$\sum_{i=0}^{p-1} 2^i = 2^p - 1. \tag{3.1}$$

A 4-bit example: the largest storable integer is $1111_2 = 15$, and $15 + 1 = 16 = 10000_2$ needs a fifth box that does not exist.

$$\begin{array}{ccccc} 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ \boxed{1} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} \end{array}$$

The leading 1 has nowhere to go: **integer overflow**

To represent negative values, one of the boxes is used as a **sign bit**, leaving $p - 1$ boxes for the magnitude. With 32 bits, a signed integer u is

$$u = (-1)^s \sum_{i=0}^{30} d_i \, 2^i, \qquad s \in \{0, 1\}. \tag{3.2}$$

The largest integer has the sign bit 0 and the other 31 boxes all equal to 1:

$$\begin{array}{cccccccc} \text{sign} & 2^{30} & 2^{29} & \cdots & 2^1 & 2^0 \\ \boxed{+} & \boxed{1} & \boxed{1} & \boxed{\cdots} & \boxed{1} & \boxed{1} \end{array}$$

$$u^{\max} = \sum_{i=0}^{30} 2^i = 2^{31} - 1 = 2147483647$$

and the smallest is its negative, $u^{\min} = -(2^{31} - 1)$. When an integer computation leaves this range, R does not wrap around silently, as C does; it returns `NA` with a warning.

```
.Machine$integer.max
```

```
[1] 2147483647
```

```
## Check upper bounds
as.integer(2^31 - 1)                    # maximum valid integer
```

```
[1] 2147483647
```

```
as.integer(2^31)                        # exceeds limit, returns NA
```

```
[1] NA
```

```
as.integer(2^31 - 1) + as.integer(1)    # overflows to NA
```

```
[1] NA
```

```
## Check lower bounds
as.integer(-(2^31 - 1))                 # minimum valid integer
```

```
[1] -2147483647
```

```
as.integer(-2^31)                       # exceeds lower limit, returns NA
```

```
[1] NA
```

A practical consequence: products of moderately large integers overflow easily. `50000L * 50000L` is `NA` in R, while `50000 * 50000` (double precision) is fine. R converts to double automatically for most arithmetic, so integer overflow is rarely a problem in R code; it is a common one in C, Java, and database systems.

```
50000L * 50000L
```

```
[1] NA
```

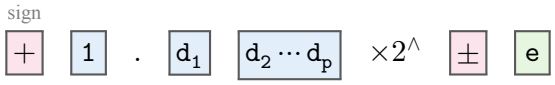
```
50000 * 50000
```

```
[1] 2.5e+09
```

3.1.2 Floating point numbers

A real number is stored in the form of scientific notation: a **sign**, a **mantissa** holding the significant digits, and an **exponent**:

$$x = (-1)^s \times 1.d_1 d_2 \cdots d_p \times 2^e. \tag{3.3}$$



More precisely, a floating point number x is represented by a vector of bits $(s, d_0, \dots, d_{t-1}, e_1, \dots, e_{k-1})$ as,

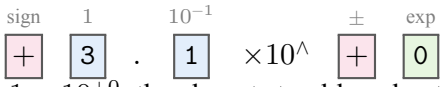
$$x = (-1)^s \sum_{i=0}^{t-1} d_i 2^{-i} \times 2^{\sum_{i=0}^{k-1} e_i 2^i - 2^{k-1}}. \tag{3.4}$$

3.1.3 Zoom Models for Representable Floating Numbers

A floating point number is a **ruler** and a **zoom setting**. The mantissa is a ruler with a fixed number of equally spaced marks; the exponent stretches it by 10^e over $[10^e, 10^{e+1})$. Stretching adds no marks, so the marks spread apart as the zoom grows.

3.1.3.1 A toy decimal computer

Each number is stored in six boxes: a sign, a two-digit mantissa $d_1.d_2$ ($d_1 \in \{1, \dots, 9\}$, $d_2 \in \{0, \dots, 9\}$), and an exponent made of a sign and one decimal digit, so $e \in \{-9, \dots, 0, \dots, +9\}$.



$+3.1 \times 10^{+0}$, the closest storable value to π

Each of the 19 zoom levels is a ruler of 90 marks. At the lowest zoom $e = -9$ the leading digit may also be 0 (**subnormal** numbers), which fills the gap between 0 and 1.0×10^{-9}.

zoom e	interval	storable numbers	gap
-9 (subnormal)	$(0, 10^{-9})$	$0.1 \times 10^{-9}, \dots, 0.9 \times 10^{-9}$	10^{-10}
-9	$[10^{-9}, 10^{-8})$	$1.0 \times 10^{-9}, 1.1 \times 10^{-9}, \dots, 9.9 \times 10^{-9}$	10^{-10}
$\vdots$	$\vdots$	$\vdots$	$\vdots$

zoom e	interval	storable numbers	gap
-1	$[0.1, 1)$	$0.10, 0.11, \ldots, 0.99$	0.01
0	$[1, 10)$	$1.0, 1.1, \ldots, 9.9$	0.1
$+1$	$[10, 100)$	$10, 11, \ldots, 99$	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$+9$	$[10^9, 10^{10})$	$1.0 \times 10^9, 1.1 \times 10^9, \ldots, 9.9 \times 10^9$	10^8

With both signs, the 9 subnormals per sign, and a reserved code for 0, the machine stores exactly $2 \times (19 \times 90 + 9) + 1 = 3439$ numbers. The 90 normal mantissas are listed below; multiplying them by 10^e, $e = -9, \ldots, 9$, gives every positive normal number.

```r
m <- outer(1:9, 0:9, function(a, b) sprintf("%.1f",a + b / 10))
dimnames(m) <- list(d1 = 1:9, d2 = 0:9)
m
```

```
   d2
d1  0     1     2     3     4     5     6     7     8     9
  1 "1.0" "1.1" "1.2" "1.3" "1.4" "1.5" "1.6" "1.7" "1.8" "1.9"
  2 "2.0" "2.1" "2.2" "2.3" "2.4" "2.5" "2.6" "2.7" "2.8" "2.9"
  3 "3.0" "3.1" "3.2" "3.3" "3.4" "3.5" "3.6" "3.7" "3.8" "3.9"
  4 "4.0" "4.1" "4.2" "4.3" "4.4" "4.5" "4.6" "4.7" "4.8" "4.9"
  5 "5.0" "5.1" "5.2" "5.3" "5.4" "5.5" "5.6" "5.7" "5.8" "5.9"
  6 "6.0" "6.1" "6.2" "6.3" "6.4" "6.5" "6.6" "6.7" "6.8" "6.9"
  7 "7.0" "7.1" "7.2" "7.3" "7.4" "7.5" "7.6" "7.7" "7.8" "7.9"
  8 "8.0" "8.1" "8.2" "8.3" "8.4" "8.5" "8.6" "8.7" "8.8" "8.9"
  9 "9.0" "9.1" "9.2" "9.3" "9.4" "9.5" "9.6" "9.7" "9.8" "9.9"
```

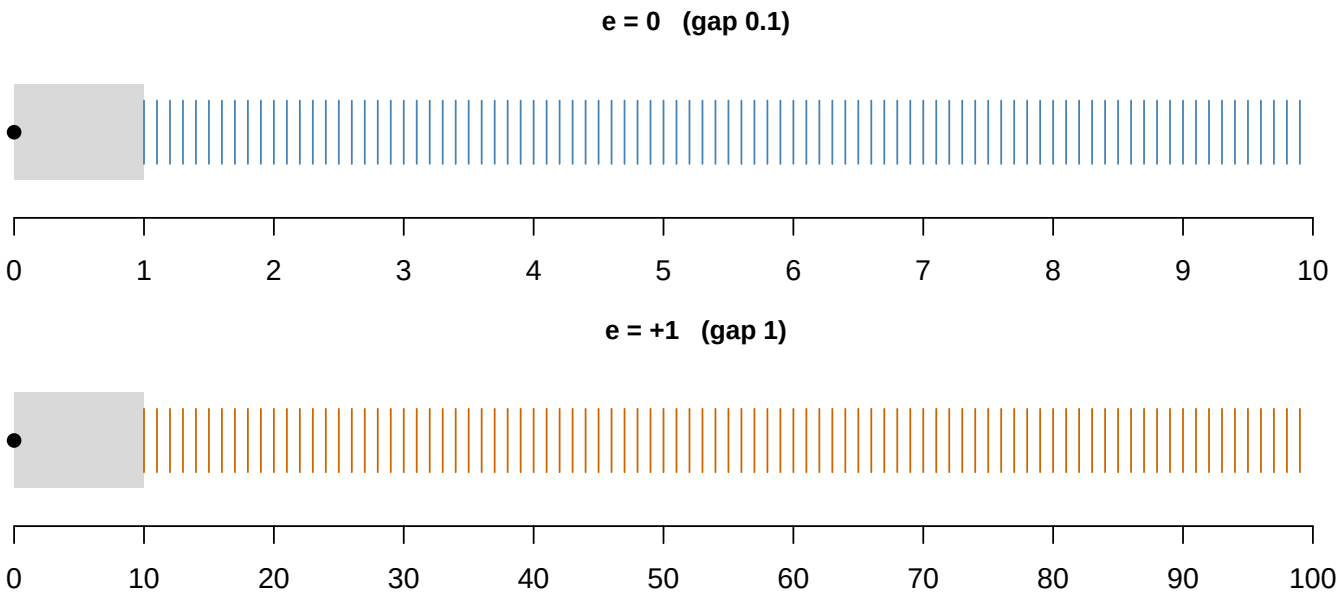


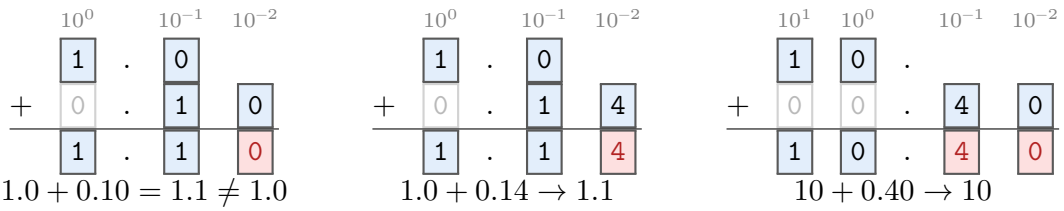
Figure 3.1: Illustrates how floating-point numbers are spaced on the real line in a toy number system: the gaps between representable numbers grow with magnitude, so precision is relative rather than absolute. Positive numbers at zoom $e = 0$ (top, gap 0.1) and $e = +1$ (bottom, gap 1). Grey bands hold the crowded marks of all lower zooms ($e < 0$, and $e \leq 0$ in the bottom panel); overflow occurs only far to the right, beyond 9.9×10^9.

3.1.3.2 Rounding error and machine epsilon

A real number between two marks is stored as the nearer mark. The three numbers below share the same mantissa boxes and differ only in the exponent, so the absolute error scales with the zoom while the relative error does not.

x	sign	1	.	10^{-1}		$\pm$	exp	stored	error
0.234	+	2	.	3	$\times 10^{\wedge}$	−	1	0.23	0.004 (1.7%)
2.34	+	2	.	3	$\times 10^{\wedge}$	+	0	2.3	0.04 (1.7%)
23.4	+	2	.	3	$\times 10^{\wedge}$	+	1	23	0.4 (1.7%)

The same rounding occurs in arithmetic. To add, the two numbers are aligned by place value; solid boxes are stored digits, dashed boxes are alignment zeros, and the result keeps only the two boxes starting at its leading digit, so digits to the right (red) are rounded off.



The **machine epsilon** ϵ is the gap next to 1, the smallest increment that the second mantissa box can register when added to 1; here $\epsilon = 0.1$ (left). Smaller increments fall into the red box: the digit 4 is lost (middle), and at zoom $e = +1$ the whole addend 0.40 is lost (right). Relative to the magnitude, the resolution is about ϵ at every zoom.

3.1.3.3 Largest and smallest numbers

	sign	1		10^{-1}		$\pm$	exp	value
largest	+	9	.	9	$\times 10^{\wedge}$	+	9	9.9×10^{9}
smallest positive	+	0	.	1	$\times 10^{\wedge}$	−	9	0.1×10^{-9}
largest negative	−	0	.	1	$\times 10^{\wedge}$	−	9	-0.1×10^{-9}
smallest	−	9	.	9	$\times 10^{\wedge}$	+	9	-9.9×10^{9}

Largest mantissa with the largest exponent; smallest subnormal mantissa with the smallest exponent

Overflow. A result with magnitude at least 9.95×10^{9} needs the exponent $+10$, which the single exponent digit cannot hold, so it is returned as $\pm\infty$:

	10^{10}	10^{9}	10^{8}
		6	0
+		5	0
	1	1	0

$6.0 \times 10^{9} + 5.0 \times 10^{9} = 1.1 \times 10^{+10} \to \infty$

sign	1		10^{-1}		$\pm$	exp
+	1	.	1	$\times 10^{\wedge}$	+	10

The exponent 10 needs two digits but has one box: overflow

Underflow. A result with magnitude below 0.05×10^{-9} has no box pattern and is returned as 0:

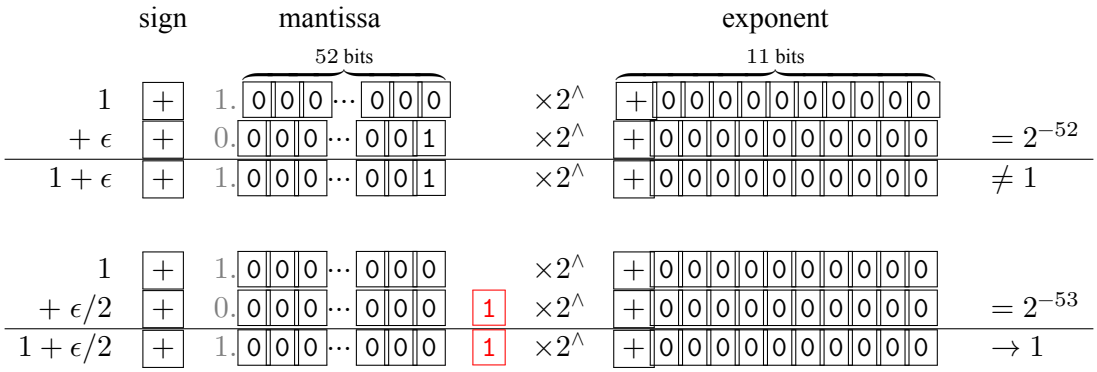
sign	1		10^{-1}	10^{-2}		$\pm$	exp
+	0	.	0	1	$\times 10^{\wedge}$	−	9

$(0.1 \times 10^{-9}) \times 0.1 = 0.01 \times 10^{-9} \to 0$: the digit 1 falls outside the mantissa

Every later operation treats the underflowed result as exactly zero.

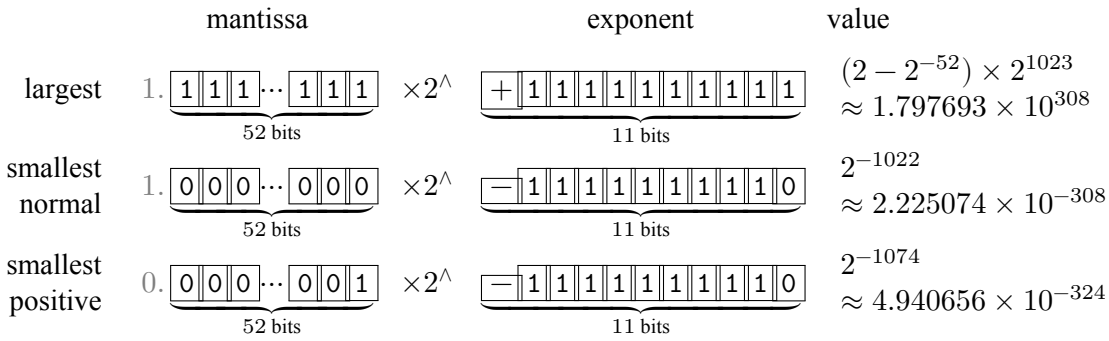
3.1.3.4 Doubles

A double uses the same boxes in base 2: 1 sign bit, 11 exponent bits, and 52 mantissa bits after an implicit leading 1. The exponent is stored with a bias of 1023, so the codes $1, \ldots, 2046$ give $e \in [-1022, 1023]$; the code 0 is reserved for 0 and the subnormals, and the code 2047 for `Inf` and `NaN`.



(3.5)

The **machine epsilon** is the smallest number that changes 1 when added to it: the place value of the last mantissa box at $e = 0$, $\epsilon = 0.0\cdots01 \times 2^0 = 2^{-52} \approx 2.220446 \times 10^{-16}$. Adding ϵ sets that box (top). Adding $\epsilon/2$ needs a 53rd box (red), which does not exist, so the sum rounds back to 1 (bottom; the tie is rounded to the even mantissa).



(3.6)

For readability, the picture shows the exponent as a sign box and 10 magnitude bits; the hardware stores the same 11 bits as the unsigned code $e + 1023$. The grey leading digit is not stored: it is 1 for normal doubles and 0 for subnormals, which occur only at $e = -1022$.

- The largest double has a full mantissa and the largest exponent, $+1023$.
- The smallest normal double has mantissa 1.0 and the smallest exponent, -1022.
- The smallest positive double is the subnormal with only the last mantissa box set at $e = -1022$, i.e. $2^{-52} \times 2^{-1022} = 2^{-1074}$.
- The **machine epsilon** $\epsilon = 2^{-52}$ is the gap next to 1, set by the last mantissa bit. Since the gap in $[2^e, 2^{e+1})$ is 2^{e-52}, ϵ is also the relative resolution at every magnitude: a double carries about 15–16 significant decimal digits.
- The **largest double** has all mantissa bits and all usable exponent bits set to 1. Results beyond it overflow to `Inf`.
- The **smallest normal double** has the smallest usable exponent and mantissa 1.0. Below it, subnormals fill the gap down to 2^{-1074} with evenly spaced but less precise values; results below half of 2^{-1074} underflow to 0.

quantity	value	R	relevance in computing
machine epsilon	$2^{-52} \approx 2.2 \times 10^{-16}$	`.Machine$double.eps`	Doubles cannot be compared exactly (`0.1 + 0.2 != 0.3`); convergence and equality tests use tolerances such as $\sqrt{\epsilon} \approx 1.5 \times 10^{-8}$ (`all.equal`), which is also the usual step size for numerical derivatives
largest double	$\approx 1.80 \times 10^{308}$	`.Machine$double.xmax`	`exp(x)` overflows for $x > 709.8$; products of large terms such as likelihood ratios or $n!$ must be computed on the log scale
smallest normal double	$2^{-1022} \approx 2.2 \times 10^{-308}$	`.Machine$double.xmin`	`exp(x)` leaves the normal range for $x < -708.4$; a product of many densities falls here, which is why likelihoods are computed as log-likelihoods and sums of exponentials via log-sum-exp
smallest subnormal double	$2^{-1074} \approx 4.9 \times 10^{-324}$	`2^-1074`	Last value before underflow to 0; subnormals keep few significant bits, so ratios of such values are unreliable
largest consecutive integer	$2^{53} \approx 9.0 \times 10^{15}$	`2^53`	Beyond it the gap exceeds 1 and integers are skipped; R's `integer` type stops even earlier at $2^{31} - 1$, so larger identifiers must be stored as character strings or `bit64::integer64`

3.1.3.5 Verify in R

```
c(eps = .Machine$double.eps, xmax = .Machine$double.xmax,
  xmin = .Machine$double.xmin, subnormal = 2^-1074, maxint = 2^53)
```

```
                        eps                        xmax
 2.220446049250313080847e-16 1.797693134862315708145e+308
```

```
                  xmin                   subnormal
2.2250738585072013830e-308 4.9406564584124654417e-324
                  maxint
 9.0071992547409920000e+15
```

These are the decimal values in the boxed displays above.

- `eps` $= 2^{-52}$ is the place value of the last mantissa box at $e = 0$, the ϵ added to 1 in the vertical addition.
- `xmax` is the "largest" row: a full mantissa with exponent $+1023$.
- `xmin` is the "smallest normal" row: mantissa 1.0 with exponent -1022.
- `subnormal` is the "smallest positive" row: only the last mantissa box set, with leading digit 0 and exponent -1022.
- `maxint` $= 2^{53}$ is where the last mantissa box becomes worth 2 (see `spacing-demo` below).

```
1 + 2^-52 == 1
```

```
[1] FALSE
```

```
1 + 2^-53 == 1                          # half an epsilon is lost
```

```
[1] TRUE
```

```
0.1 + 0.2 == 0.3                        # rounding error
```

```
[1] FALSE
```

```
isTRUE(all.equal(0.1 + 0.2, 0.3))     # test with tolerance sqrt(eps)
```

```
[1] TRUE
```

- `1 + 2^-52 == 1` is `FALSE`: the top vertical addition, where ϵ sets the last mantissa box and $1 + \epsilon$ is a different double.
- `1 + 2^-53 == 1` is `TRUE`: the bottom vertical addition, where $\epsilon/2$ needs the nonexistent 53rd box (red) and the sum rounds back to 1.
- `0.1 + 0.2 == 0.3` is `FALSE`: 0.1, 0.2 and 0.3 have infinitely repeating binary mantissas, so each is cut at the 52nd box, and the rounded sum lands on a different mark from the rounded 0.3.
- `all.equal(...)` is `TRUE`: it tests equality up to a relative tolerance $\sqrt{\epsilon} \approx 1.5 \times 10^{-8}$, the safe way to compare doubles.

```
2 * .Machine$double.xmax              # overflow: Inf
```

```
[1] Inf
```

```
exp(710)                              # overflow: Inf
```

```
[1] Inf
```

```
2^-1075                                 # underflow: 0
```

```
[1] 0
```

```
prod(rep(0.01, 200))                    # 1e-400 underflows to 0
```

```
[1] 0
```

```
sum(log(rep(0.01, 200)))                # the log scale is safe
```

```
[1] -921.0340371976157030076
```

- `2 * .Machine$double.xmax` is `Inf`: doubling the "largest" row needs exponent $+1024$, whose code is reserved for `Inf`, as in the toy $6.0 \times 10^9 + 5.0 \times 10^9$.
- `exp(710)` is `Inf`: e^{710} exceeds `xmax` $\approx e^{709.78}$.
- `2^-1075` is `0`: it is half the "smallest positive" row and would need a box beyond the last one; the tie rounds to the even pattern, which is 0.
- `prod(rep(0.01, 200))` is `0`: the true value 10^{-400} lies below $2^{-1074} \approx 4.9 \times 10^{-324}$. The same happens to a likelihood built as a product of many densities.
- `sum(log(rep(0.01, 200)))` is -921.03: on the log scale the product becomes a sum of moderate numbers, far from both limits.

```
2^52 + 1 == 2^52                        # gap 1: still distinguishable
```

```
[1] FALSE
```

```
2^53 + 1 == 2^53                        # gap 2: the +1 is lost
```

```
[1] TRUE
```

```
format(c(2^53, 2^53 + 1, 2^53 + 2), digits = 17)
```

```
[1] "9007199254740992" "9007199254740992" "9007199254740994"
```

```
.Machine$integer.max                    # 2^31 - 1
```

```
[1] 2147483647
```

- `2^52 + 1 == 2^52` is `FALSE`: at $e = 52$ the last mantissa box is worth $2^{52-52} = 1$, so adding 1 sets it, exactly as ϵ does at $e = 0$.
- `2^53 + 1 == 2^53` is `TRUE`: at $e = 53$ the last box is worth 2, so adding 1 needs a 53rd box. This is the $1 + \epsilon/2$ addition scaled by 2^{53}.
- `format(...)` prints `9007199254740992` twice and then `9007199254740994`: above 2^{53} only even integers are marks on the ruler.
- `.Machine$integer.max` is $2^{31} - 1 = 2147483647$: R's `integer` type is 32 bits, so integers beyond it, and beyond 2^{53} for doubles, must be stored as character strings or `bit64::integer64`.

3.2 Methods for Mitigating Rounding Errors

3.2.1 Roundoff errors

The stored result of $a + b$ (or $a \times b$, a/b, …) is the **nearest representable number** to the exact result; digits beyond the mantissa are rounded away, as in the vertical additions above. Each single operation therefore has relative error at most about ϵ, which is harmless on its own. The concern is when these errors **accumulate** over many operations or are **amplified** by the structure of a formula. Two patterns do the amplifying:

- adding a **large + small** number, where the small number is partly or wholly rounded away;
- subtracting **large − large** nearly equal numbers, where the leading digits cancel and only the roundoff errors survive (**catastrophic cancellation**).

3.2.2 Computing sums: avoiding large + small

Because the mantissa has finite capacity, a term smaller than half the last box of the large number is lost entirely, as in the addition $1 + \epsilon/2 \to 1$ shown earlier.

Consider $1.0 + 0.04 + 0.04$ in the toy system, whose exact value is 1.08. Evaluated left to right, $(1.0 + 0.04) + 0.04$, each addition rounds back to 1.0, so both small terms are lost. Evaluated as $1.0 + (0.04 + 0.04) = 1.0 + 0.08$, the intermediate sum is exact, and the final addition rounds once, to 1.1. Adding the small terms among themselves first commits only **one** roundoff error, whereas adding them one at a time to the large number commits one per term, and these accumulate.

The following R demonstration makes the same point at machine precision. `f2` attempts to add 2^{-53} (half an epsilon) 2^{13} times, but because $1 + 2^{-53} = 1$ at every single step, the final sum is simply 1 instead of $1 + 2^{-40}$. Grouping the small numbers and adding them together before adding them to the large number (`f6s`) recovers the correct answer.

The following R demonstration makes the same point at machine precision. `f2` attempts to add 2^{-53} (half an epsilon) 2^{13} times, but because $1 + 2^{-53} = 1$ at every single step, the final sum is simply 1 instead of $1 + 2^{-40}$. Grouping the small numbers and adding them together before adding them to the large number (`f6s`) recovers the correct answer.

```
## the target value
f0 <- 1 + 2^(-40)
f0
```

```
[1] 1.0000000000000909494702
```

```
## adding 2^-52 (one epsilon) 2^12 times works: each addition changes the sum
f1 <- 1; i <- 1
while (i <= 2^12) { f1 <- f1 + 2^(-52); i <- i + 1 }
f1
```

```
[1] 1.0000000000000909494702
```

```r
## adding 2^-53 (half an epsilon) 2^13 times fails: 1 + 2^-53 rounds to 1 every time
f2 <- 1; i <- 1
while (i <= 2^13) { f2 <- f2 + 2^(-53); i <- i + 1 }
f2
```

```
[1] 1
```

```r
## this difference is not due to number display in R
f0 == f1
```

```
[1] TRUE
```

```r
f0 == f2
```

```
[1] FALSE
```

```r
## R's built-in sum() uses pairwise summation and extended precision
f3 <- sum(c(1, rep(2^(-53), 2^13))); f3
```

```
[1] 1
```

```r
f1 == f3
```

```
[1] FALSE
```

```r
f4 <- sum(c(1, rep(2^(-54), 2^14))); f4
```

```
[1] 1
```

```r
f0 == f4
```

```
[1] FALSE
```

```r
## the fix: accumulate the small numbers first, then add to 1
f6s <- 0; i <- 1
while (i <= 2^14) { f6s <- f6s + 2^(-54); i <- i + 1 }
f6s
```

```
[1] 9.094947017729282379e-13
```

```r
f6 <- f6s + 1; f6
```

```
[1] 1.000000000000909494702
```

```r
f0 == f6
```

```
[1] TRUE
```

Rules for accurate summation

- Add numbers of similar magnitude together; when summing terms of very different sizes, add the smallest first (sort ascending).
- **Pairwise summation**: sum the two halves of the vector recursively, so that partial sums stay of similar size. R's `sum()` uses this together with extended-precision accumulation, which is why `sum()` beat the naive loop above.
- **Kahan compensated summation**: keep track of the part rounded away at each step and add it back at the next step.

```
x <- c(1, rep(1e-16, 1e4))                 # exact sum is 1 + 1e-12

naive_sum <- function(x) { s <- 0; for (v in x) s <- s + v; s }

kahan_sum <- function(x) {
  s <- 0; c <- 0                            # c holds the running roundoff error
  for (v in x) { y <- v - c; t <- s + y; c <- (t - s) - y; s <- t }
  s }

c(naive   = naive_sum(x) - 1,
  sorted  = naive_sum(sort(x)) - 1,
  kahan   = kahan_sum(x) - 1,
  builtin = sum(x) - 1,
  exact   = 1e-12)
```

```
                    naive                    sorted
0.00000000000000000000e+00 1.00008890058234101161e-12
                    kahan                   builtin
1.00008890058234101161e-12 0.00000000000000000000e+00
                    exact
9.99999999999999798866e-13
```

The naive loop loses every one of the 10^4 small terms; sorting recovers most of the answer, and Kahan summation and R's `sum()` recover it to full precision.

We can use `log()` to translate the 52-bit limit into other bases, which tells us at roughly what decimal place precision is lost (about 15 to 16 significant decimal digits).

```
log(2^(-52))               # base e: about -36
```

```
[1] -36.04365338911715355152
```

```
1 + exp(-35)
```

```
[1] 1.00000000000000666134
```

```
1 + exp(-37)               # evaluates to exactly 1
```

```
[1] 1
```

```
log(2^(-52), base = 10)    # base 10: about -15.7
```

```
[1] -15.6535977452702163077
```

```
1 + 1e-15 == 1
```

```
[1] FALSE
```

```
1 + 1e-16 == 1
```

```
[1] TRUE
```

3.2.3 Computing e^x: avoiding large − large

The Taylor series,

$$e^x \approx \sum_{k=0}^{K} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots, \tag{3.7}$$

works well for $x > 0$ but fails for large negative x. For $x = -20$ the terms alternate in sign and are huge relative to the answer $e^{-20} \approx 2 \times 10^{-9}$:

k	0	1	2	…	20	…
$x^k/k!$	1	-20	200	…	4.3×10^7	…

Each term carries a roundoff error of relative size $\epsilon \approx 10^{-16}$, i.e. absolute size $\approx 4 \times 10^{-9}$ for the largest terms, which is **as large as the answer itself**. Subtracting nearly equal large numbers leaves only their errors: catastrophic cancellation.

The function `fexp` below implements the series with a stopping rule based on the relative size of the current term. With `debug = TRUE` it returns the internal steps rather than the final answer.

```
library(knitr)

fexp <- function(x, debug = FALSE) {
  i <- 0; expx <- 1; u <- 1
  if (debug) debug_list <- list()
  while (abs(u) > 1e-20 * abs(expx)) {
    i <- i + 1
    u <- u * x / i
    expx <- expx + u
    if (debug) debug_list[[i]] <- c(Step = i, Term_u = u, exp_x = expx)
  }
  if (debug) return(do.call(rbind, debug_list))
  expx
}
```

```
test_vals <- c(10, 20, 60, -1, -10, -20, -50)
res1 <- data.frame(x = test_vals, Built_in_Exp = exp(test_vals), Taylor_fexp =
 ↪  sapply(test_vals, fexp))
kable(res1, caption = "Compares a naive Taylor-series implementation of the exponential
 ↪  function with the built-in exp() to show that the series is accurate for positive arguments
 ↪  but fails for negative ones. Built-in exp() versus the naive Taylor series fexp()")
```

Table 3.1: Compares a naive Taylor-series implementation of the exponential function with the built-in exp() to show that the series is accurate for positive arguments but fails for negative ones. Built-in exp() versus the naive Taylor series fexp()

x	Built_in_Exp	Taylor_fexp
10	2.2026465794806717894e+04	2.2026465794806710619e+04
20	4.8516519540979027748e+08	4.8516519540979021787e+08
60	1.1420073898156842345e+26	1.1420073898156847499e+26
-1	3.6787944117144233402e-01	3.6787944117144244505e-01
-10	4.5399929762484854173e-05	4.5399929623031280972e-05
-20	2.0611536224386000561e-09	5.6218844721304002413e-09
-50	2.0000000000000000972e-22	1.1072933382891969814e+04

For positive x the series is accurate to all printed digits. For $x = -10$ the answer is already off in the fourth digit, for $x = -20$ it is wrong in the first digit, and `fexp(-50)` yields a completely wrong, negative number. Running it in debug mode shows the massive terms cancelling each other out.

```
debug_matrix <- fexp(-50, debug = TRUE)
debug_summary <- rbind(
  head(debug_matrix, 10),
  matrix(NA, nrow = 1, ncol = 3, dimnames = list("...", colnames(debug_matrix))),
  tail(debug_matrix, 10))
kable(debug_summary, caption = "Exposes the internals of the failing computation of exp(-50),
 ↪  showing that very large terms of alternating sign cancel and leave only roundoff error.
 ↪  Debug trace for fexp(-50) showing catastrophic cancellation")
```

Table 3.2: Exposes the internals of the failing computation of exp(-50), showing that very large terms of alternating sign cancel and leave only roundoff error. Debug trace for fexp(-50) showing catastrophic cancellation

Step	Term_u	exp_x
1	-5.0000000000000000000e+01	-4.9000000000000000000e+01
2	1.2500000000000000000e+03	1.2010000000000000000e+03
3	-2.0833333333333212067e+04	-1.9632333333333212067e+04
4	2.6041666666666569654e+05	2.4078433333333139308e+05
5	-2.6041666666666511446e+06	-2.3633823333333022892e+06
6	2.1701388888888806105e+07	1.9338006555555596948e+07
7	-1.5500992063492068787e+08	-1.3567191407936507463e+08

Table 3.2: Exposes the internals of the failing computation of exp(-50), showing that very large terms of alternating sign cancel and leave only roundoff error. Debug trace for fexp(-50) showing catastrophic cancellation

	Step	Term_u	exp_x
	8	9.6881200396825397014 62e+08	8.3314008988888883590 70e+08
	9	-5.3822889109347438812 26e+09	-4.5491488210458545684 81e+09
	10	2.6911444554673717498 78e+10	2.2362295733627861022 95e+10
…	NA	NA	NA
[158,]	158	1.4767995501000000682 76e-12	1.1072933382891969813 50e+04
[159,]	159	-4.6440237430000000047 2e-13	1.1072933382891969813 50e+04
[160,]	160	1.4512574200000000533 39e-13	1.1072933382891969813 50e+04
[161,]	161	-4.5070106199999997346 73e-14	1.1072933382891969813 50e+04
[162,]	162	1.3910526600000000662 31e-14	1.1072933382891969813 50e+04
[163,]	163	-4.2670326999999997481 26e-15	1.1072933382891969813 50e+04
[164,]	164	1.3009246000000000056 40e-15	1.1072933382891969813 50e+04
[165,]	165	-3.9421959999999999734 45e-16	1.1072933382891969813 50e+04
[166,]	166	1.1874080000000001146 43e-16	1.1072933382891969813 50e+04
[167,]	167	-3.5551099999999998747 25e-17	1.1072933382891969813 50e+04

The largest terms reach about 10^{20}; their roundoff errors are of order 10^{4}, and the true answer $e^{-50} \approx 2 \times 10^{-22}$ is invisible beneath them.

Fix by reformulating. For $x > 0$ all terms are positive, so no cancellation occurs. Therefore compute $e^{|x|}$ and invert the result if x was negative:

$$e^{-x} = \frac{1}{e^{x}}, \qquad x > 0. \tag{3.8}$$

```
fexp2 <- function(x) {
  xa <- abs(x); i <- 0; expx <- 1; u <- 1
  while (u > 1e-20 * expx) { i <- i + 1; u <- u * xa / i; expx <- expx + u }
  if (x >= 0) expx else 1 / expx
}

test_vals_neg <- c(-10, -20, -50)
res2 <- data.frame(x = test_vals_neg, Built_in_Exp = exp(test_vals_neg), Robust_fexp2 =
 ↪  sapply(test_vals_neg, fexp2))
kable(res2, caption = "Shows that reformulating the computation as 1/exp(|x|) removes the
 ↪  cancellation and restores accuracy for negative arguments. Robust fexp2() for negative
 ↪  arguments")
```

Table 3.3: Shows that reformulating the computation as 1/exp(|x|) removes the cancellation and restores accuracy for negative arguments. Robust fexp2() for negative arguments

x	Built_in_Exp	Robust_fexp2
-10	4.5399929762484854173 15e-05	4.5399929762484860949 41e-05
-20	2.0611536224386000561 53e-09	2.0611536224386000561 53e-09
-50	2.0000000000000000971 94e-22	2.0000000000000000971 94e-22

Table 3.3: Shows that reformulating the computation as 1/exp(|x|) removes the cancellation and restores accuracy for negative arguments. Robust fexp2() for negative arguments

x	Built_in_Exp	Robust_fexp2

The general principle: rearrange a formula so that quantities of similar magnitude and **the same sign** are combined, and avoid computing a small result as the difference of two large numbers.

A second example of the same principle. The function $f(x) = e^x - 1$ is needed in interest-rate and survival calculations for small x. Computing `exp(x) - 1` directly subtracts two numbers near 1, and for $x = 10^{-10}$ loses six of the sixteen digits; `expm1(x)` uses a series for small x and is exact to machine precision. The same holds for $\log(1 + x)$ and `log1p(x)`.

```
x <- 1e-10
c(naive = exp(x) - 1, expm1 = expm1(x), series = x + x^2 / 2)
```

```
                    naive                      expm1
1.000000082740370999090e-10 1.000000000050000002534e-10
                   series
1.000000000050000002534e-10
```

```
x <- 1e-20
c(naive = log(1 + x), log1p = log1p(x))
```

```
                    naive                     log1p
0.000000000000000000000e+00 9.999999999999999451533e-21
```

3.2.4 Computing variance: avoiding large − large

Two algebraically identical formulas for the sum of squared deviations:

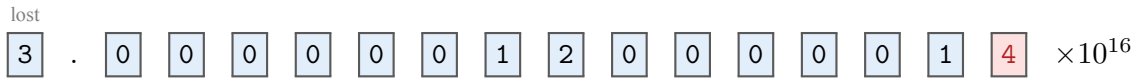
$$\underbrace{\sum_{i=1}^{n}(x_i - \overline{x})^2}_{\text{two-pass}} \;=\; \underbrace{\sum_{i=1}^{n} x_i^2 - \frac{\left(\sum_i x_i\right)^2}{n}}_{\text{one-pass}}. \tag{3.9}$$

The one-pass formula needs only $\sum x_i$ and $\sum x_i^2$, computed in a single sweep of the data, which made it attractive for hand calculators and for streaming data. But when the x_i are large and close together, $\sum x_i^2$ and $(\sum x_i)^2/n$ are both huge and nearly equal, and the subtraction is large − large.

Take $x = (1, 2, 3)$: the deviations are $(-1, 0, 1)$, so $\sum(x_i - \overline{x})^2 = 2$. Now shift by $M = 10^8$, giving $x = (M + 1, M + 2, M + 3)$. The deviations and the answer are unchanged, but the one-pass formula computes

$$\sum x_i^2 \approx 3 \times 10^{16}, \qquad \frac{(\sum x_i)^2}{n} \approx 3 \times 10^{16}, \qquad \text{difference} = 2. \tag{3.10}$$

With 16 significant digits, 3×10^{16} is stored with a gap of about 4 between neighbouring representable numbers, so the true answer 2 lives entirely in the lost digits.




```
## one-pass (computational) formula
var1 <- function(x) { n <- length(x); (sum(x^2) - sum(x)^2 / n) / (n - 1) }

## two-pass formula (centering on the mean)
var2 <- function(x) { n <- length(x); sum((x - mean(x))^2) / (n - 1) }

## small data works fine for both
x <- c(1, 2, 3)
c(var1(x), var2(x), var(x))
```

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

```
## a large shift exposes catastrophic cancellation in var1
x <- c(1, 2, 3) + 1e8
c(var1(x), var2(x), var(x))
```

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

```
x <- c(1, 2, 3) + 1e10
c(var1(x), var2(x), var(x))
```

```
[1] -32768      1      1
```

The failure sets in gradually. The error of the one-pass formula is of order $\epsilon\, M^2 n$, so it becomes visible once $M^2\epsilon$ is comparable to the true variance, i.e. around $M \approx 10^8$ for a variance of 1. The following scan shows the one-pass result drifting away from 1 and eventually becoming negative, which is impossible for a variance.

```
M <- 10^(0:10)
res <- t(sapply(M, function(m) { x <- c(1, 2, 3) + m; c(M = m, one_pass = var1(x), two_pass =
 ↪  var2(x)) }))
kable(res, caption = "Demonstrates catastrophic cancellation in the one-pass variance formula,
 ↪  which breaks down when the data have a large mean, while the two-pass formula stays
 ↪  accurate. One-pass versus two-pass variance as the data are shifted by M", digits = 6)
```

Table 3.4: Demonstrates catastrophic cancellation in the one-pass variance formula, which breaks down when the data have a large mean, while the two-pass formula stays accurate. One-pass versus two-pass variance as the data are shifted by M

M	one_pass	two_pass
1e+00	1	1
1e+01	1	1
1e+02	1	1
1e+03	1	1
1e+04	1	1

Table 3.4: Demonstrates catastrophic cancellation in the one-pass variance formula, which breaks down when the data have a large mean, while the two-pass formula stays accurate. One-pass versus two-pass variance as the data are shifted by M

M	one_pass	two_pass
1e+05	1	1
1e+06	1	1
1e+07	1	1
1e+08	0	1
1e+09	0	1
1e+10	-32768	1

Fixes

- **Two-pass**: compute $\overline{x}$ first, then $\sum(x_i - \overline{x})^2$. The deviations are small and of similar size, so no cancellation occurs. This is what `var()` does.
- **Shifted one-pass**: subtract any convenient constant $c \approx \overline{x}$, for instance $c = x_1$, before applying the one-pass formula. The identity

$$\sum(x_i - \overline{x})^2 = \sum(x_i - c)^2 - \frac{\left(\sum(x_i - c)\right)^2}{n}, \tag{3.11}$$

  holds for every c, and with c near the mean the two terms on the right are no longer huge.
- **Welford's online algorithm** updates the running mean $\overline{x}_k$ and running sum of squares $S_k = \sum_{i \le k}(x_i - \overline{x}_k)^2$ one observation at a time,

$$\overline{x}_k = \overline{x}_{k-1} + \frac{x_k - \overline{x}_{k-1}}{k}, \qquad S_k = S_{k-1} + (x_k - \overline{x}_{k-1})(x_k - \overline{x}_k), \tag{3.12}$$

  and is the method of choice for streaming data, where a two-pass computation is impossible.

```
var_shifted <- function(x) { n <- length(x); d <- x - x[1]; (sum(d^2) - sum(d)^2 / n) / (n - 1)
 ↪  }

var_welford <- function(x) {
  m <- 0; S <- 0
  for (k in seq_along(x)) { m_old <- m; m <- m + (x[k] - m) / k; S <- S + (x[k] - m_old) *
   ↪  (x[k] - m) }
  S / (length(x) - 1) }

x <- c(1, 2, 3) + 1e10
c(one_pass = var1(x), shifted = var_shifted(x), welford = var_welford(x), two_pass = var2(x))
```

```
one_pass  shifted  welford two_pass
  -32768        1        1        1
```

3.3 Methods for Mitigating Overflow and Underflow Errors

3.3.1 Where overflow and underflow arise in statistics

Likelihoods are **products of many probabilities**, and normalizing constants involve **factorials** or **exponentials**:

$$\prod_{i=1}^{2000} p(x_i) = \left(\tfrac{1}{2}\right)^{2000} \approx 10^{-602} \quad \rightarrow \quad \text{underflow to } 0, \tag{3.13}$$

$$1000! \approx 4 \times 10^{2567} \quad \rightarrow \quad \text{overflow to } \infty, \tag{3.14}$$

$$\text{logistic: } \frac{e^x}{1+e^x} \text{ at } x = 1000 \quad \rightarrow \quad \frac{\infty}{\infty} = \texttt{NaN}. \tag{3.15}$$

Two remedies are available: change the algebraic expression so that the intermediate quantities stay in range, or work on the logarithmic scale.

3.3.2 Changing the algebraic expression

Two equivalent forms of the logistic (sigmoid) function:

$$\sigma(x) = \frac{e^x}{1+e^x} = \frac{1}{1+e^{-x}}. \tag{3.16}$$

x	$e^x/(1+e^x)$	$1/(1+e^{-x})$
$+1000$	$e^x = \infty$, \$∞/∞= \$ `NaN` ✗	$e^{-x} = 0$, $1/1 = 1$ ✓
-1000	$e^x = 0$, $0/1 = 0$ ✓	$e^{-x} = \infty$, $1/\infty = 0$ ✓

Evaluating the first form at a large positive value yields ∞/∞; the second form lets e^{-x} underflow harmlessly to 0 and computes $1/(1+0) = 1$. In this case the second form is safe at both ends, because a division by ∞ gives 0 correctly. In general one picks the form in which the exponential has a **non-positive** argument:

$$\sigma(x) = \begin{cases} 1/(1+e^{-x}) & x \geq 0 \\ e^x/(1+e^x) & x < 0. \end{cases} \tag{3.17}$$

```
p1 <- function(theta) exp(theta) / (1 + exp(theta))
p2 <- function(theta) 1 / (1 + exp(-theta))

## large positive theta breaks p1, works in p2
theta <- 2000
c(p1 = p1(theta), p2 = p2(theta))
```

```
 p1  p2
NaN   1
```

```r
## large negative theta works for both
theta <- -2000
c(p1 = p1(theta), p2 = p2(theta))
```

```
p1 p2
 0  0
```

```r
## the piecewise form is safe everywhere and keeps full precision in both tails
sigmoid <- function(x) ifelse(x >= 0, 1 / (1 + exp(-x)), exp(x) / (1 + exp(x)))
sigmoid(c(-2000, -20, 0, 20, 2000))
```

```
[1] 0.00000000000000000000e+00 2.06115361819020332521e-09
[3] 5.00000000000000000000e-01 9.99999979388463078323e-01
[5] 1.00000000000000000000e+00
```

Built-in functions. Software provides functions written specifically to avoid cancellation and overflow, and they should be used in preference to the naive composition:

instead of	use	reason
`log(1 + x)`	`log1p(x)`	for tiny x, `1 + x` rounds to 1
`exp(x) - 1`	`expm1(x)`	for tiny x, `exp(x)` rounds to 1
`log(factorial(n))`	`lfactorial(n)`, `lgamma(n + 1)`	`factorial(n)` overflows
`log(choose(n, k))`	`lchoose(n, k)`	overflow
`log(dnorm(x))`	`dnorm(x, log = TRUE)`	`dnorm(x)` underflows
`log(pnorm(x))`	`pnorm(x, log.p = TRUE)`	underflow
`log(1 - pnorm(x))`	`pnorm(x, lower.tail = FALSE, log.p = TRUE)`	cancellation and underflow

```r
c(naive = log(factorial(1000)), lfactorial = lfactorial(1000))
```

```
                naive             lfactorial
                  Inf 5912.128178488163939619
```

```r
c(naive = log(dnorm(40)), dnorm_log = dnorm(40, log = TRUE))
```

```
                naive               dnorm_log
                 -Inf -800.9189385332047095289
```

```r
c(naive = log(1 - pnorm(10)), stable = pnorm(10, lower.tail = FALSE, log.p = TRUE))
```

```
                naive                   stable
                 -Inf -53.23128515051246978373
```

The last line shows both failures at once: `1 - pnorm(10)` is `1 - 1 = 0` in double precision, so the naive answer is `-Inf`, while the true value is about -53.2.

3.3.3 Representing numbers in logarithmic form

Represent a positive number x by $\ell_x = \log x$:

$$x = e^{2000} \iff \ell_x = 2000, \qquad x = e^{-2000} \iff \ell_x = -2000. \tag{3.18}$$

Numbers that overflow or underflow in ordinary form are ordinary numbers on the log scale. The cost is that each operation f must be **translated** to the log scale:

$$x_1, \dots, x_n \xrightarrow{f} y \qquad \iff \qquad \ell_1, \dots, \ell_n \xrightarrow{\tilde{f}} \ell_y = \log f(e^{\ell_1}, \dots, e^{\ell_n}). \tag{3.19}$$

Products and quotients are the easy case:

$$\log(x_1 x_2) = \ell_1 + \ell_2, \qquad \log\left(\prod_{i=1}^{n} x_i\right) = \sum_{i=1}^{n} \ell_i, \qquad \log \frac{x_1}{x_2} = \ell_1 - \ell_2. \tag{3.20}$$

In particular $\log \prod_i p(x_i) = \sum_i \log p(x_i)$: always compute the log-likelihood, never the likelihood.

```
log_x <- 800; x <- exp(log_x); x          # overflow
```

```
[1] Inf
```

```
log_y <- 805; y <- exp(log_y); y          # overflow
```

```
[1] Inf
```

```
x / y                                     # Inf/Inf = NaN
```

```
[1] NaN
```

```
log_xovery <- log_x - log_y               # subtracting logs is safe
log_xovery
```

```
[1] -5
```

```
exp(log_xovery)
```

```
[1] 0.006737946999085467000845
```

```
## likelihood versus log-likelihood for 2000 fair-coin tosses
p <- rep(0.5, 2000)
c(likelihood = prod(p), log_likelihood = sum(log(p)))
```

```
            likelihood         log_likelihood
0.00000000000000000 -1386.29436111982249713
```

3.3.4 The log-sum-exp trick

Sums are the hard case: $\log(x_1 + x_2) \neq \ell_1 + \ell_2$. We need

$$\log\left(\sum_{i=1}^{n} x_i\right) = \log\left(\sum_{i=1}^{n} e^{\ell_i}\right), \tag{3.21}$$

but the e^{ℓ_i} may overflow or underflow. Let $m = \max_i \ell_i$ and factor it out:

$$\log\left(\sum_{i=1}^{n} e^{\ell_i}\right) = \log\left(e^{m}\sum_{i=1}^{n} e^{\ell_i - m}\right) = m + \log\left(\sum_{i=1}^{n} e^{\ell_i - m}\right). \tag{3.22}$$

Every $\ell_i - m \leq 0$, so $e^{\ell_i - m} \in (0, 1]$ and nothing overflows; the largest term is $e^0 = 1$, so the sum is at least 1 and cannot underflow. Terms far below the maximum underflow to 0 individually, but they would have contributed nothing at 16-digit precision anyway.

Illustration with $x_i = e^{2000}, e^{2001}, e^{2002}$, each of which overflows to `Inf`:

	$i = 1$	$i = 2$	$i = 3$
ℓ_i	2000	2001	2002
$\ell_i - m$ ($m = 2002$)	-2	-1	0
$e^{\ell_i - m}$	0.135	0.368	1

$$\log\sum_i x_i = 2002 + \log(0.135 + 0.368 + 1) = 2002.408 \tag{3.23}$$

```
log_sum_exp <- function(log_x) {
  max_log_x <- max(log_x)
  max_log_x + log(sum(exp(log_x - max_log_x)))
}

## values that would overflow directly
log_x <- c(2000, 2010, 2030)
exp(log_x)                                  # Inf
```

```
[1] Inf Inf Inf
```

```
c(direct = log(sum(exp(log_x))), lse = log_sum_exp(log_x))
```

```
direct                    lse
   Inf 2030.000000002061142368
```

```
## values that would underflow directly
log_x <- -c(2000, 2010, 2030)
exp(log_x)                                  # 0
```

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

```
c(direct = log(sum(exp(log_x))), lse = log_sum_exp(log_x))
```

```
                direct                    lse
  -Inf -1999.99995460110062595
```

Example: log-likelihood of a mixture. The density of a two-component normal mixture is $f(y) = \pi_1 \varphi(y; \mu_1, 1) + \pi_2 \varphi(y; \mu_2, 1)$, a sum inside a product. For an observation far from both means, both component densities underflow and the naive log-likelihood is `-Inf`; with log-sum-exp applied observation by observation it is finite and correct.

```
y <- c(-1, 0.5, 2, 45)                        # the last value is far from both means
mu <- c(0, 3); w <- c(0.6, 0.4)
naive <- sum(log(w[1] * dnorm(y, mu[1]) + w[2] * dnorm(y, mu[2])))
stable <- sum(sapply(y, function(yi) log_sum_exp(log(w) + dnorm(yi, mu, log = TRUE))))
c(naive = naive, stable = stable)
```

```
                naive                  stable
  -Inf -889.3332627696936469874
```

Example: log-softmax. The softmax probabilities $p_k = e^{x_k} / \sum_j e^{x_j}$ appear in multinomial logistic regression and neural networks. Their logarithm is $\log p_k = x_k - \log \sum_j e^{x_j}$, and the second term is a log-sum-exp.

```
x <- c(1000, 1001, 1005)
naive <- log(exp(x) / sum(exp(x)))
stable <- x - log_sum_exp(x)
rbind(naive, stable, probabilities = exp(stable))
```

```
                                    [,1]                    [,2]
naive                                NaN                     NaN
stable        -5.02474489013877700926968 -4.02474489013877700926968
probabilities  0.00657326318530938261824  0.01786798187030531892553
                                    [,3]
naive                                NaN
stable        -0.02474489013877700926969
probabilities  0.97555875494443089479546
```

Software implementation. The log-sum-exp trick is already implemented, tested, and optimized in the **matrixStats** package (a dependency of hundreds of CRAN/Bioconductor packages), so production code should call it rather than the version above:

```
library(matrixStats)
c(mine = log_sum_exp(c(2000, 2001, 2002)), matrixStats = logSumExp(c(2000, 2001, 2002)))
```

```
                mine            matrixStats
2002.407605964444428537 2002.407605964444428537
```

3.3.5 Differences on the log scale

For $x > y > 0$ with $\ell_x = \log x$, $\ell_y = \log y$:

$$\log(x - y) = \log\left(e^{\ell_x}\left(1 - e^{\ell_y - \ell_x}\right)\right) = \ell_x + \log\left(1 - e^{\ell_y - \ell_x}\right). \tag{3.24}$$

Since $\ell_y - \ell_x < 0$, $e^{\ell_y - \ell_x} \in (0, 1)$ and there is no overflow. When $x \approx y$, the factor $1 - e^{\ell_y - \ell_x}$ is itself a large − large subtraction; `log1p(-exp(ly - lx))` avoids it.

```
log_minus_exp <- function(log_x, log_y) {
  if (log_x < log_y) stop("the first argument must be the larger")
  log_x + log1p(-exp(log_y - log_x))
}

log_minus_exp(2020, 2000)
```

```
[1] 2019.999999997938857632
```

```
exp(2000) - exp(1999)                          # direct subtraction: Inf - Inf = NaN
```

```
[1] NaN
```

```
## nearly equal arguments: log1p keeps the precision that log(1 - .) would lose
lx <- 10; ly <- 10 - 1e-12
c(with_log1p = log_minus_exp(lx, ly), with_log = lx + log(1 - exp(ly - lx)), exact =
 ↪  log(exp(lx) - exp(ly)))
```

```
            with_log1p                with_log                   exact
-17.630932219297630325 -17.630932219297630325 -17.630788066160636873 29
```

A statistical use of the difference formula is the survival function of a discrete distribution, $\log P(X > k) = \log\left(1 - P(X \le k)\right)$, when $P(X \le k)$ is available only as a log-probability close to 0.

Four extreme choices of a, b show that every one of the four failure modes met in this chapter — overflow, cancellation, underflow, and absorption — can destroy a naive `log(exp(a) - exp(b))`, and that `log_minus_exp` survives all four because it never forms `exp(a)` or `exp(b)` when they would be out of range:

```
a <- c(2000, 1e-20, -1000,    0)
b <- c(1999,     0, -1001, -500)
g <- function(x) formatC(x, format = "g", digits = 8)
kable(data.frame(a = g(a), b = g(b), u = g(a - b),
                 naive = g(log(exp(a) - exp(b))),
                 log_minus_exp = g(mapply(log_minus_exp, a, b))),
      caption = "Shows why log(exp(a) - exp(b)) should be computed on the log scale: the naive
       ↪  expression overflows for large a and b, whereas log_minus_exp stays finite and
       ↪  accurate. log(exp(a) - exp(b)) by the naive expression and by log_minus_exp",
      align = c("r", "r", "r", "r", "r"))
```

Table 3.5: Shows why log(exp(a) - exp(b)) should be computed on the log scale: the naive expression overflows for large a and b, whereas log_minus_exp stays finite and accurate. log(exp(a) - exp(b)) by the naive expression and by log_minus_exp

a	b	u	naive	log_minus_exp
2000	1999	1	NaN	1999.5413
1e-20	0	1e-20	-Inf	-Inf
-1000	-1001	1	-Inf	-1000.4587
0	-500	500	0	-7.1245764e-218

Reading down the naive column: $\log(e^{2000} - e^{1999})$ is `Inf - Inf`, `NaN` (overflow); $\log(e^{10^{-20}} - e^{0})$ is `1 - 1`, so $-\infty$ (cancellation, since $u = 10^{-20}$ is tiny); $\log(e^{-1000} - e^{-1001})$ is `0 - 0`, so $-\infty$ (underflow, even though $u = 1$ is not small); and $\log(e^{0} - e^{-500})$ silently returns 0 because $e^{-500} \approx 7 \times 10^{-218}$ is absorbed into 1 before the subtraction (absorption). `log_minus_exp` is correct in all four rows, because it factors out e^{a} and reduces every case to `log1p` evaluated at a safely bounded argument — even recovering the tiny but nonzero last entry. The same function is available as `DPQ::logspace.sub`, with `DPQ::logspace.add` as the corresponding stable version of $\log(e^{a} + e^{b})$ for a single pair (`logSumExp` above is the version for a whole vector).

3.3.6 Computing $\log(1 - e^{-u})$: large + small and large − large together

The log of a difference. As in the log-sum-exp trick, factor out the larger term; a difference taken on the log scale then reduces to a single kernel: for $a > b$,

$$\log(e^{a} - e^{b}) = a + \log(1 - e^{-u}), \qquad u = a - b > 0. \tag{3.25}$$

The term a is exact, so computing the log of a difference accurately comes down to evaluating $\log(1 - e^{-u})$ accurately for every $u > 0$.

Where the literal formula fails, and the remedy for each case. Written literally as `log(1 - exp(-u))`, the kernel fails at **both** ends of the range, from the two failure modes introduced above. Each failure has its own remedy among R's functions for arguments near the landmark 1: `expm1(x)` computes $e^{x} - 1$, and `log1p(x)` computes $\log(1 + x)$, both with full relative precision for small x (Section on avoiding large − large, above).

1. u **small: large − large.** Here $e^{-u} \approx 1$, so $1 - e^{-u}$ subtracts two nearly equal numbers. The true value is $\approx u$, far below the roundoff already present in e^{-u}, so the leading digits cancel and little survives; for $u < 2^{-53}$, `exp(-u)` itself rounds to exactly 1 and the result is $-\infty$.

   Remedy: compute the difference with `expm1`, as `log(-expm1(-u))`. The function `expm1` never forms $1 - e^{-u}$; it evaluates the series directly,

   $$-\operatorname{expm1}(-u) = 1 - e^{-u} = u - \frac{u^{2}}{2} + O(u^{3}), \qquad \text{so} \qquad \log(1 - e^{-u}) \approx \log u \quad (u \to 0), \tag{3.26}$$

   and the outer `log` then receives a value near u, far from the landmark 1, which is safe.

2. u **large: large + small.** Here e^{-u} is tiny, so in $1 - e^{-u}$ the small term is absorbed: `1 - exp(-u)` rounds to exactly 1, and `log` returns 0 instead of the true value $\approx -e^{-u}$.

   Remedy: compute the log with `log1p`, as `log1p(-exp(-u))`. The function `log1p` never forms $1 + x$ at $x = -e^{-u}$; it evaluates the series directly,

$$\operatorname{log1p}(-e^{-u}) = \log(1 - e^{-u}) = -e^{-u} - \frac{e^{-2u}}{2} + O(e^{-3u}) \approx -e^{-u} \quad (u \to \infty). \tag{3.27}$$

Each remedy removes only **its own** failure, and neither removes the other:

```
f_naive <- function(u) log(1 - exp(-u))      # both failures
f_expm1 <- function(u) log(-expm1(-u))       # never forms the difference 1 - exp(-u)
f_log1p <- function(u) log1p(-exp(-u))       # never forms the sum exp(-u)
```

Neither remedy carries over to the other side. For large u, `f_expm1` returns $-\operatorname{expm1}(-u) = 1$ exactly and its outer `log` gives 0, just as `f_naive` does. For small u, the argument $-e^{-u}$ of `f_log1p` has already rounded to -1 before `log1p` ever sees it, so `f_log1p` fails exactly where `f_naive` does.

Table 3.6 compares the three formulas with the branched solution `correct` derived below. Numbers below 0.01 in absolute value are written as $\pm e^{x}$, meaning $\pm\exp(x)$, so that for example $\log u$ and $-e^{-u}$ can be read off directly from the entries.

```
library(gt)
u <- c(500, 50, 20, 1, 2, exp(-10),  exp(-20), exp(-500))   # large u at the top, small u at
 ↪  the bottom
log1mexp <- function(u) ifelse(u <= log(2), log(-expm1(-u)), log1p(-exp(-u)))
m <- function(tex) paste0("\\(", tex, "\\)")   # inline math, typeset by MathJax
## plain decimals, except that numbers below 0.01 in absolute value are written +/- e^{x}, x =
 ↪  log|value|
g <- function(v) vapply(v, function(v) {
  if (is.nan(v)) return("NaN")
  if (is.infinite(v)) return(m(if (v < 0) "-\\infty" else "\\infty"))
  if (v == 0) return(m("0"))
  if (abs(v) < 0.01) return(m(paste0(if (v < 0) "-", "e^{", signif(log(abs(v)), 6), "}")))
  m(signif(v, 6))
}, "")
tab <- data.frame(u = g(u), naive = g(f_naive(u)), expm1 = g(f_expm1(u)),
                  log1p = g(f_log1p(u)), correct = g(log1mexp(u)))
gt(tab) |>
  fmt_passthrough(everything(), escape = FALSE) |>
  cols_label(u       = html(m("u")),
             naive   = html(paste("naive:", m("\\log(1-e^{-u})"))),
             expm1   = html(paste("expm1:", m("\\log(-\\mathrm{expm1}(-u))"))),
             log1p   = html(paste("log1p:", m("\\mathrm{log1p}(-e^{-u})"))),
             correct = html(paste("correct:", m("\\mathrm{log1mexp}(u)")))) |>
  cols_align("right") |>
  tab_footnote(
    footnote = html(paste0("For small ", m("u"), ": ",
      m("-\\mathrm{expm1}(-u) = u - \\frac{u^2}{2} + O(u^3)"), ", so ",
```

Table 3.6: Computing log(1 - exp(-u)) four ways, across eighteen orders of magnitude of u; numbers below 0.01 in absolute value are written as powers of e.

u	naive: $\log(1 - e^{-u})$	expm1: $\log(-\text{expm1}(-u))^*$	log1p: $\text{log1p}(-e^{-u})^\dagger$	correct: $\text{log1mexp}(u)$
500	0	0	$-e^{-500}$	$-e^{-500}$
50	0	0	$-e^{-50}$	$-e^{-50}$
20	$-e^{-20}$	$-e^{-20}$	$-e^{-20}$	$-e^{-20}$
1	-0.458675	-0.458675	-0.458675	-0.458675
2	-0.145413	-0.145413	-0.145413	-0.145413
e^{-10}	-10	-10	-10	-10
e^{-20}	-20	-20	-20	-20
e^{-500}	$-\infty$	-500	$-\infty$	-500

*For small u: $-\text{expm1}(-u) = u - \frac{u^2}{2} + O(u^3)$, so $\log(1 - e^{-u}) \approx \log\left(u - \frac{u^2}{2}\right) \approx \log u$ as $u \to 0$.
†For large u: $\text{log1p}(-e^{-u}) = -e^{-u} - \frac{e^{-2u}}{2} + O(e^{-3u})$, so $\log(1 - e^{-u}) \approx -e^{-u}$ as $u \to \infty$.

```
    m("\\log(1-e^{-u}) \\approx \\log\\big(u - \\frac{u^2}{2}\\big) \\approx \\log u"),
    " as ", m("u \\to 0"), ".")),
  locations = cells_column_labels(expm1)) |>
tab_footnote(
  footnote = html(paste0("For large ", m("u"), ": ",
    m("\\mathrm{log1p}(-e^{-u}) = -e^{-u} - \\frac{e^{-2u}}{2} + O(e^{-3u})"), ", so ",
    m("\\log(1-e^{-u}) \\approx -e^{-u}"), " as ", m("u \\to \\infty"), ".")),
  locations = cells_column_labels(log1p)) |>
opt_footnote_marks("standard") |>
opt_row_striping()
```

- For **small** u, `naive` tracks `log1p`: both lose digits and eventually collapse to $-\infty$; only `expm1` is correct.
- For **large** u, `naive` tracks `expm1`: both collapse to 0; only `log1p` is correct.
- In the **middle** ($10^{-8} \leq u \leq 20$), all three agree, because u is nowhere near either landmark.

Solution: branch between the two remedies. Split at $u = \log 2$, where $e^{-u} = 1/2$ is safely far from both traps, and use whichever incomplete remedy is valid on that side:

```
log1mexp <- function(u) ifelse(u <= log(2), log(-expm1(-u)), log1p(-exp(-u)))
```

The same construction, named `log1mexp`, is shipped by several CRAN packages (including **copula** and **VGAM**), all tracing to Mächler (2012); `DPQ::logspace.sub(0, -u)` computes the same quantity by an equivalent route. Prefer a package function in production code, and write your own only to understand *why* it works.

3.3.7 A combined example: $\log\{1 - \Phi(x)\}$ at both tails

The rounding fix `log1mexp` and the underflow fix of working on the log scale are often needed **together**. Consider $\log\{1 - \Phi(x)\}$, where Φ is the standard normal CDF (`pnorm`), over a range of x from the far left tail to the far

Table 3.7: Computing log(1 - Phi(x)) from l = log Phi(x) = pnorm(x, log.p = TRUE); numbers below 0.01 in absolute value are written as powers of e.

x	$\ell = \log \Phi(x)$	naive: $\log(1 - e^{\ell})$	expm1: $\log(-\text{expm1}(\ell))$	log1p: $\text{log1p}(-e^{\ell})$	log1mexp: $\text{log1mexp}(-\ell)$	pnorm: <code>lower.tail = FALSE</code>
-38	-726.557	0	0	$-e^{-726.557}$	$-e^{-726.557}$	$-e^{-726.557}$
-20	-203.917	0	0	$-e^{-203.917}$	$-e^{-203.917}$	$-e^{-203.917}$
-10	-53.2313	0	0	$-e^{-53.2313}$	$-e^{-53.2313}$	$-e^{-53.2313}$
-5	-15.065	$-e^{-15.065}$	$-e^{-15.065}$	$-e^{-15.065}$	$-e^{-15.065}$	$-e^{-15.065}$
-1	-1.84102	-0.172754	-0.172754	-0.172754	-0.172754	-0.172754
0	-0.693147	-0.693147	-0.693147	-0.693147	-0.693147	-0.693147
1	-0.172754	-1.84102	-1.84102	-1.84102	-1.84102	-1.84102
5	$-e^{-15.065}$	-15.065	-15.065	-15.065	-15.065	-15.065
10	$-e^{-53.2313}$	$-\infty$	-53.2313	$-\infty$	-53.2313	-53.2313
20	$-e^{-203.917}$	$-\infty$	-203.917	$-\infty$	-203.917	-203.917
38	$-e^{-726.557}$	$-\infty$	-726.557	$-\infty$	-726.557	-726.557

right tail. Computing $\Phi(x)$ itself and then $1 - \Phi(x)$ fails immediately for large x: `pnorm(10)` already rounds to exactly 1, and no reformulation of the outer expression can recover digits lost inside `pnorm`. So we ask R for the log, $\ell = \log \Phi(x) =$ `pnorm(x, log.p = TRUE)`, which R returns accurately for every x, and then take the complement on the log scale:

$$\log\{1 - \Phi(x)\} = \log(1 - e^{\ell}) = \log(1 - e^{-u}), \qquad u = -\ell > 0. \tag{3.28}$$

This is exactly the kernel of the previous section, so the same four formulas apply, with $e^{-u} = e^{\ell}$:

```
x <- c(-38, -20, -10, -5, -1, 0, 1, 5, 10, 20, 38)
l <- pnorm(x, log.p = TRUE)                                  # log Phi(x), accurate for all x
tab <- data.frame(x       = g(x),
                  l       = g(l),
                  naive   = g(log(1 - exp(l))),
                  expm1   = g(log(-expm1(l))),
                  log1p   = g(log1p(-exp(l))),
                  log1mexp = g(log1mexp(-l)),
                  pnorm   = g(pnorm(x, lower.tail = FALSE, log.p = TRUE)))  # R's direct upper
                   ↪ tail
gt(tab) |>
  fmt_passthrough(everything(), escape = FALSE) |>
  cols_label(x       = html(m("x")),
             l       = html(m("\\ell = \\log\\Phi(x)")),
             naive   = html(paste("naive:", m("\\log(1-e^{\\ell})"))),
             expm1   = html(paste("expm1:", m("\\log(-\\mathrm{expm1}(\\ell))"))),
             log1p   = html(paste("log1p:", m("\\mathrm{log1p}(-e^{\\ell})"))),
             log1mexp = html(paste("log1mexp:", m("\\mathrm{log1mexp}(-\\ell)"))),
             pnorm   = html(paste("pnorm:", "<code>lower.tail = FALSE</code>"))) |>
  cols_align("right") |>
  opt_row_striping()
```

- **Left tail** ($x \le -10$). $\Phi(x)$ is tiny, so ℓ is large and negative and $u = -\ell$ is **large**. `naive` and `expm1` both return 0: the tiny e^{ℓ} is absorbed in $1 - e^{\ell}$ (large + small). `log1p` is correct, and the true value is $\approx -e^{\ell} = -\Phi(x)$, a tiny negative number.
- **Right tail** ($x \ge 10$). $\Phi(x)$ is within roundoff of 1, so $\ell \approx -\{1 - \Phi(x)\}$ is a tiny negative number and u is **small**. `naive` and `log1p` both return $-\infty$: e^{ℓ} rounds to exactly 1 before the subtraction (large − large).

`expm1` is correct, giving $\log u \approx \log\{1 - \Phi(x)\}$, a large negative number (≈ -53.23 at $x = 10$). Note that the accuracy here depends on starting from ℓ: had we computed `pnorm(x)` instead, the $1 - \Phi(x)$ digits would already have been lost.
- **Middle** ($|x| \leq 5$). All formulas agree.
- `log1mexp` branches at $u = \log 2$ and matches the correct remedy in every row; R's `pnorm(x, lower.tail = FALSE, log.p = TRUE)` computes $\log\{1 - \Phi(x)\} = \log \Phi(-x)$ directly, never forming $\Phi(x)$, and agrees with `log1mexp` throughout.

The general lesson is to stay on the log scale throughout: ask R for $\log \Phi(x)$ rather than $\Phi(x)$, and take the complement with `log1mexp`, which selects the correct branch automatically so that the tail direction never has to be diagnosed by hand. When a distribution function offers `lower.tail = FALSE`, as `pnorm` does, use it directly.

3.4 Summary

- Integers occupy a fixed number of bits; exceeding $2^{31} - 1$ gives `NA` in R.
- Doubles store a sign, a 52-bit mantissa (about 16 decimal digits), and an 11-bit exponent (magnitudes 10^{-308} to 10^{308}). Neighbouring doubles are spaced at a relative distance of about $\epsilon = 2^{-52}$; integers above 2^{53} are not all representable.
- Overflow produces `Inf`, underflow produces `0`; the latter is silent and therefore more dangerous.
- Roundoff error is amplified by adding large + small (fix: add small terms first, pairwise or compensated summation) and by subtracting large − large (fix: reformulate so that same-sign quantities of similar size are combined, as in $e^{-x} = 1/e^x$ and the two-pass variance). Some quantities, such as $\log(1 - e^{-u})$, suffer from **both** failure modes at opposite ends of their range and need a function like `log1mexp` that branches between two incomplete remedies (`expm1`, `log1p`).
- Overflow and underflow are avoided by rewriting expressions so that exponentials have non-positive arguments, by using purpose-built functions (`log1p`, `expm1`, `lgamma`, `dnorm(log = TRUE)`), and by working on the log scale, where products become sums and sums are handled by the log-sum-exp trick (`log_sum_exp`/`matrixStats::logSumExp` for a vector, `log_minus_exp`/`DPQ::logspace.sub` for a difference of two).
- Rounding and underflow fixes often compose: computing $\log\{1 - \Phi(x)\}$ correctly at both tails needs `pnorm(x, log.p = TRUE)` to avoid underflow *and* `log1mexp` to avoid cancellation, in either order depending on the sign of x.

3.5 Exercises

Question 1. How do you compute

$$\log\left(\frac{1}{1 + e^{-x}}\right), \tag{3.29}$$

so as to avoid overflow and underflow of e^{-x}, for both large positive and large negative x? (Hint: write it as $-\log(1 + e^{-x})$ and consider `log1p` and the two forms of the sigmoid.)

Question 2 (log-softmax). How do you compute

$$\log\left(\frac{e^{x_k}}{\sum_{j=1}^{K} e^{x_j}}\right), \tag{3.30}$$

so as to avoid underflow and overflow of e^{x_j}? Verify your formula against the `log-softmax` example above for $x = (1000, 1001, 1005)$ and for $x = (-1000, -1001, -1005)$.

Question 3. The one-pass variance formula fails for $x = (M+1, M+2, M+3)$ with $M = 10^8$. Predict the smallest M (a power of 10) at which the one-pass result in R first differs from 2, then verify using the scan in the variance section.

Question 4. Write a function `log_dbinom(k, n, p)` that returns $\log \binom{n}{k} p^k (1-p)^{n-k}$ for $n = 10^5$ without overflow or underflow, using `lchoose` and `log1p`, and compare it with `dbinom(k, n, p, log = TRUE)`.

Question 5. Implement Welford's algorithm for the sample covariance of two variables and check it against `cov()` on data shifted by 10^{10}.

Question 6. For a Poisson(λ) random variable, $\log P(X > 0) = \log(1 - e^{-\lambda})$. Using `log1mexp`, compute this for $\lambda = 10^{-12}$ and for $\lambda = 700$, and explain in each case which of `expm1` or `log1p` alone would have failed and why.

4 Random Numbers and Monte Carlo Methods

This chapter introduces Monte Carlo methods in two stages. The first half asks how a computer produces "random" numbers at all, and how those numbers are transformed into draws from any distribution we like. The second half puts those draws to work: once we can simulate data, we can estimate almost any quantity — the risk of an estimator, the size or power of a hypothesis test, a p-value with no closed form — simply by repeating an experiment many times and averaging.

4.1 Monte Carlo Methods

4.1.1 Pseudo Random Numbers

A computer is deterministic, so it cannot produce truly random numbers. It produces **pseudo random numbers**: a deterministic sequence that *behaves* like independent Unif$(0, 1)$ draws. One of the most common methods is the **Linear Congruential Generator (LCG)**. It produces a sequence of pseudo-random integers X_n defined by the recurrence relation:

$$X_n = (aX_{n-1} + c) \pmod{M} \tag{4.1}$$

where X_0 is the seed, a is the multiplier, c is the increment, and M is the modulus. To get numbers in the interval $[0, 1]$, we divide by $M - 1$. In this example, we use the parameters $a = 7^5$, $c = 0$, and $M = 2^{31} - 1$ (a minimal standard LCG).

- The starting value X_0 is the **seed**. The same seed gives the same sequence, which makes simulations reproducible.
- The sequence eventually repeats. A good generator has a very long period and passes statistical tests of uniformity and independence.
- R's default generator is the Mersenne Twister, with period $2^{19937} - 1$. All other distributions in R are built from Unif$(0, 1)$ draws.

```
A <- 7^5
M <-  2^31-1

N <- 500
rn <- rep (0, N)
rn[1] <- 10 # Initial seed
for (i in 2:length (rn))
{
    rn[i] <- (A * rn[i-1] ) %% M
```

```r
}

## Scale the numbers to fall between 0 and 1
nrn <- rn/(M-1)

## Compare with R's built-in runif()
n <- 500
a <- runif(n)

par(mfrow=c(2,3),mar=c(4,4,3,1))
plot(nrn[1:100], main="Trace Plot", ylab="Value", xlab="Index")
acf(nrn, main="Autocorrelation")
hist(nrn, main="Histogram")
plot.new()
hist(a,xlab="Random Numbers",main="Built-in runif()")
acf(a,main="Built-in acf")
```

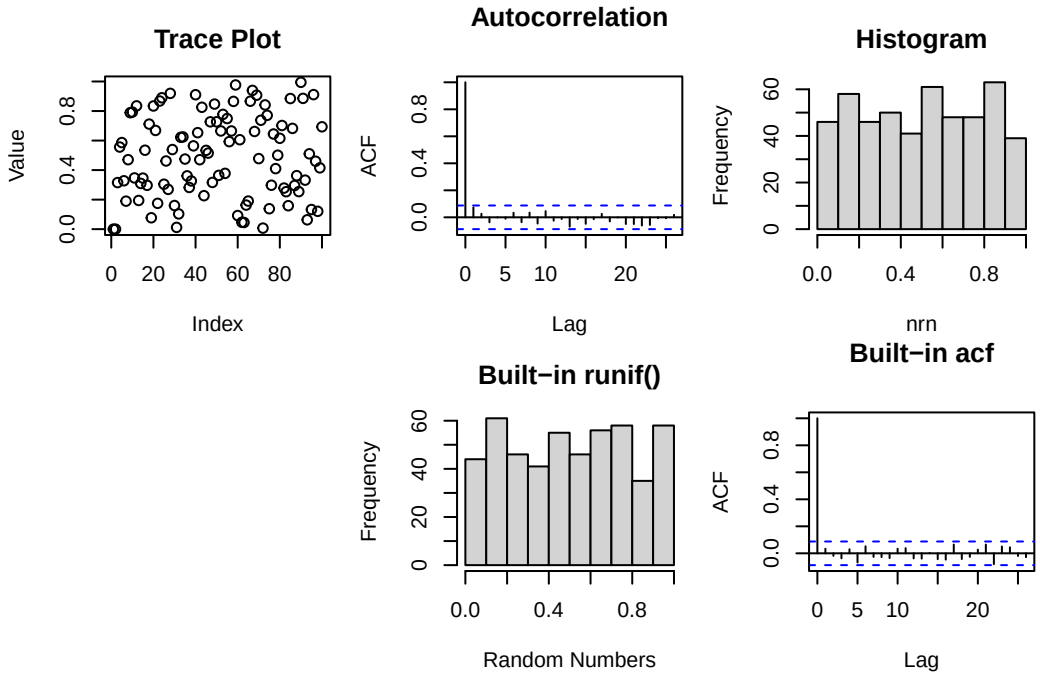


Figure 4.1: Checks the quality of a pseudo-random number generator written from scratch by comparing it with R's built-in generator on three diagnostics: behaviour over time, autocorrelation, and uniformity. A hand-rolled linear congruential generator versus R's built-in `runif()`: trace plot, autocorrelation, and histogram for each.

The trace shows no visible trend, drift or periodicity, the autocorrelations are negligible, and the histogram is flat. This is what "behaves like iid uniform" means in practice, although a convincing assessment requires formal tests of uniformity and serial correlation rather than a few plots of 500 values.

4.1.2 Seeds and Reproducibility

Setting the seed restarts the generator at a fixed point of its sequence:

```
set.seed(1); runif(3)
```

```
[1] 0.2655087 0.3721239 0.5728534
```

```
set.seed(1); runif(3)      # identical
```

```
[1] 0.2655087 0.3721239 0.5728534
```

```
runif(3)                   # continues the sequence
```

```
[1] 0.9082078 0.2016819 0.8983897
```

Always call `set.seed()` at the top of a simulation study so that its results can be reproduced and debugged.

4.1.3 Generating $\mathrm{Unif}(a, b)$

The simplest transformation of a uniform draw is a change of location and scale. If $U \sim \mathrm{Unif}(0, 1)$, then $X = a + (b - a)\,U \sim \mathrm{Unif}(a, b)$, because

$$P(X \le x) = P\Big(U \le \frac{x - a}{b - a}\Big) = \frac{x - a}{b - a}, \qquad a \le x \le b. \tag{4.2}$$

```
u <- runif(10000); x <- 2 + 3 * u          # Unif(2, 5)
hist(x, breaks = 30, freq = FALSE, main = TeX(r"($a + (b - a) U$ with $a = 2$, $b = 5$)"))
abline(h = 1/3, col = 2, lwd = 2)
```

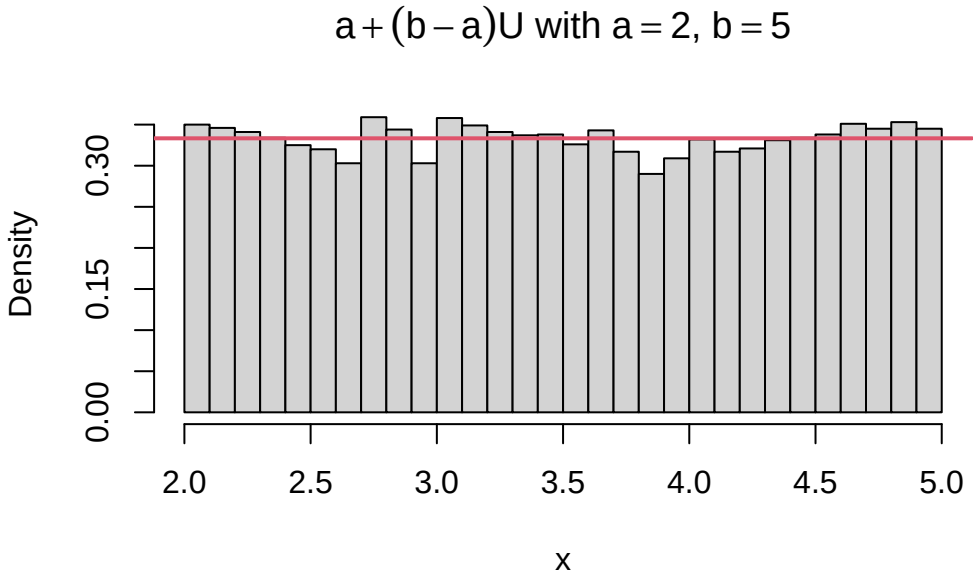


Figure 4.2: Shows the simplest transformation of uniform random numbers, a change of location and scale. Histogram of 10,000 draws of $2 + 3U$ with $U \sim \text{Unif}(0, 1)$, against the $\text{Unif}(2, 5)$ density $1/3$ (red).

4.1.4 Inverting the CDF

If we can generate $U \sim \text{Unif}(0, 1)$, we can generate a random variable X from any continuous distribution using the **inverse transform method**. Let F_X be a continuous CDF with inverse (quantile function) F_X^{-1}.

Theorem. If $U \sim \text{Unif}(0, 1)$, then $X = F_X^{-1}(U)$ has CDF F_X.

Proof.

$$P(X \leq x) = P(F_X^{-1}(U) \leq x) = P(U \leq F_X(x)) = F_X(x). \tag{4.3}$$

Equivalently, by the change-of-variable formula with $u = F_X(x)$,

$$f_X(x) = f_U(F_X(x)) \left| \frac{du}{dx} \right| = 1 \times f(x). \tag{4.4}$$

The converse also holds: if $X \sim F_X$, then $F_X(X) \sim \text{Unif}(0, 1)$. This is the **probability integral transform**, which we will use for p-values in the section on hypothesis testing.

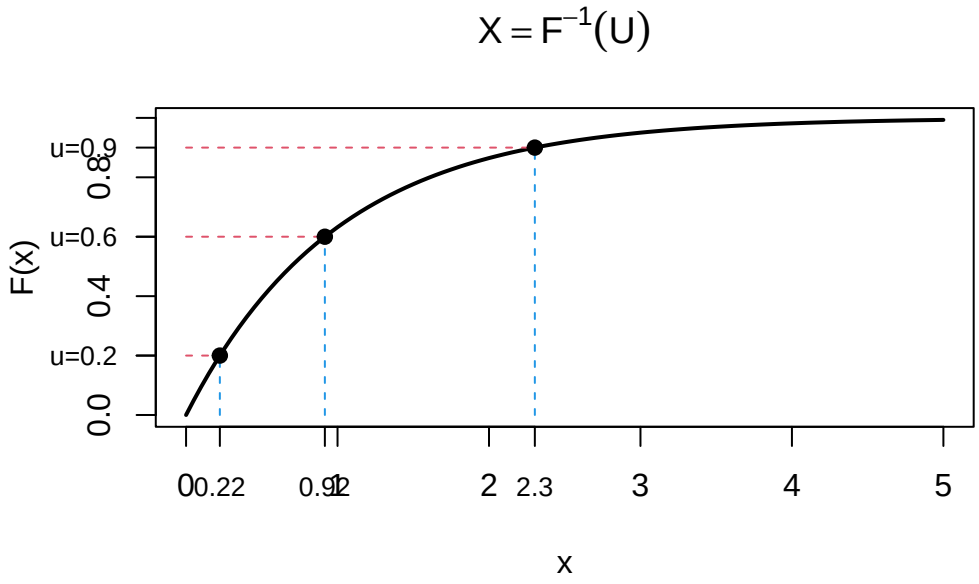


Figure 4.3: Shows geometrically how the inverse-CDF method turns uniform draws into draws from a target distribution. Three uniform values u on the vertical axis are mapped through the $\text{Exp}(1)$ CDF to $x = F^{-1}(u)$ on the horizontal axis; where F is steep (high density), an interval of u values maps to a short interval of x, so more draws land there.

4.1.5 Example: The Exponential Distribution

$X \sim \text{Exp}(\lambda)$ has CDF $F_X(x) = 1 - e^{-\lambda x}$ for $x > 0$. Setting $u = 1 - e^{-\lambda x}$ and solving for x gives

$$x = F_X^{-1}(u) = -\frac{1}{\lambda}\log(1-u). \tag{4.5}$$

Since $1 - U \sim \text{Unif}(0,1)$ as well, we may use $X = -\log(U)/\lambda$.

```r
lambda <- 2
u <- runif(10000)
x <- -log(u) / lambda
hist(x, breaks = 40, freq = FALSE, main = "Exp(2) via the inverse CDF")
curve(dexp(x, lambda), add = TRUE, col = 2, lwd = 2)
c(sample_mean = mean(x), true_mean = 1 / lambda)
```

```
sample_mean   true_mean
  0.4989619   0.5000000
```

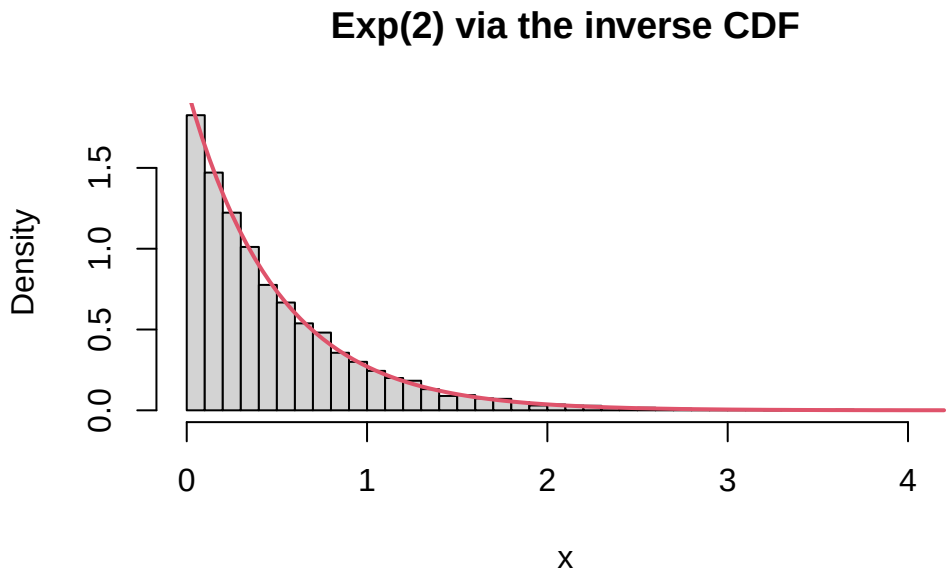


Figure 4.4: Verifies the inverse-CDF method on a distribution whose inverse CDF has a closed form. Histogram of 10,000 draws of $-\log(U)/\lambda$ with $\lambda = 2$, against the $\text{Exp}(2)$ density (red).

4.1.6 Shinylive App for Inverse CDF Sampling

There is a shinylive app to demonstrate the inverse-CDF method for generating random numbers from a target distribution by transforming uniform draws.

4.1.7 A Special Transformation for Generating Normal Samples

The normal CDF Φ has no closed-form inverse, so the inverse-CDF method is not directly usable. The **Box–Muller transform** instead generates *pairs* of independent standard normal variables from pairs of independent uniforms, by working in polar coordinates.

4.1.8 Two Standard Normals in Polar Coordinates

Write a pair $(Z_1, Z_2) \overset{iid}{\sim} N(0,1)$ in **polar coordinates**:

$$Z_1 = R\cos\Theta, \qquad Z_2 = R\sin\Theta, \qquad R^2 = Z_1^2 + Z_2^2, \quad \Theta = \text{atan2}(Z_2, Z_1). \tag{4.6}$$

The joint density $\frac{1}{2\pi}e^{-(z_1^2+z_2^2)/2}$ depends on (z_1, z_2) only through $r^2 = z_1^2 + z_2^2$, so it is **rotationally symmetric** about the origin. Consequently:

- The angle $\Theta \sim \text{Unif}(0, 2\pi)$, independently of R.
- The squared radius $R^2 = Z_1^2 + Z_2^2 \sim \chi^2_2$, and χ^2_2 **is** the exponential distribution with rate $1/2$:

$$f_{\chi^2_2}(w) = \frac{1}{2}e^{-w/2}, \quad w > 0 \qquad \Longrightarrow \qquad R^2 \sim \text{Exp}(\text{rate} = 1/2). \tag{4.7}$$

Both R^2 and Θ **can** be generated by inverting a CDF.

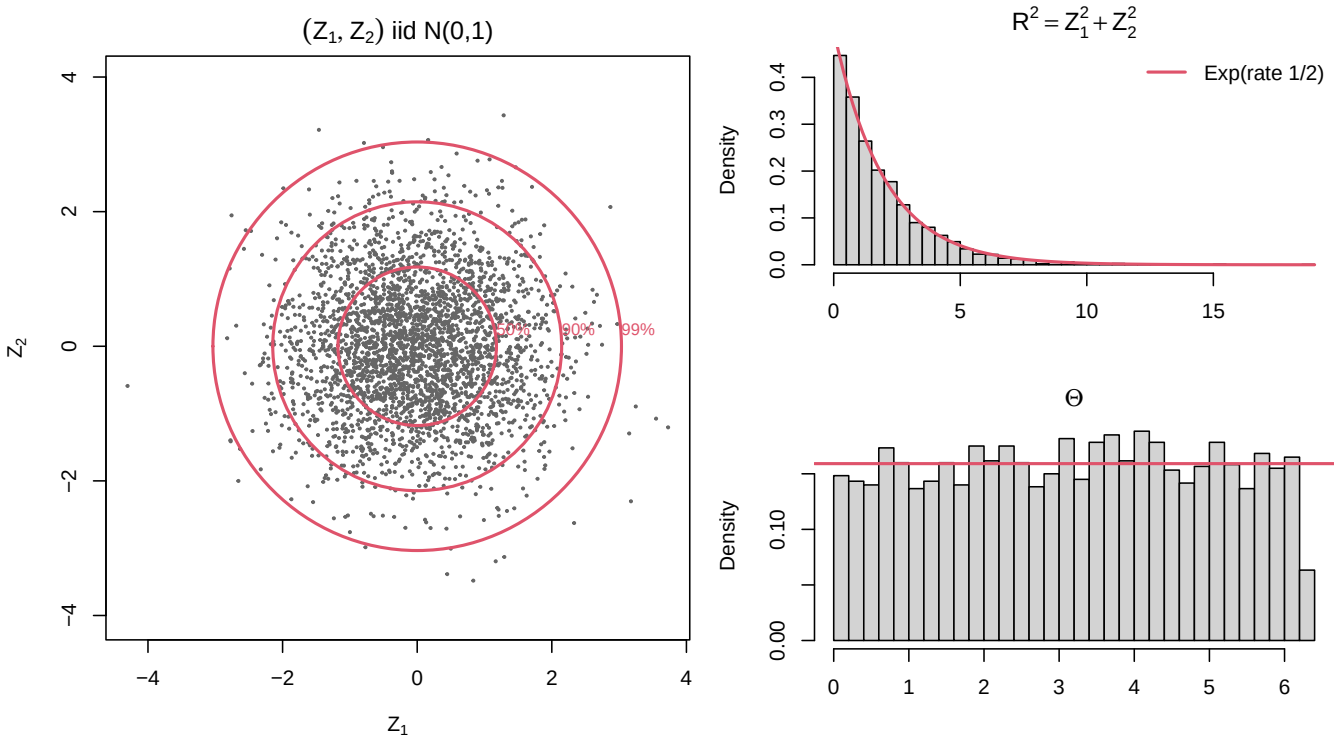


Figure 4.5: Shows the two facts behind the Box-Muller transform: the angle of a pair of independent standard normals is uniform, and its squared radius is exponential. 3000 pairs (Z_1, Z_2) of iid $N(0,1)$ draws (left), with red circles enclosing 50%, 90% and 99% of the points; the squared radius R^2 follows $\text{Exp}(1/2)$ (top right) and the angle Θ is uniform on $(0, 2\pi)$ (bottom right).

4.1.9 The Box–Muller Transformation

Reversing the picture, we **generate** R^2 and Θ and then rotate back to (Z_1, Z_2):

1. $R^2 \sim \text{Exp}(1/2)$ by the inverse CDF: $F(w) = 1 - e^{-w/2}$, so $R^2 = -2\log U_1$, that is, $R = \sqrt{-2\log U_1}$.
2. $\Theta \sim \text{Unif}(0, 2\pi)$: $\Theta = 2\pi U_2$.
3. $Z_1 = R\cos\Theta$ and $Z_2 = R\sin\Theta$.

Theorem (Box–Muller). If $U_1, U_2 \overset{iid}{\sim} \text{Unif}(0,1)$, then

$$Z_1 = \sqrt{-2\log U_1}\,\cos(2\pi U_2), \qquad Z_2 = \sqrt{-2\log U_1}\,\sin(2\pi U_2) \tag{4.8}$$

are independent $N(0,1)$ random variables. Two uniforms produce two normals, and $\mu + \sigma Z \sim N(\mu, \sigma^2)$ then gives any normal distribution.

```r
box_muller <- function(n) {
  u1 <- runif(n); u2 <- runif(n)
  r  <- sqrt(-2 * log(u1))            # R^2 ~ Exp(1/2), i.e. chi-square(2)
  th <- 2 * pi * u2                   # Theta ~ Unif(0, 2 pi)
  cbind(z1 = r * cos(th), z2 = r * sin(th))
}

z <- box_muller(500)
normal_sample <- c(z)                 # 1000 draws: both coordinates of 500 pairs

## R's built-in generator for comparison
nsample2 <- rnorm (1000)

par(mfrow=c(2,2),mar=c(4,4,2,1))
hist(normal_sample,main="Box-Muller Normal")
qqnorm(normal_sample, main="Box-Muller QQ")
qqline(normal_sample)
hist(nsample2, main="Built-in Normal")
qqnorm(nsample2, main="Built-in QQ")
qqline(nsample2)

c(mean = mean(z), sd = sd(z), corr = cor(z)[1, 2])
```

```
       mean          sd        corr
0.003366576  0.998441551 -0.000468918
```

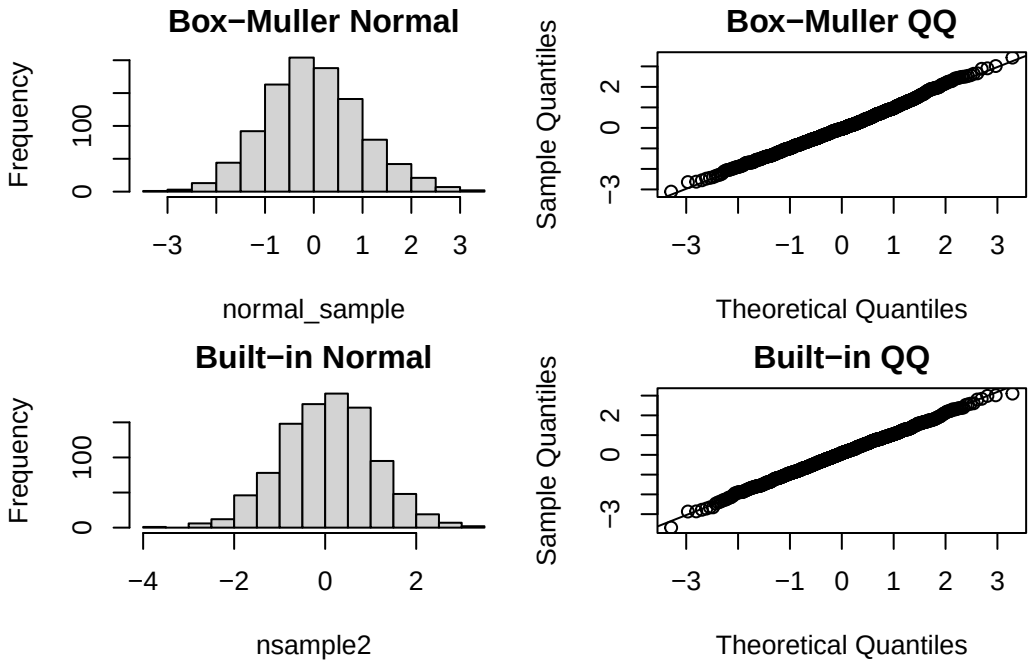


Figure 4.6: Shows how uniform random numbers can be transformed into normal ones, and verifies the Box-Muller construction against R's rnorm(). Normal samples generated by the Box-Muller transform, compared with R's built-in `rnorm()`: histogram and QQ plot for each.

The two coordinates have mean near 0, standard deviation near 1, and are uncorrelated, as the theorem asserts.

4.1.10 Law of Large Numbers and Central Limit Theorem

Monte Carlo estimates an expectation $\mu = E(g(X))$ by an average of simulated values,

$$\hat{\mu}_n = \frac{1}{n}\sum_{i=1}^{n} g(X_i), \qquad X_i \overset{iid}{\sim} f. \tag{4.9}$$

Two theorems justify this estimator and quantify its error.

Theorem (Strong Law of Large Numbers). If $X_1, X_2, \ldots$ are iid with finite mean μ, then

$$P\left(\frac{X_1 + \cdots + X_n}{n} \to \mu\right) = 1. \tag{4.10}$$

The LLN guarantees that the average **converges** to the truth, but it does not say how fast.

Theorem (Central Limit Theorem). If, in addition, the variance σ^2 is finite, then

$$Z = \frac{\bar{X}_n - \mu}{\sigma/\sqrt{n}} \xrightarrow{d} N(0,1) \quad \text{as} \quad n \to \infty. \tag{4.11}$$

The CLT gives the **error bound**: the Monte Carlo error is of order $\sigma/\sqrt{n}$.

4.1.11 Monte Carlo Error Bound from the CLT

By the CLT, with confidence level 95%,

$$P\left(|\bar{X}_n - \mu| \le 1.96\,\frac{\sigma}{\sqrt{n}}\right) \approx 0.95, \tag{4.12}$$

so a 95% interval for μ is

$$\bar{X}_n \pm 1.96\,\frac{s}{\sqrt{n}}, \tag{4.13}$$

where the unknown σ is replaced by the sample standard deviation s of the simulated values.

- Halving the error requires **four times** as many samples, and each extra decimal digit of accuracy costs a **100-fold** increase in n.
- The rate $n^{-1/2}$ does not depend on the dimension of X, which is why Monte Carlo beats numerical quadrature in high dimensions.

4.1.12 Demonstrations

Here we draw from a Gamma distribution with shape parameter $\alpha = 2$ and rate $\beta = 1$. The theoretical mean is $\mu = \alpha/\beta = 2$, and the variance is $\sigma^2 = \alpha/\beta^2 = 2$.

```
n <- 100
rn <- rgamma(n, shape = 2)

par(mfrow = c(2,2))
plot(rn[1:100], main="Raw Gamma Draws", ylab="Value")
hist(rn, main="Histogram of Draws")

## Calculate running averages (LLN)
xbar.rep <- replicate(5, cumsum(rgamma(n, shape = 2))/(1:n))
## Calculate running standardized means (CLT)
se.rep <- replicate(5, (cumsum(rgamma(n, shape = 2))/(1:n) - 2) / sqrt(2/(1:n)))

## Plot LLN: should converge to mu = 2
plot(xbar.rep[,1], main = "Demonstrating LLN", log = "x", type = "b", ylim = c(0, 6),
 ↪  ylab="Sample Mean")
abline(h = 2, lwd=2)
for (i in 2:ncol(xbar.rep)) points(xbar.rep[,i], col = i, pch = i, type = "b")

## Plot CLT: should stabilize within roughly [-3, 3] standard deviations
plot(se.rep[,1], ylim = c(-3,3), type="b", log = "x", main="Demonstrating CLT", ylab="Z-Score")
for (i in 2:ncol(se.rep)) points(se.rep[,i], col = i, pch = i, type = "b")
abline(h = c(0, -2, 2), lty = c(1,2,2), lwd=2)
```

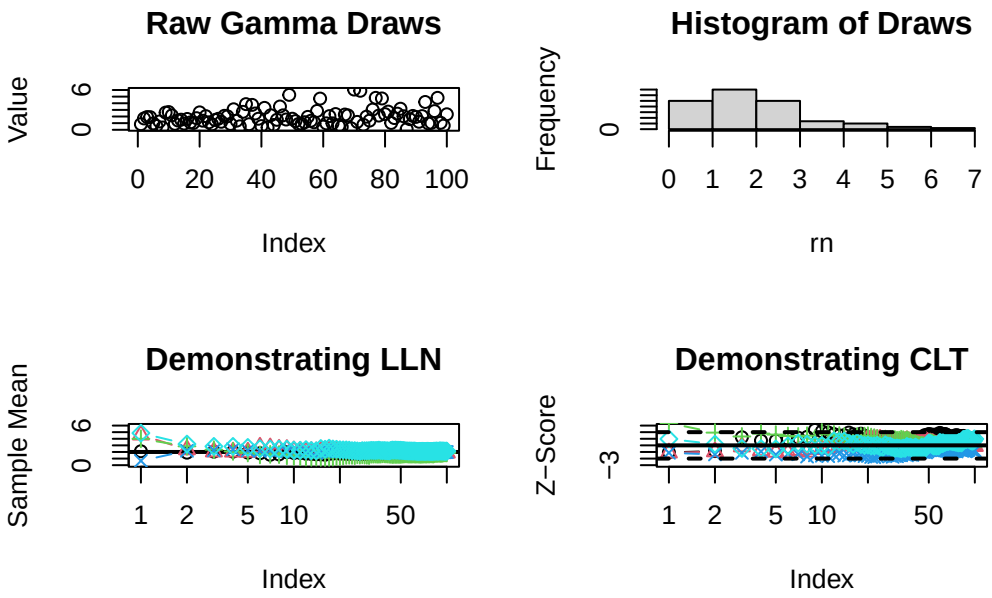


Figure 4.7: Illustrates the two theorems behind Monte Carlo methods: the Law of Large Numbers, under which averages converge to the mean, and the Central Limit Theorem, under which their fluctuations are approximately normal. Law of Large Numbers and Central Limit Theorem for the sample mean of Gamma(2,1) draws: raw draws, their histogram, running averages converging to $\mu = 2$, and standardized running means stabilizing to $N(0, 1)$.

The running standardized means in the last panel only show that Z stays on the $N(0, 1)$ scale. To see its *distribution*, we repeat the experiment $N = 5000$ times: for each repetition we draw n values from the skewed $\text{Exp}(1)$ distribution and compute $(\bar{X}_n - 1)/(1/\sqrt{n})$.

```
clt_demo <- function(n, N = 5000) replicate(N, (mean(rexp(n)) - 1) * sqrt(n))
par(mfrow = c(1, 3), mar = c(4, 4, 2, 1))
for (n in c(2, 10, 100)) {
  hist(clt_demo(n), breaks = 40, freq = FALSE, xlim = c(-4, 4),
       main = TeX(sprintf(r"($n = %d$)", n)), xlab = "")
  curve(dnorm, add = TRUE, col = 2, lwd = 2)
}
```

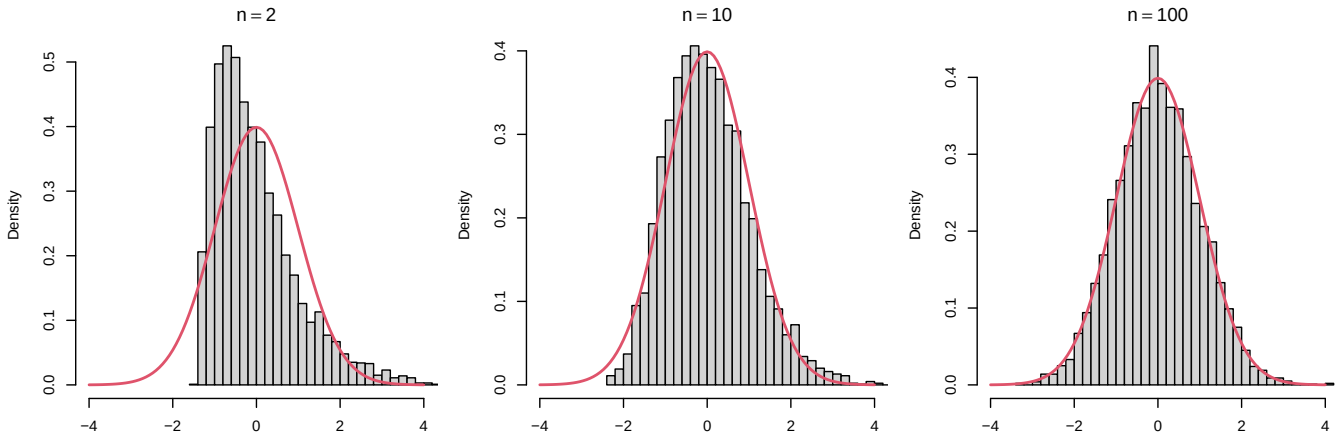


Figure 4.8: Shows the Central Limit Theorem as a statement about distributions: the standardized mean of skewed data becomes approximately normal as the sample size grows. Histograms of 5000 standardized means $(\bar{X}_n - 1)\sqrt{n}$ of Exp(1) samples for $n = 2, 10, 100$, against the $N(0, 1)$ density (red).

The skewness of the exponential is still visible at $n = 2$ and $n = 10$, and has almost disappeared at $n = 100$.

4.1.13 An Example of Monte Carlo for Estimating π

We can estimate the value of π by throwing random darts at a square with side length 2 (from -1 to 1) and counting how many land inside the inscribed circle of radius 1.

The area of the square is $(2)^2 = 4$, and the area of the circle is $\pi(1)^2 = \pi$. Therefore, the probability that a uniform random point (X, Y) falls inside the circle is:

$$P(X^2 + Y^2 \leq 1) = \frac{\text{Area of Circle}}{\text{Area of Square}} = \frac{\pi}{4} \tag{4.14}$$

4.1.13.1 Visualizing the Method

Before we estimate π, let's visually simulate throwing 1,000 darts:

```
set.seed(123)
n_visual <- 1000
x_pts <- runif(n_visual, -1, 1)
y_pts <- runif(n_visual, -1, 1)
inside_circle <- (x_pts^2 + y_pts^2) <= 1

## Plot points
plot(x_pts, y_pts, col = ifelse(inside_circle, "steelblue", "tomato"),
     pch = 20, asp = 1, xlab = "X", ylab = "Y",
     main = paste(sum(inside_circle), "out of", n_visual, "points in circle"))

## Draw the boundaries
rect(-1, -1, 1, 1, border = "black", lwd = 2)
```

```r
theta_seq <- seq(0, 2*pi, length.out = 100)
lines(cos(theta_seq), sin(theta_seq), col = "black", lwd = 2)
```

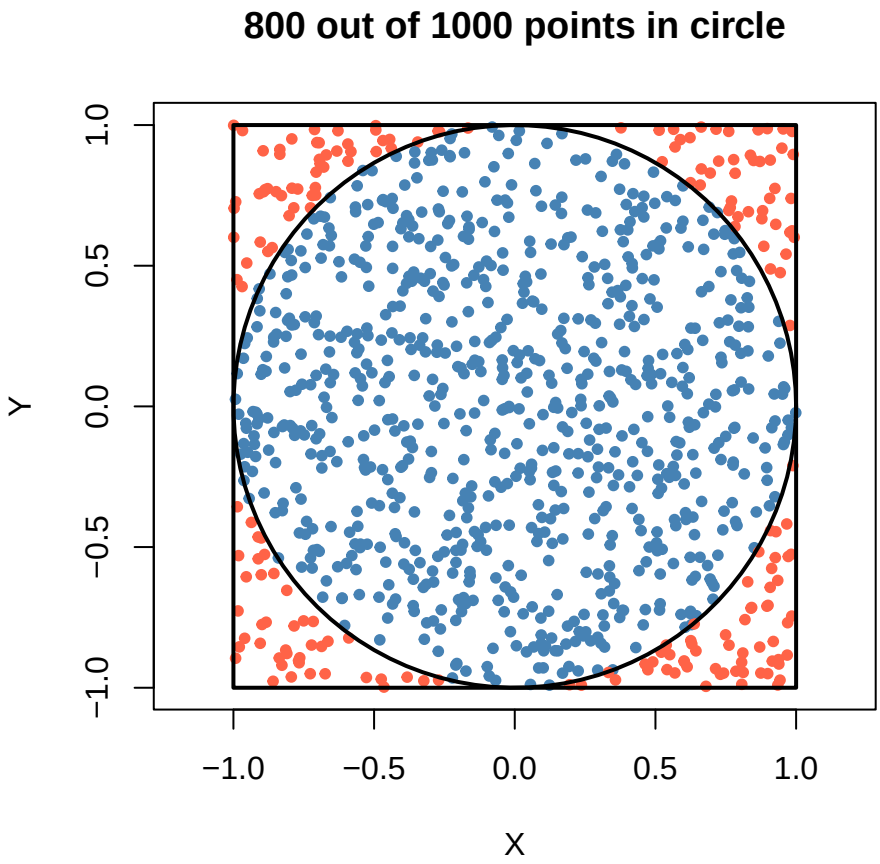


Figure 4.9: Illustrates Monte Carlo integration with a geometric example, estimating π as the proportion of random points that fall inside a circle. 1,000 uniform points ('darts') thrown at the $[-1, 1]^2$ square; the fraction landing inside the inscribed circle gives a Monte Carlo estimate of π.

4.1.13.2 The Monte Carlo Estimate

If we define an indicator variable I_i that equals 1 if the i-th dart lands in the circle and 0 otherwise, then $E[I_i] = \pi/4$. We define $Z_i = 4 \times I_i$, which gives $E[Z_i] = \pi$ and

$$\text{Var}(Z_i) = 16 \cdot \frac{\pi}{4}\left(1 - \frac{\pi}{4}\right) \approx 2.70, \qquad \text{SD}(Z_i) \approx 1.64. \tag{4.15}$$

Our Monte Carlo estimate $\hat{\pi}$ is the sample average of the Z_i values:

$$\hat{\pi} = \bar{Z} = \frac{1}{n}\sum_{i=1}^{n} Z_i = 4 \times \frac{\#\{i : X_i^2 + Y_i^2 \leq 1\}}{n} \tag{4.16}$$

The Standard Error (SE) measures the uncertainty of our estimate. Using the sample standard deviation S_Z, the 95% margin of error is calculated as:

$$\text{Margin of Error} = 1.96 \times \text{SE} = 1.96 \times \frac{S_Z}{\sqrt{n}} \tag{4.17}$$

The standard error is computed from the **same** simulated values as the estimate, at no extra cost, and should always be reported with a Monte Carlo estimate.

4.1.13.3 Results by Sample Size

We encapsulate the logic in a function and track how precision improves as n increases. The table below outlines our findings, including the actual error $\hat{\pi} - \pi$ and the theoretical margin of error $1.96 \times 1.64/\sqrt{n}$.

```
library(knitr)

## n is the number of samples drawn uniformly from the rectangle (-1,1)  * (-1,1)
## an estimate of pi is returned
pi_est_mc <- function(n)
{
    X <- runif(n,-1,1)
    Y <- runif(n,-1,1)

    # 4 times the indicator function
    Z <- 4 * (X^2 + Y^2 <= 1)

    mu <- mean(Z)
    error <- 1.96 * sd(Z) / sqrt(n)

    list(n = n, pi.est = mu, error.95perc = error, ci.lower = mu - error, ci.upper = mu +
     ↪  error)
}

## Run the estimation for varying sample sizes
sample_sizes <- c(100, 10000, 100000, 10000000)
results_list <- lapply(sample_sizes, pi_est_mc)

## Combine results into a dataframe for tabular display
results_df <- data.frame(
  N = sapply(results_list, function(x) x$n),
  Estimate = sapply(results_list, function(x) x$pi.est),
  Actual_Error = sapply(results_list, function(x) x$pi.est - pi),
  Error_Margin = sapply(results_list, function(x) x$error.95perc),
  Theory_Margin = 1.96 * 1.64 / sqrt(sample_sizes),
  CI_Lower = sapply(results_list, function(x) x$ci.lower),
  CI_Upper = sapply(results_list, function(x) x$ci.upper)
)

## Render the formatted table
```

```
kable(results_df,
      col.names = c("Sample Size (N)", "Estimated Pi", "Actual Error", "95% Margin of Error",
                    "Theoretical Margin", "Lower CI", "Upper CI"),
      digits = 6,
      caption = "Shows how the accuracy of the Monte Carlo estimate of $\\pi$, as measured by
       ↪  its margin of error, improves with the number of points. Monte Carlo Estimation of Pi
       ↪  across different sample sizes")
```

Table 4.1: Shows how the accuracy of the Monte Carlo estimate of π, as measured by its margin of error, improves with the number of points. Monte Carlo estimation of pi across different sample sizes.

Table 4.1: Shows how the accuracy of the Monte Carlo estimate of π, as measured by its margin of error, improves with the number of points. Monte Carlo Estimation of Pi across different sample sizes

Sample Size (N)	Estimated Pi	Actual Error	95% Margin of Error	Theoretical Margin	Lower CI	Upper CI
1e+02	2.920000	-0.221593	0.349818	0.321440	2.570182	3.269818
1e+04	3.156400	0.014807	0.031985	0.032144	3.124415	3.188385
1e+05	3.143600	0.002007	0.010170	0.010165	3.133430	3.153770
1e+07	3.141769	0.000176	0.001018	0.001016	3.140751	3.142787

- The actual error is typically within the 95% margin of error, that is, within about 2 standard errors.
- The margin of error shrinks by a factor of $\sqrt{k}$ when n grows k-fold (a factor of 10 for each 100-fold increase), as the CLT predicts, and agrees with its theoretical value.

4.1.13.4 Convergence of $\hat{\pi}_n$

Following a single sequence of darts shows the same $n^{-1/2}$ behaviour as it unfolds.

```
set.seed(1)
n <- 1e5; z <- 4 * (runif(n, -1, 1)^2 + runif(n, -1, 1)^2 <= 1)
k <- 1:n; pihat <- cumsum(z) / k
par(mar = c(4, 4.5, 0.5, 0.5), mgp = c(2.5, 0.8, 0))
plot(k, pihat, type = "l", log = "x", ylim = c(2, 4),
     xlab = "n (log scale)", ylab = expression(hat(pi)[n]))
lines(k, pi + 1.96 * 1.64 / sqrt(k), lty = 2, col = 2)
lines(k, pi - 1.96 * 1.64 / sqrt(k), lty = 2, col = 2)
abline(h = pi, col = 4)
```

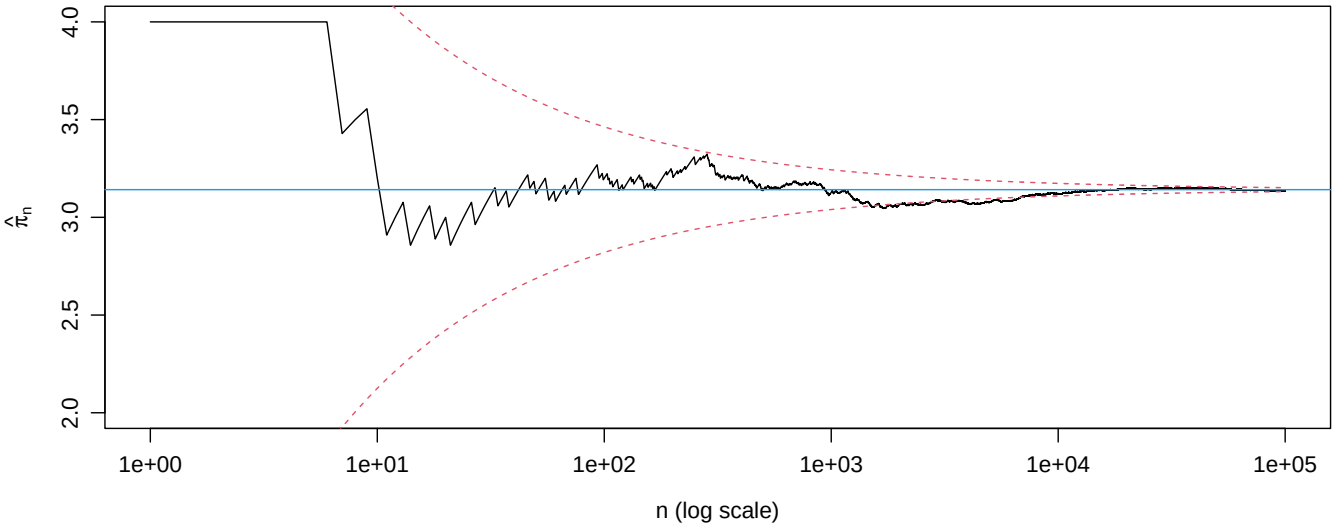


Figure 4.10: Shows the Monte Carlo error bound at work along a single simulation run: the running estimate of π stays within a band that narrows at rate $1/\sqrt{n}$. Running estimate $\hat{\pi}_n$ over 10^5 darts (black), with the true value π (blue) and the band $\pi \pm 1.96 \times 1.64/\sqrt{n}$ (red, dashed).

The running estimate stays inside the red band most of the time.

4.2 Point Estimation

Assume data $X = (X_1, \dots, X_n)$ is drawn from a distribution $f(x;\theta)$ indexed by an unknown parameter $\theta \in \Theta$. An estimator $\hat{\theta} = T(X)$ is a function derived exclusively from the data. Because it relies on random samples, $\hat{\theta}$ is itself a random variable. Its behavior is defined by its sampling distribution.

The two fundamental summaries of an estimator's performance are bias and variance:

$$\text{bias}(\hat{\theta}) = E_\theta(\hat{\theta}) - \theta, \qquad \text{Var}_\theta(\hat{\theta}) = E_\theta\left[\left(\hat{\theta} - E_\theta(\hat{\theta})\right)^2\right]. \tag{4.18}$$

The most comprehensive single-number summary is the Mean Squared Error (MSE), defined as $\text{MSE}_\theta(\hat{\theta}) = E_\theta[(\hat{\theta} - \theta)^2]$. By adding and subtracting $E_\theta(\hat{\theta})$ inside the square, the cross term vanishes, yielding the MSE decomposition:

$$\text{MSE}_\theta(\hat{\theta}) = \text{Var}_\theta(\hat{\theta}) + \left[\text{bias}_\theta(\hat{\theta})\right]^2. \tag{4.19}$$

Key Insight: Unbiasedness is not strictly necessary. An estimator that introduces a small bias in exchange for a massive reduction in variance will frequently achieve a lower MSE than a strictly unbiased competitor.

Because MSE is a *function* of θ, it acts as the risk function under squared-error loss. Two estimators rarely have one absolute winner; their risk curves often cross. To choose a single estimator globally, statisticians summarize the risk curve via two main criteria:

- **Minimax:** Minimize the worst-case scenario: $\sup_{\theta \in \Theta} \mathrm{MSE}_\theta(\hat{\theta})$
- **Bayes Risk:** Average the error against a prior distribution π: $\int_\Theta \mathrm{MSE}_\theta(\hat{\theta})\, \pi(\theta)\, d\theta$

Asymptotically, $\hat{\theta}_n$ is consistent if it converges to θ in probability. Under regularity conditions, Maximum Likelihood Estimators (MLEs) are consistent and asymptotically normal, achieving the Cramér–Rao lower variance bound $1/I(\theta)$. However, in small samples, the MLE can often be outperformed.

4.2.1 Monte Carlo Evaluation of an Estimator

Risk functions are rarely available in closed form, but because they are expectations, we can estimate them efficiently using Monte Carlo simulation.

The Monte Carlo Recipe:

1. Define a grid of θ values.
2. For each θ, generate N independent datasets $X^{(j)} = (X_1^{(j)}, \dots, X_n^{(j)})$, $j = 1, \dots, N$.
3. Compute the estimate $\hat{\theta}^{(j)} = \hat{\theta}(X^{(j)})$ for each dataset.
4. Average the squared errors:

$$\widehat{\mathrm{MSE}}_\theta = \frac{1}{N} \sum_{j=1}^{N} \left(\hat{\theta}^{(j)} - \theta \right)^2 \tag{4.20}$$

The bias and variance are estimated from the same replicates:

$$\widehat{\mathrm{bias}}_\theta = \bar{\hat{\theta}} - \theta, \qquad \widehat{\mathrm{Var}}_\theta = \frac{1}{N} \sum_{j=1}^{N} \left(\hat{\theta}^{(j)} - \bar{\hat{\theta}} \right)^2, \qquad \bar{\hat{\theta}} = \frac{1}{N} \sum_{j=1}^{N} \hat{\theta}^{(j)}. \tag{4.21}$$

Two sample sizes appear: n, the number of observations in each dataset, which is fixed by the problem; and N, the number of replicates, which we choose large enough that the Monte Carlo error is negligible.

Because $\widehat{\mathrm{MSE}}_\theta$ is an average of independent, identically distributed (i.i.d.) quantities, its standard error is $s/\sqrt{N}$, where s^2 is the sample variance of the squared errors. Always report the Monte Carlo error boundaries ($\widehat{\mathrm{MSE}}_\theta \pm 1.96\, s/\sqrt{N}$) to distinguish true differences in risk curves from mere simulation noise.

4.2.2 Example 1: Two Estimators of Proportions

Consider estimating a binomial proportion p from n independent trials.

- **Maximum Likelihood Estimator ($\hat{p}_1$):** The sample mean, $\hat{p}_1 = \bar{X}$. This is unbiased but exhibits high variance in small samples, especially near boundaries ($p \approx 0$ or 1).
- **Smoothed Estimator ($\hat{p}_2$):** Often called the Laplace or "add-one-in" estimator, $\hat{p}_2 = (\sum_i X_i + 1)/(n + 2)$. This introduces bias by shrinking the estimate toward $1/2$ (the posterior mean under a uniform prior), but reduces variance by a factor of $[n/(n+2)]^2$.

We can estimate the risk curves on a grid of p values using the Monte Carlo method. Both estimators rely strictly on the sufficient statistic $S = \sum_i X_i \sim \text{Binomial}(n, p)$, allowing us to vectorize the simulation without looping over replicates.

```
###Estimators, written in terms of the sufficient statistic S = sum(X_i)
p_est1 <- function(s, n) s / n
p_est2 <- function(s, n) (s + 1) / (n + 2)

### Monte Carlo estimate of the MSE and its standard error.  All no_sim
### replicates are generated in a single vectorized draw.
mse_est_mc <- function(n, p, no_sim, p_est)
{
    sq_error <- (p_est(rbinom(no_sim, size = n, prob = p), n) - p)^2
    c(mse = mean(sq_error), sd = sqrt(var(sq_error) / no_sim))
}

### MSE over a grid of p values, returned as a 2 by length(p_set) matrix
mse_curve <- function(n, p_set, no_sim, p_est)
    vapply(p_set, function(p) mse_est_mc(n, p, no_sim, p_est), c(mse = 0, sd = 0))
```

```
no_sim <- 10000
p_set <- seq(0, 1, by = 0.05)
n_set <- c(2, 10, 50, 100)

par(mfrow = c(2, 2), mar = c(4, 4.5, 2.5, 1))

for (n in n_set)
{
    r1 <- mse_curve(n, p_set, no_sim, p_est1)
    r2 <- mse_curve(n, p_set, no_sim, p_est2)

    bands <- cbind(r1["mse", ], r1["mse", ] + 1.96 * r1["sd", ], r1["mse", ] - 1.96 * r1["sd",
↪ ],
                   r2["mse", ], r2["mse", ] + 1.96 * r2["sd", ], r2["mse", ] - 1.96 * r2["sd",
↪ ])

    matplot(p_set, bands, type = "l", lty = c(1, 2, 2, 1, 2, 2), col = rep(c(1, 2), each = 3),
            xlab = TeX(r"($p$)"),
            ylab = TeX(r"($E[(\hat{p} - p)^2]$)"),
            main = TeX(sprintf(r"($n = %d$)", n)))

    legend("top", bty = "n", lty = 1, col = c(1, 2), cex = 0.9,
           legend = TeX(c(r"($\hat{p}_1 = S/n$)", r"($\hat{p}_2 = (S+1)/(n+2)$)")))
}
```

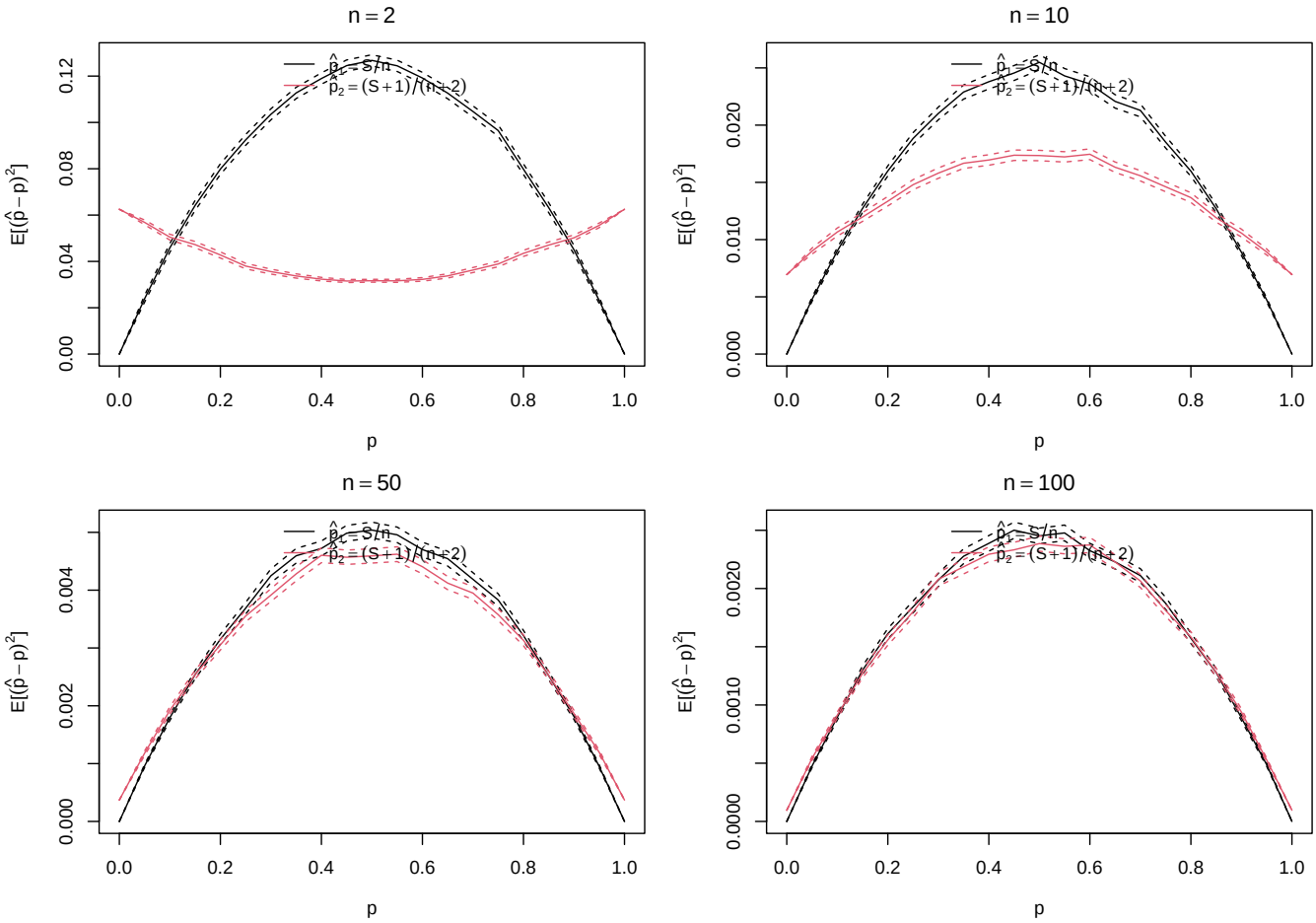


Figure 4.11: Uses simulation to compare two estimators of a binomial proportion by their mean squared error, showing when a small bias pays off through lower variance. Simulated MSE (with 95% Monte Carlo error bands) of $\hat{p}_1 = S/n$ versus the smoothed $\hat{p}_2 = (S+1)/(n+2)$, across $n = 2, 10, 50, 100$.

When n is small, the smoothed estimator significantly outperforms the MLE across most of the parameter space. As n grows, the curves converge, reflecting the asymptotic efficiency of the MLE. The exception occurs strictly at the extremes ($p \approx 0$ or 1), where the shrinkage penalty overrides the variance reduction, making $\hat{p}_1$ superior. The confidence bands confirm this crossing is a true mathematical property, not simulation variance.

4.2.3 Example 2: Mean, Median and Trimmed Mean

The choice between estimators can also depend on the shape of the data distribution rather than on the parameter value. Consider estimating the centre $\theta = 0$ of a symmetric distribution from $n = 20$ observations with three estimators: the sample mean, the sample median, and the 10% trimmed mean, which averages the observations after discarding the smallest and largest 10%. We estimate their bias, variance and MSE from $N = 5000$ replicates, for normal data and for heavy-tailed t_3 data.

```r
eval_est <- function(rdist, n = 20, N = 5000) {
  est <- replicate(N, { x <- rdist(n); c(mean = mean(x), median = median(x), trim10 = mean(x,
 ↪  trim = 0.1)) })
  rbind(bias = rowMeans(est), var = apply(est, 1, var), mse = rowMeans(est^2))
}
kable(rbind(eval_est(rnorm),                          # normal data
            eval_est(function(n) rt(n, df = 3))),     # heavy-tailed t3 data
      digits = 4,
      col.names = c("mean", "median", "10% trimmed mean"),
      caption = "Monte Carlo bias, variance and MSE: normal data (rows 1-3) and $t_3$ data
       ↪  (rows 4-6).")
```

Table 4.2: Compares three estimators of the centre of a symmetric distribution under normal and heavy-tailed data, to show that the best estimator depends on the data distribution. Monte Carlo bias, variance and MSE of the mean, the median and the 10% trimmed mean, for $n = 20$ observations from $N(0, 1)$ (top) and t_3 (bottom), with $N = 5000$ replicates.

Table 4.2: Monte Carlo bias, variance and MSE: normal data (rows 1-3) and t_3 data (rows 4-6).

	mean	median	10% trimmed mean
bias	-0.0033	-0.0028	-0.0026
var	0.0491	0.0730	0.0520
mse	0.0491	0.0730	0.0520
bias	-0.0039	-0.0025	-0.0031
var	0.1461	0.0871	0.0812
mse	0.1461	0.0871	0.0812

All three estimators are unbiased by symmetry, so the comparison is decided by the variance. No estimator is best everywhere: the mean wins under normality, where it is the MLE, while the trimmed mean and the median win under heavy tails, where a few extreme observations inflate the variance of the mean. The trimmed mean is a good compromise, losing little under normality.

4.3 Hypothesis Testing

A hypothesis test maps data to a binary decision about a null hypothesis H_0 versus an alternative H_1. It is built from a test statistic T, whose large values are evidence against H_0. This section shows that both the error rates of a test and, when the null distribution is unknown, the p-value itself are expectations, and so can be estimated by Monte Carlo.

4.3.1 The Two Types of Error

A test rejects H_0 when the statistic T exceeds a critical value c_α. Crossing the decision with the truth gives four outcomes:

	reject: $T > c_\alpha$	do not reject: $T \le c_\alpha$
H_0 true	Type I error (false positive)	true negative
H_1 true	true positive	Type II error (false negative)

The two error rates of interest are the **size** and the **power**:

$$\text{size} = P(T > c_\alpha \mid H_0) = \alpha, \qquad \text{power} = P(T > c_\alpha \mid H_1) = 1 - \beta. \tag{4.22}$$

When H_0 is composite, the size is the largest type I error probability over all parameter values in H_0, and a test has *level* α if its size is at most α. Both size and power are probabilities, hence expectations of indicators, so both can be **estimated by Monte Carlo** once we can simulate data under H_0 and under H_1.

4.3.2 Estimating False and True Positive Rates by Monte Carlo

For a test that rejects when $T > c$, simulate N data sets under H_0 and N data sets under H_1, and compute T on each:

replicate	$T^{(j)}_{H_0}$	$I(T^{(j)}_{H_0} > c)$	$T^{(j)}_{H_1}$	$I(T^{(j)}_{H_1} > c)$
1	$T^{(1)}_{H_0}$	0	$T^{(1)}_{H_1}$	1
2	$T^{(2)}_{H_0}$	1	$T^{(2)}_{H_1}$	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
N	$T^{(N)}_{H_0}$	0	$T^{(N)}_{H_1}$	0

The false and true positive rates are estimated by the averages of the two indicator columns:

$$\widehat{\text{FPR}}(c) = \frac{1}{N}\sum_{j=1}^{N} I(T^{(j)}_{H_0} > c), \qquad \widehat{\text{TPR}}(c) = \frac{1}{N}\sum_{j=1}^{N} I(T^{(j)}_{H_1} > c). \tag{4.23}$$

$\widehat{\text{FPR}}$ estimates the size (the type I error rate) and $\widehat{\text{TPR}}$ estimates the power, $1-\beta$. To obtain a test of size α when the null distribution of T is unknown, take $\widehat{c}_\alpha$ to be the $(1-\alpha)$ sample quantile of $T^{(1)}_{H_0}, \ldots, T^{(N)}_{H_0}$; this replaces the null distribution of T by its **empirical** distribution.

4.3.3 The p-value: A Probability Integral Transform of T under H_0

The **p-value** repackages the test so that no α has to be fixed in advance. It is the probability, under H_0, of a statistic at least as extreme as the observed one:

$$\text{p-value}(t) = P(T \ge t \mid H_0) = 1 - F_{H_0}(t), \tag{4.24}$$

that is, the **survival function** of T under H_0 evaluated at t^{obs}. Rejecting when p-value$(t^{obs}) < \alpha$ is the same as rejecting when $t^{obs} > c_\alpha$.

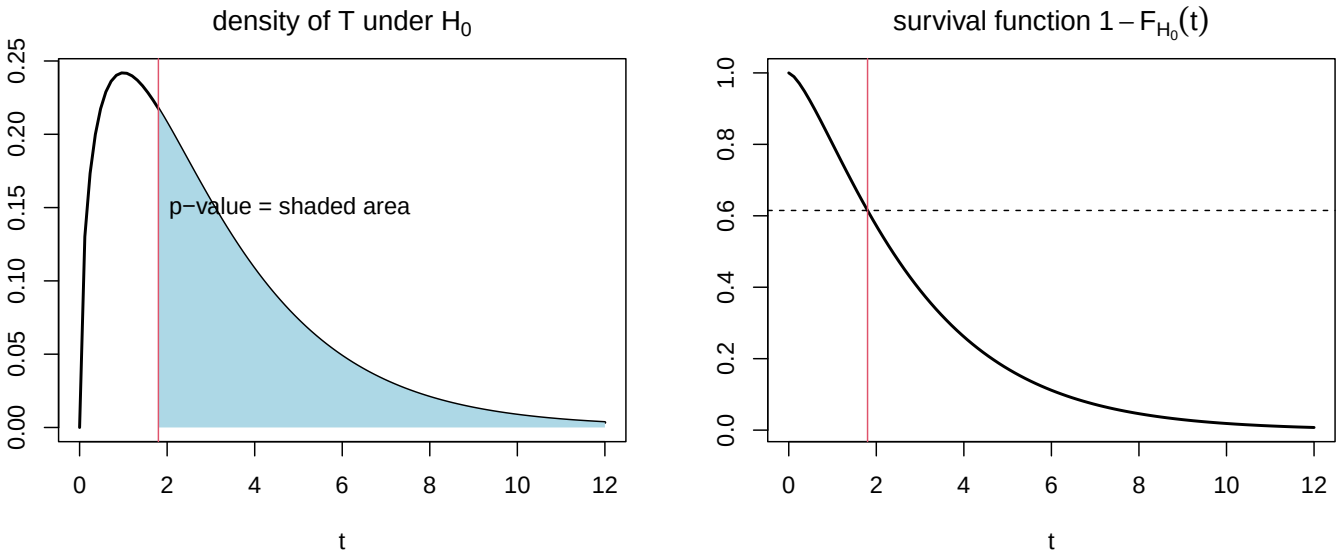


Figure 4.12: Shows that the p-value is a tail area of the null distribution of the test statistic, and equivalently the null survival function evaluated at the observed statistic. Left: the null density of T (here χ^2_3) with the p-value of $t^{obs} = 1.8$ shaded. Right: the survival function $1 - F_{H_0}(t)$, whose height at t^{obs} is the same p-value.

4.3.4 Uniformity of the p-value under H_0

Theorem. If $T \mid H_0$ has a continuous CDF F_{H_0}, then p-value$(T) = 1 - F_{H_0}(T) \sim$ Unif$(0, 1)$ under H_0.

Proof. For $t \in (0, 1)$,

$$\begin{aligned} P(\text{p-value}(T) < t) &= P(1 - F_{H_0}(T) < t) = P(F_{H_0}(T) > 1 - t) \\ &= P(T > F^{-1}_{H_0}(1 - t)) = 1 - F_{H_0}(F^{-1}_{H_0}(1 - t)) = t. \end{aligned} \tag{4.25}$$

Two consequences follow.

- $P(\text{p-value} < \alpha \mid H_0) = \alpha$ **exactly**, so rejecting when the p-value is below α gives a test of size α.
- A histogram of p-values simulated under H_0 should be flat. This is the standard **test of a test**: a histogram with excess mass near 0 means the test is liberal (its true size exceeds α), and one with excess mass near 1 means it is conservative (it wastes power).

4.3.5 Monte Carlo Estimates of Size and Power

Simulating N data sets under H_0 and N under H_1 and computing the p-value of each, the size and power at level α are estimated by the rejection rates

$$\widehat{\text{size}} = \frac{1}{N}\sum_{j=1}^{N} I(\text{p-value}_{H_0}^{(j)} < \alpha), \qquad \widehat{\text{power}} = \frac{1}{N}\sum_{j=1}^{N} I(\text{p-value}_{H_1}^{(j)} < \alpha). \tag{4.26}$$

Each estimate is a proportion, so its Monte Carlo standard error is $\sqrt{\hat{p}(1-\hat{p})/N}$, about 0.003 for a size near 0.05 with $N = 5000$.

Throughout this section, simulated p-values are displayed with the helper below. Its bins have width α, so the first bar (in red) holds exactly the rejected data sets, and the legend reports the rejection rate: the estimated size under H_0, or the estimated power under H_1.

4.3.6 Example 1: The t-test under Different Data Settings

The one-sample t-test of $H_0 : \mu = 0$ assumes normally distributed observations. We first check it where that assumption holds, and then under heavy-tailed data, where it does not.

4.3.6.1 Normal Data

With $n = 20$ normal observations and $N = 5000$ simulated data sets, we compute the t-test p-value under $H_0 : \mu = 0$ and under $H_1 : \mu = 0.5$.

```
pv_sim <- function(mu, n = 20, N = 5000) replicate(N, t.test(rnorm(n, mu))$p.value)
pv0 <- pv_sim(0); pv1 <- pv_sim(0.5)

par(mfrow = c(1, 2), mar = c(4, 4, 2.5, 1))
hist_reject(pv0, TeX(r"($H_0: \mu = 0$)"))
hist_reject(pv1, TeX(r"($H_1: \mu = 0.5$)"))

c(size = mean(pv0 < 0.05), power = mean(pv1 < 0.05),
  exact_power = power.t.test(n = 20, delta = 0.5, sd = 1, type = "one.sample")$power)
```

```
      size      power exact_power
 0.0516000  0.5552000  0.5644829
```

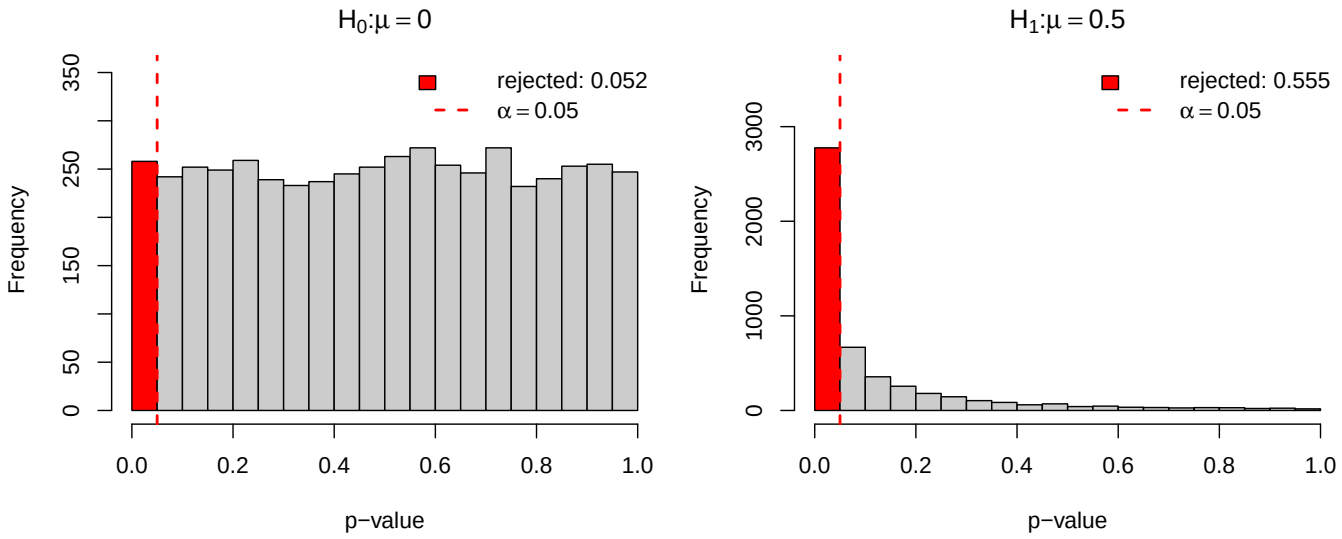


Figure 4.13: Illustrates the basic simulation check of a test: p-values are uniform under the null hypothesis and concentrate near 0 under the alternative. p-values of the one-sample t-test for $n = 20$ normal observations under $H_0 : \mu = 0$ (left) and $H_1 : \mu = 0.5$ (right); red bars are the rejections at $\alpha = 0.05$.

Under H_0 the p-values are uniform; under H_1 they pile up near 0. The simulated power agrees with the exact power from `power.t.test()` to within Monte Carlo error.

4.3.6.2 Heavy-tailed Data: Null Distribution Diagnostics

We next simulate data from a t-distribution with ν degrees of freedom. Smaller ν gives heavier tails, and $\nu = 1$ is the Cauchy distribution, which has no mean. The function below simulates `N` data sets of size `n` at once and returns their t-test p-values.

```
t.test.sim <- function (N, n, df, mu.true = 0, mu = 0)
{
    X <- matrix(rt(n * N, df), n, N) + mu.true
    m <- colMeans(X)
    v <- colSums((X - rep(m, each = n))^2) / (n - 1)
    2 * pt(-abs((m - mu) / sqrt(v / n)), n - 1)
}

t_main <- function (n, df, mu.true = 0)
    TeX(sprintf(r"($n = %d$, $\nu = %d$, $\mu = %g$)", n, df, mu.true))

N <- 2000
```

100 degrees of freedom (nearly normal). The data are practically normal, and the p-values are uniform for both $n = 2$ and $n = 20$.

```
par(mfrow = c(1, 2), mar = c(4, 4, 2.5, 1))
hist_reject(t.test.sim(N, 2, 100), t_main(2, 100))
hist_reject(t.test.sim(N, 20, 100), t_main(20, 100))
```

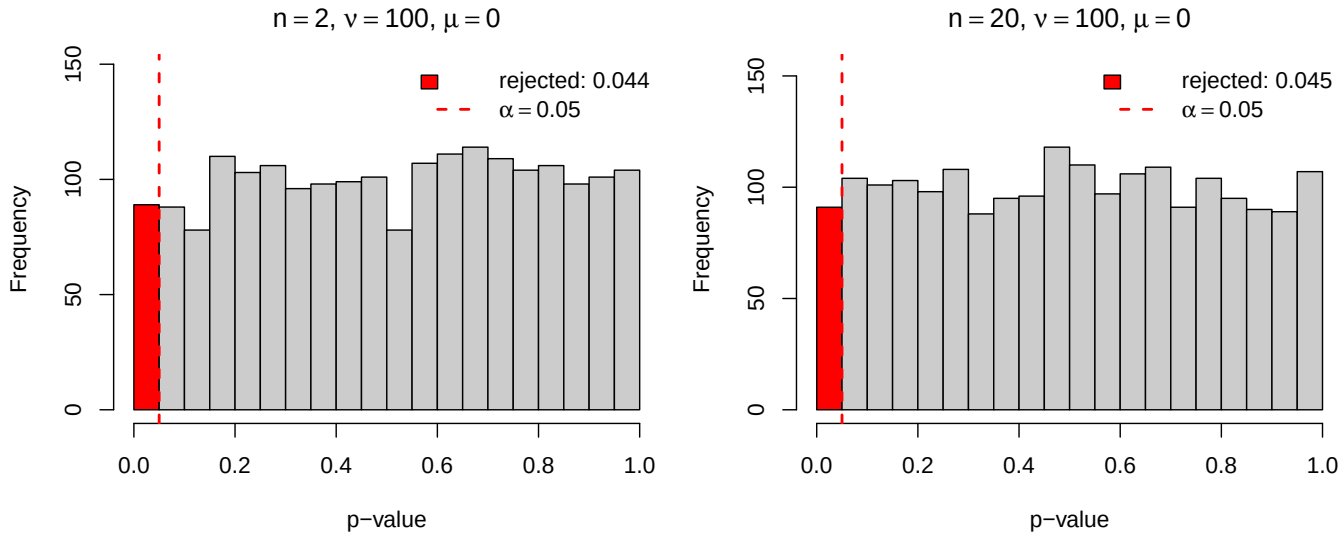


Figure 4.14: Uses simulation to check the size of the t-test: under the null hypothesis a valid test gives uniform p-values, and here the data are nearly normal. Null distribution of t-test p-values at $\nu = 100$ (nearly normal), for $n = 2$ and $n = 20$: uniform, as expected.

3 degrees of freedom (heavy tails). Heavier tails distort the p-value distribution, and the nominal type I error rate is no longer exactly calibrated.

```
par(mfrow = c(1, 2), mar = c(4, 4, 2.5, 1))
hist_reject(t.test.sim(N, 2, 3), t_main(2, 3))
hist_reject(t.test.sim(N, 20, 3), t_main(20, 3))
```

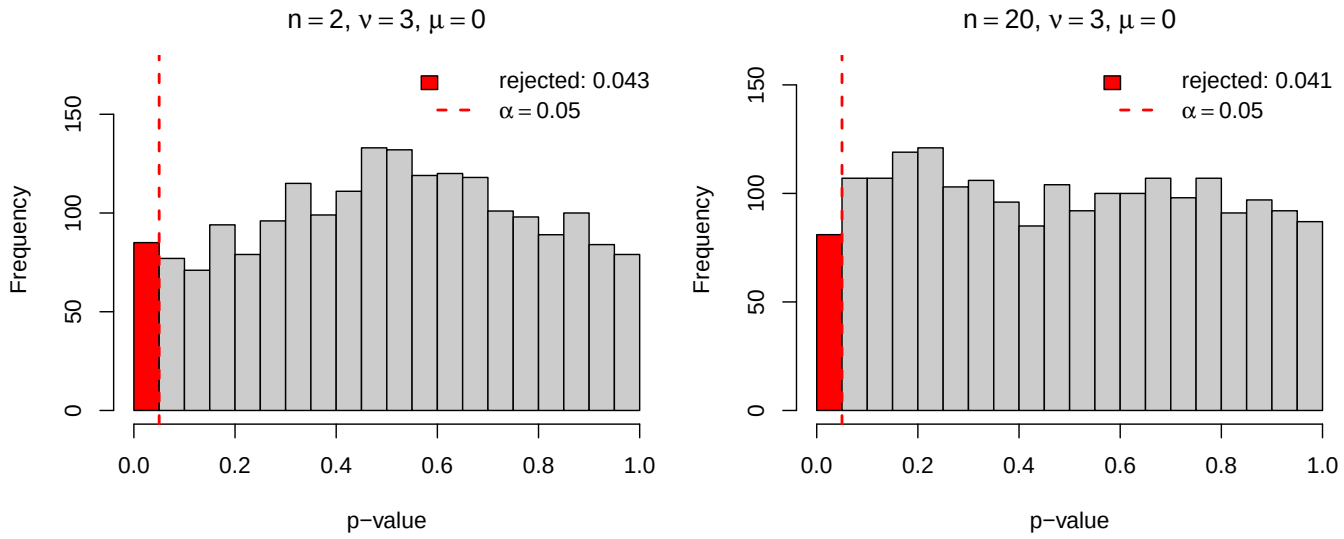


Figure 4.15: Repeats the size check of the t-test for heavy-tailed data to show how violating the normality assumption distorts the p-values. Null distribution of t-test p-values at $\nu = 3$ (heavy tails), for $n = 2$ and $n = 20$: the tails pull the p-value distribution away from uniform.

1 degree of freedom (Cauchy). With no finite mean or variance, a single extreme observation dominates both $\bar{X}$ and s, which pushes $|T|$ toward 1 and the p-value toward the middle of $[0, 1]$. The p-values are far from uniform and the rejection rate falls below α: the test is conservative, and it does not improve as n grows.

```
par(mfrow = c(1, 2), mar = c(4, 4, 2.5, 1))
hist_reject(t.test.sim(N, 2, 1), t_main(2, 1))
hist_reject(t.test.sim(N, 20, 1), t_main(20, 1))
```

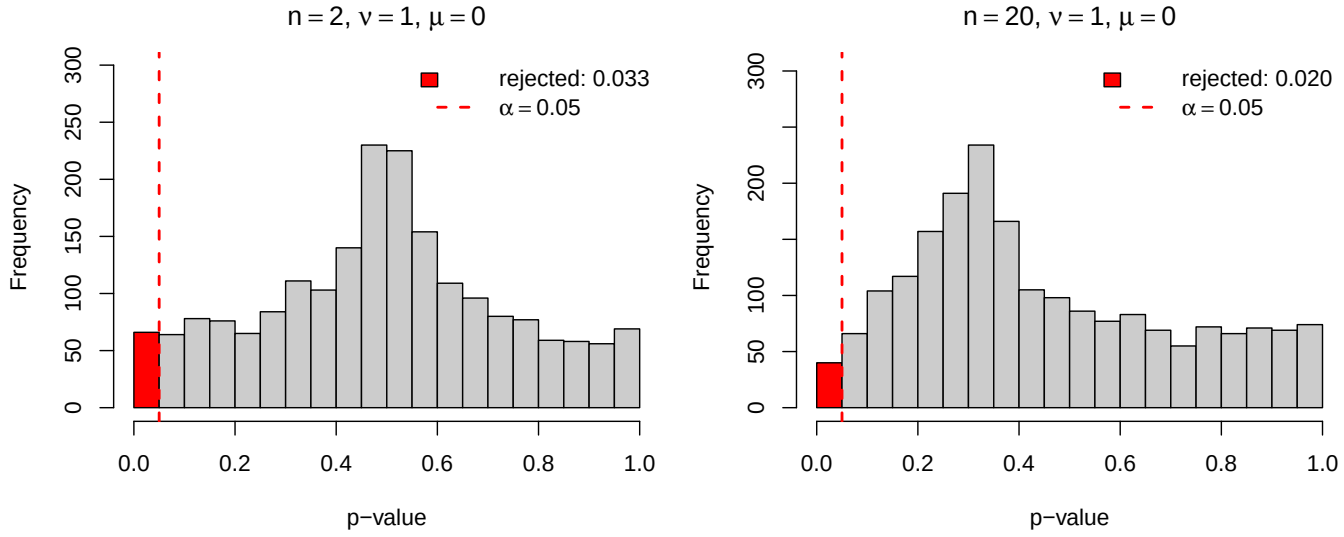


Figure 4.16: Pushes the size check to the extreme case of Cauchy data, which have no mean, to show where the t-test breaks down completely. Null distribution of t-test p-values at $\nu = 1$ (Cauchy), for $n = 2$ and $n = 20$: the p-values bunch in the middle of $[0, 1]$ and the rejection rate is below 0.05.

Larger sample size ($n = 30$). When the variance is finite, the Central Limit Theorem makes $\bar{X}$ nearly normal as n grows, and the p-values for $\nu = 10$ and $\nu = 5$ are close to uniform again.

```
par(mfrow = c(1, 2), mar = c(4, 4, 2.5, 1))
hist_reject(t.test.sim(N, 30, 10), t_main(30, 10))
hist_reject(t.test.sim(N, 30, 5), t_main(30, 5))
```

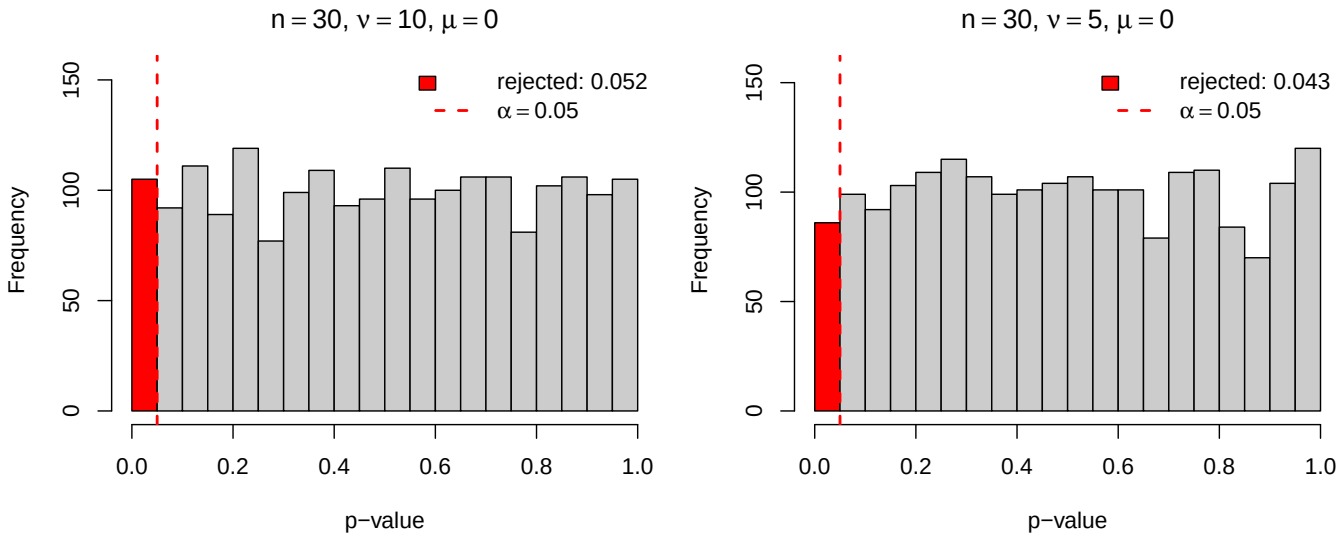


Figure 4.17: Shows that a larger sample size can rescue the t-test for heavy-tailed data, because the Central Limit Theorem makes the sample mean nearly normal. Null distribution of t-test p-values at $n = 30$ for $\nu = 10$ and $\nu = 5$: close to uniform even with heavy-tailed data.

4.3.6.3 Heavy-tailed Data: Power

To estimate power, we shift the data to a true mean $\mu = 0.5$ while still testing $H_0 : \mu = 0$. The rejection rate is now the power.

```
par(mfrow = c(1, 2), mar = c(4, 4, 2.5, 1))
hist_reject(t.test.sim(N, 30, 5, mu.true = 0.5), t_main(30, 5, 0.5))
hist_reject(t.test.sim(N, 60, 5, mu.true = 0.5), t_main(60, 5, 0.5))
```

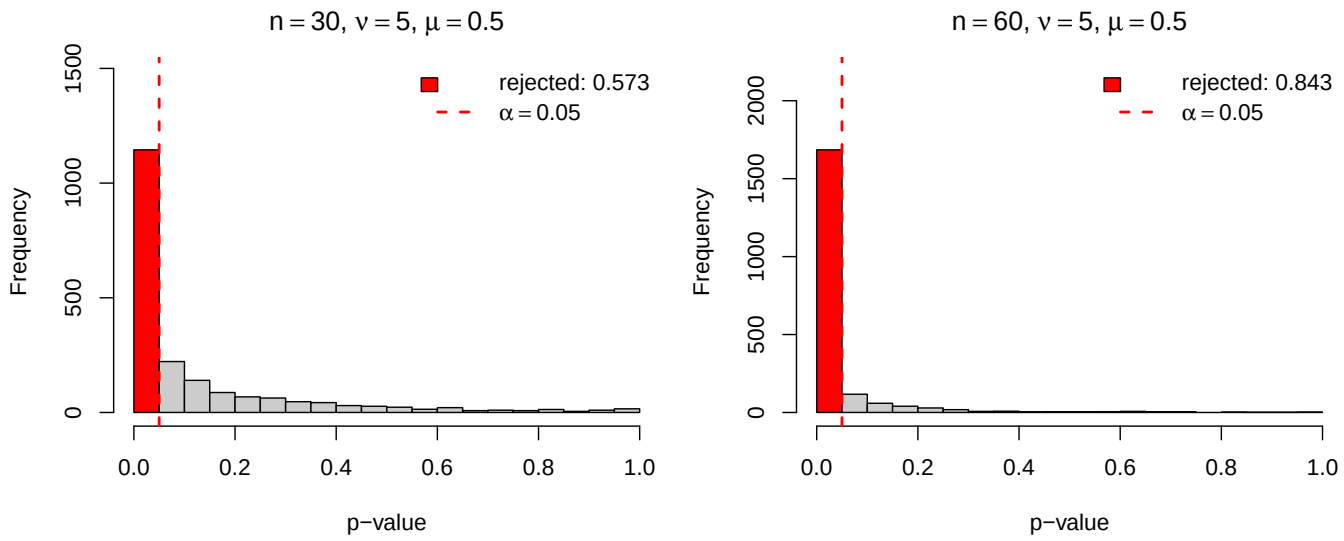


Figure 4.18: Uses simulation to estimate the power of the t-test, its ability to detect a true mean of 0.5, when the data are moderately heavy-tailed. t-test p-values at $\nu = 5$ with true mean $\mu = 0.5$, for $n = 30$ and $n = 60$; the red rejection rate is the power.

```
par(mfrow = c(1, 2), mar = c(4, 4, 2.5, 1))
hist_reject(t.test.sim(N, 30, 3, mu.true = 0.5), t_main(30, 3, 0.5))
hist_reject(t.test.sim(N, 60, 3, mu.true = 0.5), t_main(60, 3, 0.5))
```

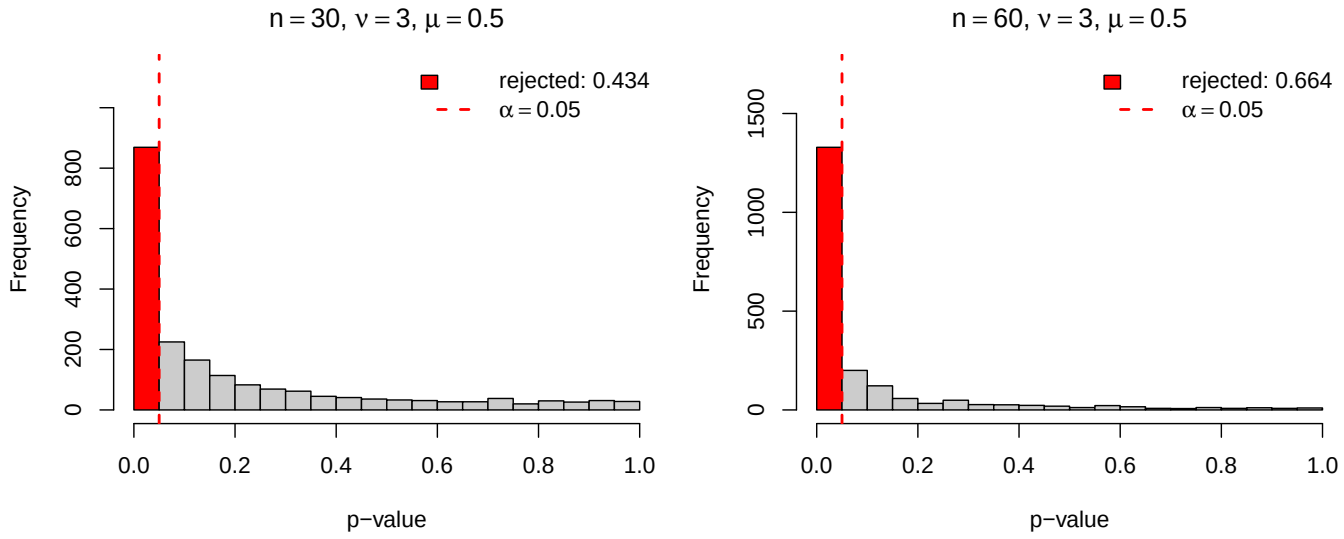


Figure 4.19: Repeats the power study for heavier-tailed data to show how tail weight affects the ability of the t-test to detect a real effect. t-test p-values at $\nu = 3$ with true mean $\mu = 0.5$, for $n = 30$ and $n = 60$; the red rejection rate is the power.

Heavier tails inflate the sample variance and so cost power; doubling n recovers much of it.

4.3.7 Monte Carlo for Calculating (Not Evaluating) a p-value

So far Monte Carlo has been used to *evaluate* a test whose p-value is known in closed form. When F_{H_0} is unknown, Monte Carlo can also *calculate* the p-value, by simulating T under H_0:

$$\widehat{\text{p-value}}(t^{obs}) = \frac{1 + \sum_{j=1}^{N} I(T^{(j)} \geq t^{obs})}{N+1}. \tag{4.27}$$

The "+1" counts the observed statistic as one draw from H_0; it keeps the estimate above 0 and makes the test exactly valid for any finite N.

As an example, we test $H_0 : \mu = 0$ for $n = 15$ normal observations using $T = |\bar{X}|/(s/\sqrt{n})$, pretending that we do not know its t-distribution.

```
x <- rnorm(15, mean = 0.6); tobs <- abs(mean(x)) / (sd(x) / sqrt(15))
Tnull <- replicate(5000, { y <- rnorm(15); abs(mean(y)) / (sd(y) / sqrt(15)) })
c(mc_pv = (1 + sum(Tnull >= tobs)) / 5001, exact_pv = 2 * pt(-tobs, df = 14))
```

```
    mc_pv  exact_pv
0.5278944 0.5319034
```

This works only because the null hypothesis here fully specifies the distribution of the data. When it does not, as in a test of independence, a **permutation test** generates the null distribution from the data themselves.

4.3.8 Example 2: A Permutation Test for the Correlation Coefficient

4.3.8.1 Setting and Procedure

We observe pairs (X_i, Y_i), $i = 1, \ldots, n$, and test $H_0 : X \perp Y$ (which implies $\text{corr}(X, Y) = 0$) with $T = |\hat{\rho}|$. Under H_0, **every pairing of the X values with the Y values is equally likely**, so the null distribution of T is obtained by re-computing T after randomly permuting Y:

1. Compute $\rho^{obs} = \hat{\rho}(X, Y)$.
2. For $j = 1, \ldots, m$: permute Y to get $Y^{(j)}$, and compute $\rho^{(j)} = \hat{\rho}(X, Y^{(j)})$.
3. Compute $\widehat{\text{p-value}} = \dfrac{1 + \#\{j : |\rho^{(j)}| \geq |\rho^{obs}|\}}{m+1}$.

No normality assumption is needed; the null distribution is generated **from the data itself**.

	X	observed Y	$Y^{(1)}$	$Y^{(2)}$	...	$Y^{(m)}$

4.3.8.2 Replicated Data Sets by Permutation

X is held fixed, and each of the m permutations re-shuffles the Y column (shown for $n = 5$):

	X	observed Y	$Y^{(1)}$	$Y^{(2)}$	...	$Y^{(m)}$
$i = 1$	x_1	y_1	y_3	y_5	$\cdots$	y_2
$i = 2$	x_2	y_2	y_1	y_4	$\cdots$	y_5
$i = 3$	x_3	y_3	y_5	y_1	$\cdots$	y_4
$i = 4$	x_4	y_4	y_2	y_3	$\cdots$	y_1
$i = 5$	x_5	y_5	y_4	y_2	$\cdots$	y_3
statistic		ρ^{obs}	$\rho^{(1)}$	$\rho^{(2)}$	...	$\rho^{(m)}$
as extreme?			$I(\lvert\rho^{(1)}\rvert \geq \lvert\rho^{obs}\rvert)$	$I(\lvert\rho^{(2)}\rvert \geq \lvert\rho^{obs}\rvert)$	...	$I(\lvert\rho^{(m)}\rvert \geq \lvert\rho^{obs}\rvert)$

$\rho^{(1)}, \ldots, \rho^{(m)}$ are draws from the null distribution of $\hat{\rho}$. The p-value is 1 plus the sum of the last row, divided by $m + 1$.

4.3.8.3 Implementation in R

```r
perm_cor_test <- function(x, y, m = 2000) {
  robs <- cor(x, y)
  rperm <- replicate(m, cor(x, sample(y)))
  list(rho = robs, pv = (1 + sum(abs(rperm) >= abs(robs))) / (m + 1), rperm = rperm)
}
```

4.3.8.4 Evaluation under H_0 and H_1

We generate non-normal (X, Y): independent under H_0, and $Y = 0.4X + \epsilon$ under H_1. We check that the permutation p-values are uniform with size α under H_0, and estimate the power under H_1. Each of the N simulated data sets is analysed with its own complete permutation test:

data set	data under H_0 (or H_1)	ρ^{obs}	permutation test	p-value	reject?
$k = 1$	$(X^{[1]}, Y^{[1]})$	ρ_1^{obs}	m permutations of $Y^{[1]}$	pv_1	$I(pv_1 < \alpha)$
$k = 2$	$(X^{[2]}, Y^{[2]})$	ρ_2^{obs}	m permutations of $Y^{[2]}$	pv_2	$I(pv_2 < \alpha)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

data set	data under H_0 (or H_1)	ρ^{obs}	permutation test	p-value	reject?
$k = N$	$(X^{[N]}, Y^{[N]})$	ρ_N^{obs}	m permutations of $Y^{[N]}$	pv_N	$I(pv_N < \alpha)$
summary				histogram	$\widehat{\text{size}}$ (or $\widehat{\text{power}}$) $= \frac{1}{N}\sum_k I(pv_k < \alpha)$

Below, $N = 500$ and $m = 500$.

```
sim_cor <- function(n = 30, b = 0) { x <- rexp(n); list(x = x, y = b * x + rexp(n)) }
eval_perm_cor <- function(b, N = 500) replicate(N, {
  d <- sim_cor(b = b)
  perm_cor_test(d$x, d$y, m = 500)$pv
})
pv_cor0 <- eval_perm_cor(b = 0)      # H0: independent
pv_cor1 <- eval_perm_cor(b = 0.4)    # H1: dependent

par(mfrow = c(1, 2), mar = c(4, 4, 2.5, 1))
hist_reject(pv_cor0, "permutation p-values under H0")
hist_reject(pv_cor1, "permutation p-values under H1")
```

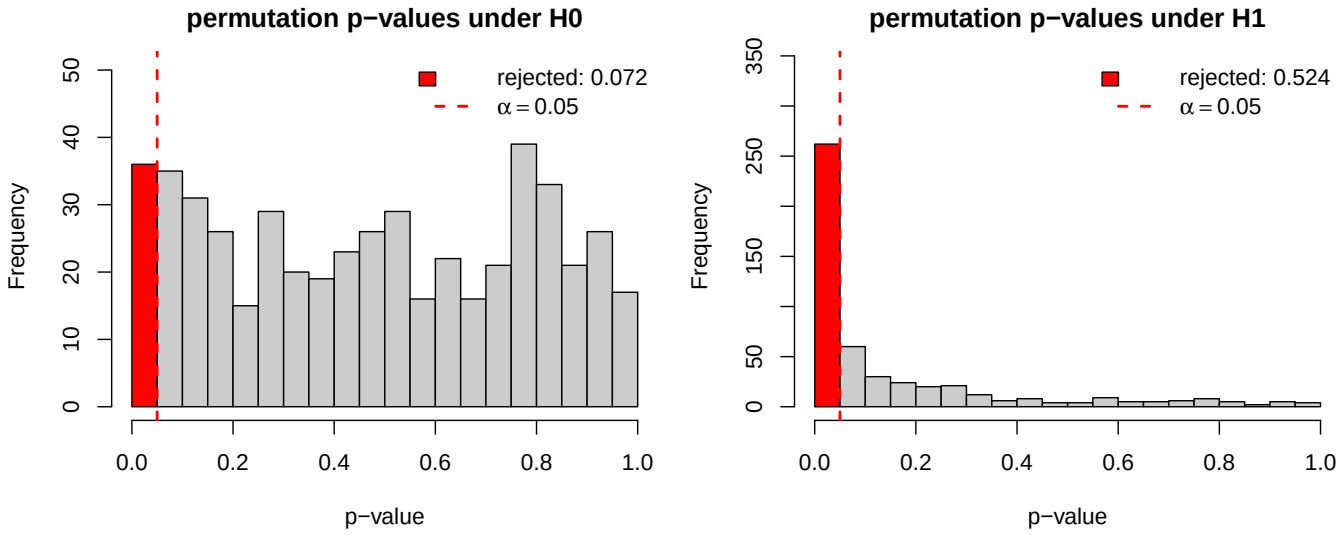


Figure 4.20: Verifies by simulation that the permutation test for correlation has the correct size without any normality assumption, and estimates its power. Permutation p-values for non-normal (X, Y): independent under H_0 (left), and $Y = 0.4X + \epsilon$ under H_1 (right); red bars are the rejections at $\alpha = 0.05$.

Under H_0 the permutation p-values are uniform on $[0, 1]$ and the size is close to 0.05, with no distributional assumption on (X, Y). Under H_1 they pile up near 0, and the proportion below 0.05 is the power.

4.3.9 Example 3: Permuation Correction for Post-Selection Linear Regression

4.3.9.1 Setting

We have a response Y and $p = 20$ candidate predictors $X_1, \ldots, X_{20}$, with $n = 50$, and want to test H_0: Y is unrelated to **all** of the X's. A common practice is to:

1. Pick the predictor X_{k^*} most correlated with Y.
2. Fit $Y \sim X_{k^*}$ and report the usual t-test p-value for its slope.

Is this a valid test of H_0? Does it have size α?

```
test_lm_sel <- function(y, X) {
  k <- which.max(abs(cor(X, y)))
  summary(lm(y ~ X[, k]))$coefficients[2, 4]    # t-test p-value of selected slope
}
```

The t-test p-value assumes that X_{k^*} was **chosen in advance**. Selecting the best of 20 predictors *after* looking at the data means that, even under H_0, $\max_k |\hat{\rho}_k|$ tends to be large. The reported p-value ignores this selection.

4.3.9.2 Simulation for Evaluating `test_lm_sel`

We simulate $N = 2000$ data sets with $Y = \beta X_1 + \epsilon$: $\beta = 0$ under H_0, and $\beta = 0.5$ under H_1.

```
sim_lm <- function(n = 50, p = 20, beta = 0) {
  X <- matrix(rnorm(n * p), n, p); list(y = beta * X[, 1] + rnorm(n), X = X)
}
pv_sel0 <- replicate(2000, { d <- sim_lm(beta = 0);   test_lm_sel(d$y, d$X) })
pv_sel1 <- replicate(2000, { d <- sim_lm(beta = 0.5); test_lm_sel(d$y, d$X) })

par(mfrow = c(1, 2), mar = c(4, 4, 2.5, 1))
hist_reject(pv_sel0, "test_lm_sel p-values under H0")
hist_reject(pv_sel1, "test_lm_sel p-values under H1")
```

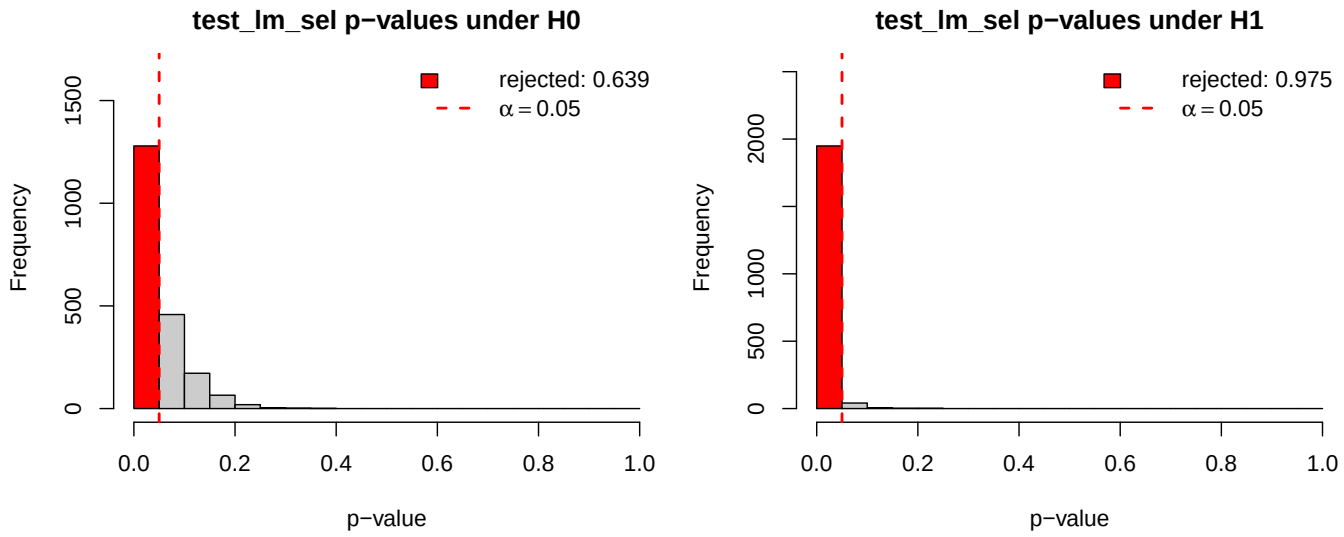


Figure 4.21: Demonstrates that choosing the best of many predictors using the data and then testing it as if it had been chosen in advance gives an invalid p-value. p-values of `test_lm_sel` under H_0 (left) and $H_1 : \beta = 0.5$ (right); under H_0 they are far from uniform, with a rejection rate far above 0.05.

The true size is far above 0.05: `test_lm_sel` rejects a true H_0 far too often. Under H_0 its p-value is **not** uniform, which is a general diagnostic for an invalid test. Its high rejection rate under H_1 is not real power, since the test does not have size α.

4.3.9.3 The Permutation Fix

The fix is to repeat the **whole procedure** (select the most correlated variable, fit the linear model, report its p-value) on permuted responses, and to compare the observed p-value with the permutation distribution of p-values:

1. Compute p^{obs}, the p-value of `test_lm_sel` on (Y, X).
2. For $j = 1, \ldots, m$: permute Y, and compute $p^{(j)}$, the p-value of `test_lm_sel` on $(Y^{(j)}, X)$, **re-doing the selection** each time.
3. Compute $\widehat{\text{p-value}} = \dfrac{1 + \#\{j : p^{(j)} \leq p^{obs}\}}{m + 1}$, the proportion of permuted p-values at least as small as the observed one.

The selected index $k^*(Y) = \arg\max_k |\hat{\rho}(X_k, Y)|$ is a function of the response, so each permuted $Y^{(j)}$ gets its **own** selected predictor $X_{k^*(Y^{(j)})}$, as Figure 4.22 shows.

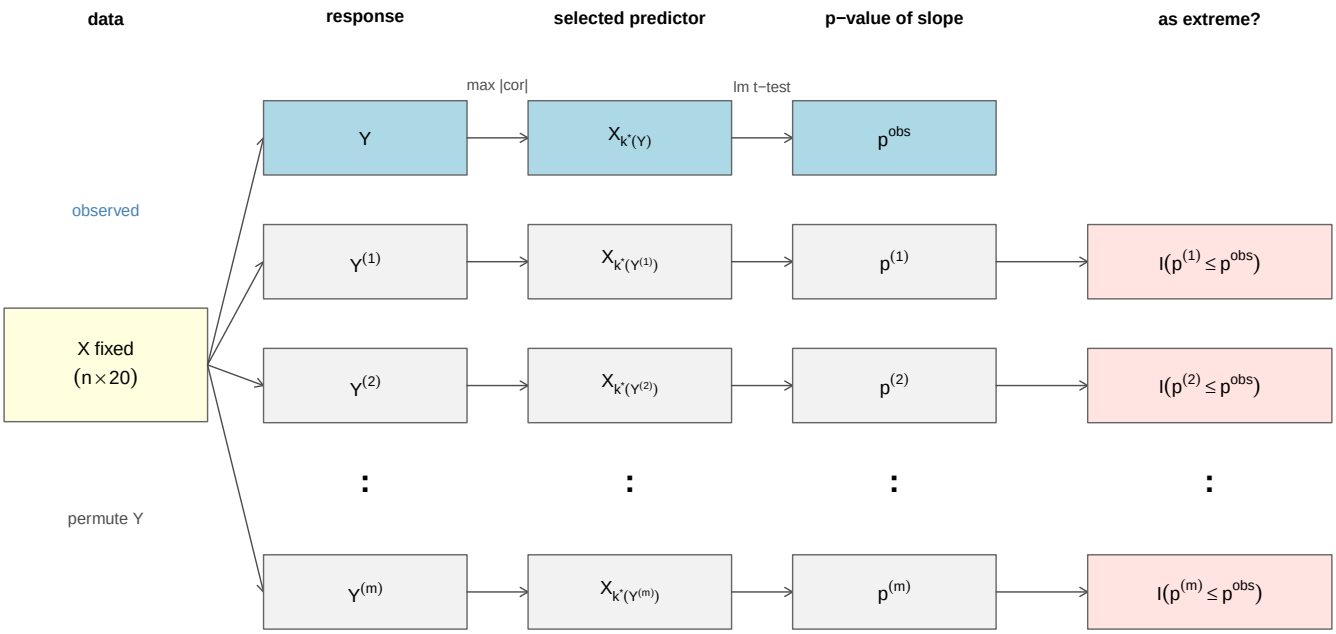


Figure 4.22: Shows why the selection step must be repeated inside the permutation loop: the selected predictor depends on the response, so every permuted response selects its own predictor. Tree of the permutation test after variable selection: the fixed X is paired with the observed Y and with each permuted $Y^{(j)}$, each response selects its own predictor $X_{k^*(Y^{(j)})}$, and the resulting slope p-value $p^{(j)}$ is compared with p^{obs}.

Keeping $X_{k^*(Y)}$ fixed and only permuting Y would **not** reproduce the selection step, and the resulting test would still be invalid.

```
perm_lm_sel <- function(y, X, m = 500) {
  pobs <- test_lm_sel(y, X)
  pperm <- replicate(m, test_lm_sel(sample(y), X))
  (1 + sum(pperm <= pobs)) / (m + 1)
}
```

4.3.9.4 Evaluation under H_0 and H_1

We simulate (Y, X) from `sim_lm`, with $\beta = 0$ under H_0 and $\beta = 0.5$ under H_1, and run a complete permutation test on each data set:

data set	data under H_0 (or H_1)	p^{obs}	permutation test	p-value	reject?
$k = 1$	$(Y^{[1]}, X^{[1]})$	p_1^{obs}	m permutations of $Y^{[1]}$	pv_1	$I(pv_1 < \alpha)$
$k = 2$	$(Y^{[2]}, X^{[2]})$	p_2^{obs}	m permutations of $Y^{[2]}$	pv_2	$I(pv_2 < \alpha)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

data set	data under H_0 (or H_1)	p^{obs}	permutation test	p-value	reject?
$k = N$	$(Y^{[N]}, X^{[N]})$	p_N^{obs}	m permutations of $Y^{[N]}$	pv_N	$I(pv_N < \alpha)$
summary				histogram	$\widehat{\text{size}}$ (or $\widehat{\text{power}}$) $= \frac{1}{N}\sum_k I(pv_k < \alpha)$

Below, $N = 300$ and $m = 500$.

```
eval_perm_lm_sel <- function(beta, N = 300) replicate(N, {
  d <- sim_lm(beta = beta)
  perm_lm_sel(d$y, d$X, m = 500)
})
pv_perm0 <- eval_perm_lm_sel(beta = 0)      # H0: Y unrelated to X
pv_perm1 <- eval_perm_lm_sel(beta = 0.5)    # H1: Y depends on X1

par(mfrow = c(1, 2), mar = c(4, 4, 2.5, 1))
hist_reject(pv_perm0, "perm_lm_sel p-values under H0")
hist_reject(pv_perm1, "perm_lm_sel p-values under H1")
```

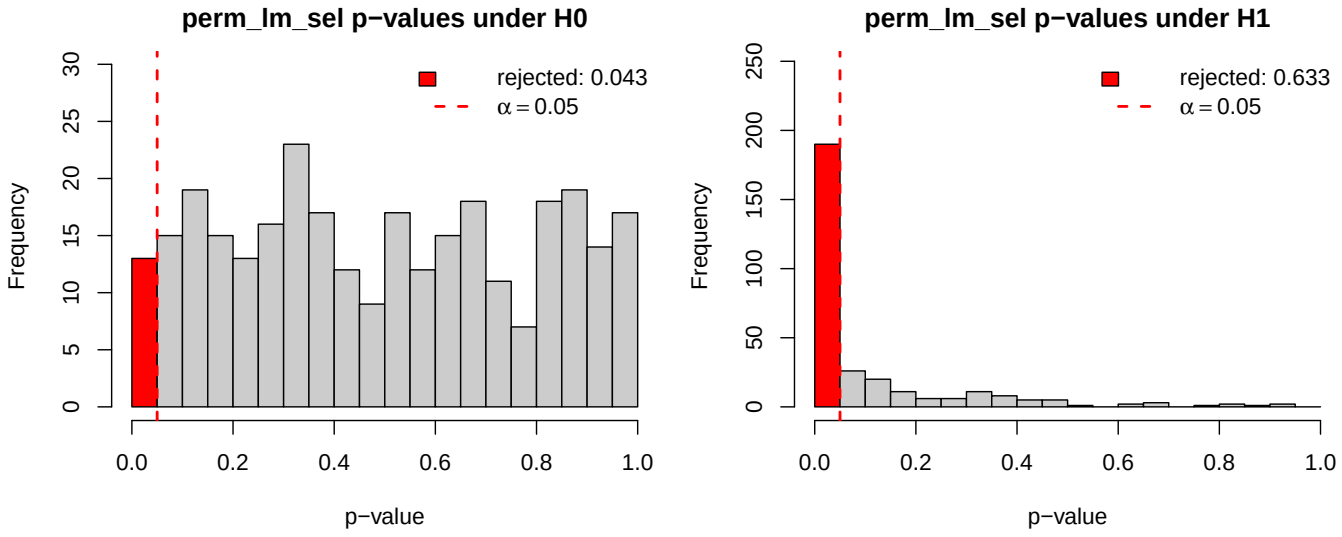


Figure 4.23: Shows that a permutation test which repeats the whole selection step on each shuffled data set restores valid p-values while keeping good power. Permutation p-values after variable selection under H_0 (left) and $H_1 : \beta = 0.5$ (right): uniform with size near 0.05 under H_0, and concentrated near 0 under H_1.

Under H_0 the permutation p-values are uniform and the size is close to 0.05; under H_1 they pile up near 0, and the proportion below 0.05 is an honest power. The naive test rejects more often under H_1 only because it also rejects far too often under H_0: power comparisons are meaningful only **between tests of the same size**.

4.3.10 Advanced Computation-Intensive Tests

The examples above exposed two ways in which a textbook p-value can be biased:

1. **A misspecified null distribution.** The reference distribution is derived under assumptions (normality, known parameters, large samples) that the data do not satisfy, as for the t-test on heavy-tailed data.
2. **A data-dependent procedure.** The statistic is the result of a search (over predictors, models, thresholds or tuning parameters), but its p-value is computed as if the final choice had been fixed in advance, as for `test_lm_sel`.

Every method reviewed below corrects these biases with the same recipe: identify what is random under H_0, **re-generate data from H_0 by simulation or resampling, re-run the entire procedure on each replicate**, and read the p-value off the resulting reference distribution. The methods differ in how the null data are produced: by permutation when H_0 is an exchangeability hypothesis, by a parametric bootstrap from a fitted null model when H_0 leaves parameters unknown, and by resampling one variable given the others when H_0 is a conditional independence hypothesis.

4.3.10.1 Monte Carlo Tests

The idea of computing a p-value by ranking the observed statistic among statistics simulated under H_0 goes back to Barnard (1963) and was developed by Hope (1968). If $T^{(1)}, \dots, T^{(m)}$ are simulated under a **simple** null hypothesis, then under H_0 the observed T is exchangeable with them, so

$$\widehat{\text{p-value}} = \frac{1 + \#\{j : T^{(j)} \geq T^{obs}\}}{m + 1} \tag{4.28}$$

gives an **exactly** valid test for any m, not just as $m \to \infty$. Phipson and Smyth (2010) show that the tempting estimate $\#\{j : T^{(j)} \geq T^{obs}\}/m$ can be zero and is anti-conservative, which matters most when many tests are corrected for multiplicity. Besag and Clifford (1989) extend Monte Carlo tests to nulls that can only be simulated by MCMC, and Besag and Clifford (1991) give a sequential version that stops simulating early once the p-value is clearly large, saving most of the computation for unremarkable data.

4.3.10.2 Parametric Bootstrap Tests with Estimated Parameters

Most null hypotheses are composite: they specify a family of distributions but not its parameters. Plugging estimated parameters into a reference distribution derived for known parameters biases the test. The classic example is the Kolmogorov–Smirnov (KS) test of normality with the mean and variance estimated from the data. The fitted normal is closer to the data than the true normal, so the KS distance is too small and the test is **very conservative**. Lilliefors (1967) tabulated the correct null distribution by Monte Carlo.

The general remedy is the **parametric bootstrap test**: fit the model under H_0, simulate data sets from the fitted null model, **re-estimate the parameters on each simulated data set**, and recompute the statistic. For normality testing, the KS statistic of standardized data is location–scale invariant, so its null distribution does not depend on (μ, σ) at all, and simulating from $N(0, 1)$ gives an exact Monte Carlo test. Stute et al. (1993) establish the validity of this bootstrap for general goodness-of-fit tests with estimated parameters.

```r
ks_stat <- function(x) {                        # KS distance to the fitted normal
  z <- sort((x - mean(x)) / sd(x)); n <- length(z); p <- pnorm(z)
  max(pmax((1:n) / n - p, p - (0:(n - 1)) / n))
}
ks_naive <- function(x)                         # pretends (mean, sd) were known
  suppressWarnings(ks.test(x, "pnorm", mean(x), sd(x))$p.value)
ks_mc <- function(x, m = 200) {                 # re-estimates (mean, sd) on each null data set
  Dnull <- replicate(m, ks_stat(rnorm(length(x))))
  (1 + sum(Dnull >= ks_stat(x))) / (m + 1)
}
```

```r
eval_ks <- function(rdata, N = 1000) replicate(N, {
  x <- rdata(); c(naive = ks_naive(x), mc = ks_mc(x))
})
ks_settings <- list("normal (H0)" = function() rnorm(50),
                    "t3 (H1)"     = function() rt(50, df = 3),
                    "exp (H1)"    = function() rexp(50))
ks_pv <- lapply(ks_settings, eval_ks)

par(mfrow = c(2, 3), mar = c(4, 4, 2.5, 1))
for (test in c("naive", "mc"))
  for (s in names(ks_pv))
    hist_reject(ks_pv[[s]][test, ], paste(test, "KS test:", s))
```

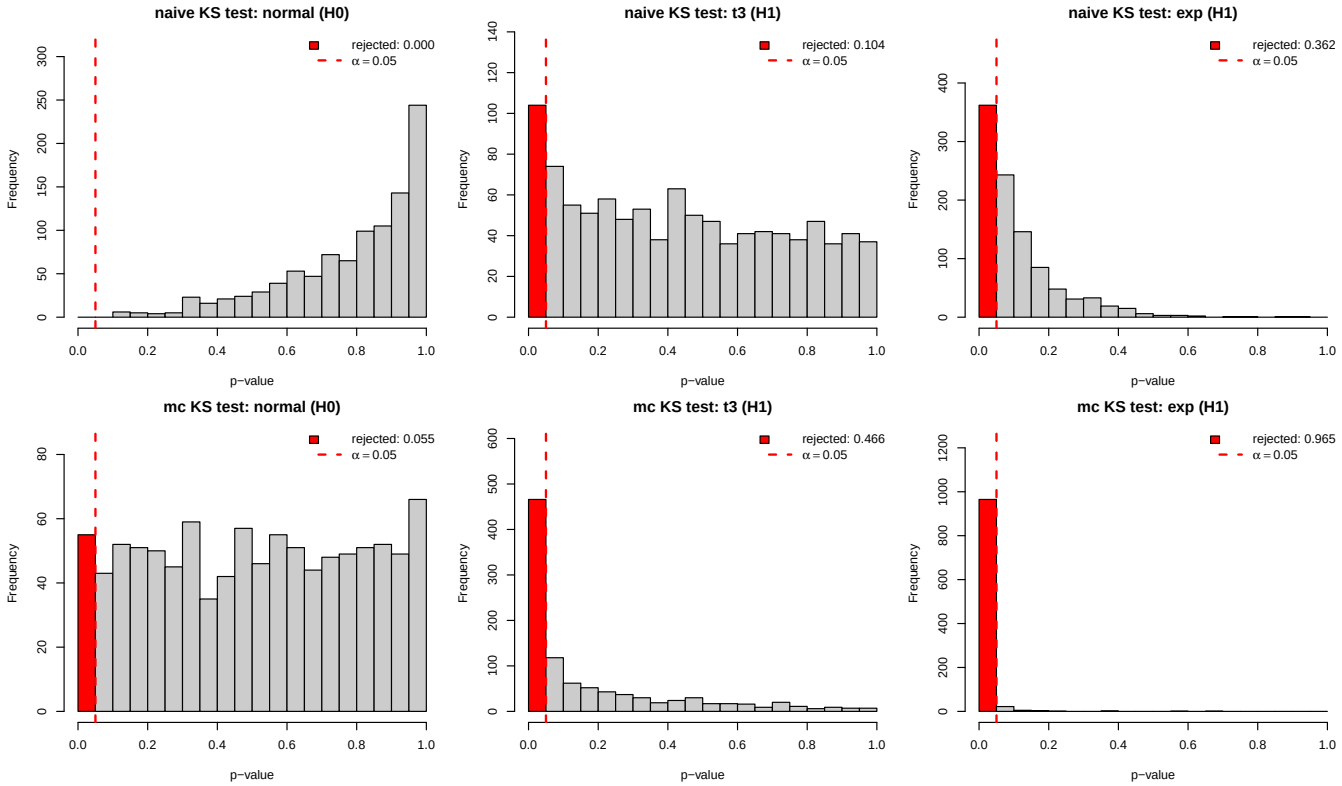


Figure 4.24: Shows how plugging estimated parameters into a textbook null distribution biases a goodness-of-fit test, and how a parametric bootstrap that re-estimates the parameters on every simulated data set removes the bias. p-values of the KS normality test with estimated mean and variance for $n = 50$: the naive test (top) against the Monte Carlo test (bottom), for normal data (H_0), t_3 data and exponential data; red bars are the rejections at $\alpha = 0.05$.

Under H_0 the naive p-values pile up near 1 and almost never reject, while the Monte Carlo p-values are uniform. Because the naive test is so conservative, it also has little power against both heavy-tailed and skewed alternatives. Correcting the size recovers most of that power.

Two guidelines from Hall and Wilson (1991) apply to every bootstrap test. First, resample from a distribution that **satisfies** H_0 (for example, re-centre the data before resampling when testing a mean), not from the empirical distribution of the original data. Second, prefer an approximately **pivotal** statistic, such as a studentized one, whose null distribution depends little on the unknown parameters.

4.3.10.3 Maximum Statistics and Multiple Testing

The selection example is one instance of a general problem: the reported statistic is the **maximum** of many statistics, $T_{\max} = \max_k T_k$, and each T_k is referred to its own marginal null distribution. Westfall and Young (1993) turned the permutation fix into a general method for multiple testing. Permuting the rows of the response jointly across all K hypotheses and recording $\max_k T_k$ on each permutation gives adjusted p-values

$$\tilde{p}_k = \frac{1 + \#\{j : \max_l T_l^{(j)} \geq T_k^{obs}\}}{m+1}, \tag{4.29}$$

which control the family-wise error rate. Unlike the Bonferroni correction, the **maxT** (and the analogous **minP**) procedure automatically accounts for the correlation among the T_k, and so is less conservative when the tests are dependent, as with nearby genes, voxels or genetic markers.

Scan statistics follow the same pattern with a search over locations. The spatial scan statistic of Kulldorff (1997) maximizes a likelihood ratio over many circular windows to find a disease cluster, and obtains its p-value by Monte Carlo replication of the whole search under a null of no clustering.

4.3.10.4 Nuisance Parameters Identified Only under the Alternative

In some problems the search is over a parameter that has no meaning under H_0. Examples are the unknown date of a structural break in a regression, the threshold of a threshold autoregression, and the location of an extra component in a mixture model. Davies (1977) showed that the natural statistic is then a supremum over the unidentified parameter, and that it does not have the usual χ^2 distribution. Andrews (1993) derived the asymptotic null distribution of the sup-Wald (sup-F) test for a change point at an unknown date. Because that distribution depends on the design, Hansen (1996) proposed obtaining p-values by simulation.

Testing the **number of components** of a mixture is a related case in which the regularity conditions for the likelihood ratio test fail because the null lies on the boundary of the parameter space. McLachlan (1987) proposed calibrating the likelihood ratio statistic by a **parametric bootstrap**. Data sets are simulated from the fitted g-component model, and **both** the g- and $(g+1)$-component models are refitted on each one, since the maximization over the extra component is exactly the search that biases the naive χ^2 reference.

4.3.10.5 Calibrating Asymptotic Tests: The Double Bootstrap

A test based on an asymptotic reference distribution has a level error that shrinks only slowly with n. Beran (1988) proposed **prepivoting**: transform the statistic by its own estimated (bootstrap) null CDF, which makes it closer to pivotal and reduces the level error. Iterating the idea gives the **double bootstrap**. The p-value $\hat{p}$ is computed by an inner bootstrap, and an outer bootstrap estimates the distribution of $\hat{p}$ under the fitted null model. The corrected p-value is $\Pr^*(\hat{p}^* \leq \hat{p})$, which is exactly the "p-value of a p-value" idea behind the permutation fix for variable selection. The cost is about $B_1 \times B_2$ model fits; Davison and Hinkley (1997) discuss the implementation and when the extra accuracy is worth it.

4.3.10.6 Conditional Randomization Tests

Permuting a response tests **marginal** independence. In regression we usually want to test whether X_k is associated with Y **given** the other covariates X_{-k}, which cannot be done by simply permuting X_k when the covariates are correlated. The **conditional randomization test** of Candès et al. (2018) assumes the distribution of $X_k \mid X_{-k}$ is known or well estimated. It re-draws $X_k^{(j)}$ from that conditional distribution, recomputes any test statistic (for example, the absolute lasso coefficient of X_k, with the tuning parameter re-selected each time), and ranks the observed statistic among the simulated ones. The p-value is exactly valid for any statistic, however it was tuned, as

long as the whole fitting procedure is repeated on every draw. The **model-X knockoffs** in the same paper replace the m refits by a single fit with synthetic "knockoff" copies of all the covariates, which makes the idea feasible for thousands of variables.

4.3.10.7 Testing the Performance of Tuned Predictive Models

A cross-validated accuracy is itself an optimistically biased statistic when the classifier's features or tuning parameters were chosen on the same data. Ojala and Garriga (2010) test whether a classifier has learned anything by permuting the class labels and re-running the **complete** pipeline of feature selection, tuning and cross-validation on each permuted data set. The permutation distribution of the accuracy then serves as its null distribution. Re-running only the final cross-validation step on permuted labels, with the features selected on the original labels, repeats the mistake of keeping $X_{k^*(Y)}$ fixed in the selection example.

4.3.10.8 Summary

method	source of bias	how null data are generated	what is re-run on each replicate
Monte Carlo test	unknown null distribution	simulate from the simple H_0	the statistic
Parametric bootstrap (Lilliefors, mixtures)	parameters estimated from the data	simulate from the fitted null model	parameter estimation and the statistic
Permutation maxT / minP	maximum over many tests	permute the response jointly	all K statistics and their maximum
Scan and sup-tests	search over locations or change points	simulate or permute under H_0	the whole search
Double bootstrap	slow asymptotics	nested bootstrap from the fitted null model	the inner bootstrap p-value
Conditional randomization	correlated covariates, tuned fits	draw $X_k \mid X_{-k}$	the full model fit, including tuning
Permutation test of a classifier	feature selection and tuning	permute the labels	the complete training pipeline

The common lesson is the one from the variable-selection example: a resampling test is valid only if **everything that looked at the data** is repeated on every null replicate.

4.3.10.9 References

- Andrews, D. W. K. (1993). Tests for parameter instability and structural change with unknown change point. *Econometrica*, 61(4), 821–856.
- Barnard, G. A. (1963). Discussion of "The spectral analysis of point processes" by M. S. Bartlett. *Journal of the Royal Statistical Society, Series B*, 25, 294.

- Beran, R. (1988). Prepivoting test statistics: A bootstrap view of asymptotic refinements. *Journal of the American Statistical Association*, 83(403), 687–697.
- Besag, J. and Clifford, P. (1989). Generalized Monte Carlo significance tests. *Biometrika*, 76(4), 633–642.
- Besag, J. and Clifford, P. (1991). Sequential Monte Carlo p-values. *Biometrika*, 78(2), 301–304.
- Candès, E., Fan, Y., Janson, L. and 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.
- Davies, R. B. (1977). Hypothesis testing when a nuisance parameter is present only under the alternative. *Biometrika*, 64(2), 247–254.
- Davison, A. C. and Hinkley, D. V. (1997). *Bootstrap Methods and Their Application*. Cambridge University Press.
- Hall, P. and Wilson, S. R. (1991). Two guidelines for bootstrap hypothesis testing. *Biometrics*, 47(2), 757–762.
- Hansen, B. E. (1996). Inference when a nuisance parameter is not identified under the null hypothesis. *Econometrica*, 64(2), 413–430.
- Hope, A. C. A. (1968). A simplified Monte Carlo significance test procedure. *Journal of the Royal Statistical Society, Series B*, 30(3), 582–598.
- Kulldorff, M. (1997). A spatial scan statistic. *Communications in Statistics – Theory and Methods*, 26(6), 1481–1496.
- Lilliefors, H. W. (1967). On the Kolmogorov–Smirnov test for normality with mean and variance unknown. *Journal of the American Statistical Association*, 62(318), 399–402.
- McLachlan, G. J. (1987). On bootstrapping the likelihood ratio test statistic for the number of components in a normal mixture. *Applied Statistics*, 36(3), 318–324.
- Ojala, M. and Garriga, G. C. (2010). Permutation tests for studying classifier performance. *Journal of Machine Learning Research*, 11, 1833–1863.
- Phipson, B. and Smyth, G. K. (2010). Permutation p-values should never be zero: Calculating exact p-values when permutations are randomly drawn. *Statistical Applications in Genetics and Molecular Biology*, 9(1), Article 39.
- Stute, W., González Manteiga, W. and Presedo Quindimil, M. (1993). Bootstrap based goodness-of-fit tests. *Metrika*, 40, 243–256.
- Westfall, P. H. and Young, S. S. (1993). *Resampling-Based Multiple Testing: Examples and Methods for p-Value Adjustment*. Wiley.

5 Optimization for Maximum Likelihood Estimation

This chapter develops maximum likelihood estimation and the numerical methods used to find it, first for a scalar parameter and then for a parameter vector. The univariate case, where the negative log-likelihood can be plotted and a root of its derivative can be bracketed, motivates the two ideas that carry over unchanged to higher dimensions: replacing that function locally by a quadratic and solving for its minimum (Newton–Raphson), and using the curvature of that quadratic to report a standard error. The multivariate case, where neither plotting nor bracketing is available, shows what survives of that idea and what has to be added — safeguards against divergence, and a wider menu of derivative-free and gradient-only methods (gradient descent, conjugate gradient, BFGS, stochastic gradient descent, and Nelder–Mead) for when the Hessian is unavailable or too expensive. Several of these methods are illustrated below with live, in-browser animations that let you change the objective function, the starting point, and the algorithm before watching the iterates respond.

5.1 Maximum Likelihood: Theory

5.1.1 Likelihood, Negative Log-Likelihood, and Information

A model specifies the density or probability of the data D given a parameter θ, written $f(D \mid \theta)$. As a function of D with θ fixed it is the sampling distribution; as a function of θ with D observed it is the likelihood. For independent observations $x_1, \dots, x_n$,

$$L(\theta) = \prod_{i=1}^{n} f(x_i \mid \theta), \qquad \ell(\theta) = \log L(\theta) = \sum_{i=1}^{n} \log f(x_i \mid \theta). \tag{5.1}$$

We never work with L itself: the product underflows to zero for even moderate n, and sums are easier to differentiate. Maximizing L is the same as maximizing ℓ, which in turn is the same as **minimizing** the negative log-likelihood

$$\mathcal{L}(\theta) = -\ell(\theta) = -\sum_{i=1}^{n} \log f(x_i \mid \theta) = \sum_{i=1}^{n} \mathcal{L}_i(\theta), \tag{5.2}$$

and it is the minimization form that the rest of this chapter — and every optimizer in R — actually uses. Where $\mathcal{L}$ is smooth with an interior minimum, $\widehat{\theta}$ solves,

$$g(\theta) = \mathcal{L}'(\theta) = 0, \qquad g'(\widehat{\theta}) > 0. \tag{5.3}$$

Three quantities recur throughout the chapter:

- the **gradient** $g(\theta) = \mathcal{L}'(\theta) = -\ell'(\theta)$, whose root locates the minimum;
- the **observed information** $J(\theta) = g'(\theta) = \mathcal{L}''(\theta) = -\ell''(\theta)$, the curvature of $\mathcal{L}$ at θ;

- the **expected information** $nI(\theta) = E_D\left[g'(\theta; D)\right]$, its expectation under the model.

The derivative of the *log*-likelihood, $\ell'(\theta) = -g(\theta)$, is what the statistical literature calls the **score**, and $g(\theta) = 0$ is the **score equation** written with the opposite sign. That terminology survives in two standard names used later — the *method of scoring*, and Fisher information defined as the variance of the score — but the algorithms below are all stated in terms of $\mathcal{L}$ and g, so the score does not appear as a separate object.

The second derivative plays a double role. It determines how a Newton-type algorithm steps toward the minimum, and it determines how precisely the data pin the parameter down. An optimizer that computes the curvature has computed the standard error at the same time.

5.1.2 Example: iid Bernoulli

Before turning to a model with a continuous parameter and an exactly quadratic negative log-likelihood, it is worth solving $g(\theta) = 0$ by hand in the simplest possible case. Let $Y_1, \ldots, Y_n \overset{iid}{\sim} \text{Bernoulli}(\theta)$, so that $P(y_i \mid \theta) = \theta^{y_i}(1-\theta)^{1-y_i}$. Writing $n_1 = \sum_i y_i$ for the number of successes and $n_0 = n - n_1$ for the number of failures,

$$L(\theta) = \theta^{n_1}(1-\theta)^{n_0}, \qquad \mathcal{L}(\theta) = -n_1 \log\theta - n_0 \log(1-\theta). \tag{5.4}$$

Its derivative is,

$$g(\theta) = -\frac{n_1}{\theta} + \frac{n_0}{1-\theta}, \tag{5.5}$$

and setting $g(\theta) = 0$ gives $n_1(1-\theta) = n_0\theta$, i.e. $n_1 = (n_1 + n_0)\theta = n\theta$, so

$$\hat{\theta} = \frac{n_1}{n} = \overline{y}. \tag{5.6}$$

The second derivative $g'(\theta) = n_1/\theta^2 + n_0/(1-\theta)^2$ is positive wherever it is defined, confirming that $\overline{y}$ is a minimum of $\mathcal{L}$ and not a saddle point. With only three observations, $y = (1, 1, 0)$, both the likelihood and $\mathcal{L}$ can be tabulated and plotted directly, which makes $\hat{\theta} = 2/3 = \overline{y}$ visible without any calculus at all — the value that makes the data most probable is exactly the value that makes $\mathcal{L}$ smallest.

θ	$L(\theta) = \theta^2(1-\theta)$	$\mathcal{L}(\theta) = -\log L(\theta)$
0	0	∞
$1/2$	$1/8 = 0.1250$	2.079
$2/3$	$4/27 = 0.1481$ **(max)**	1.910 **(min)**
$3/4$	$9/64 = 0.1406$	1.962
1	0	∞

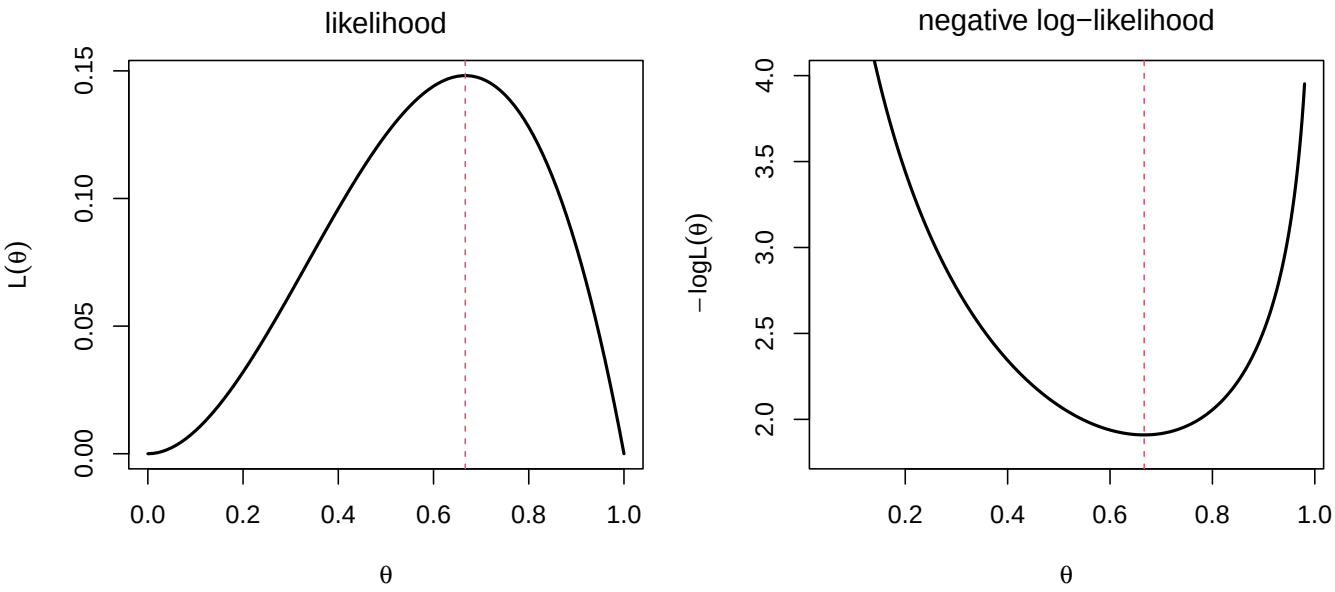


Figure 5.1: Introduces the likelihood and the negative log-likelihood with the simplest possible example, showing that the parameter value making the data most probable is the one that minimizes the function every optimizer in this chapter works with. Likelihood and negative log-likelihood of θ for three Bernoulli observations $y = (1, 1, 0)$: the maximum of L and the minimum of $\mathcal{L}$ both sit at $\hat{\theta} = \bar{y} = 2/3$.

5.1.3 Closed Form: the Normal Mean

Let $X_1, \dots, X_n$ be iid $N(\mu, \sigma_0^2)$ with σ_0^2 known. Using the identity $\sum_i (x_i - \mu)^2 = \sum_i (x_i - \overline{x})^2 + n(\overline{x} - \mu)^2$,

$$\mathcal{L}(\mu) = \frac{n}{2} \log\left(2\pi\sigma_0^2\right) + \frac{1}{2\sigma_0^2} \sum_{i=1}^{n} (x_i - \mu)^2 = C + \frac{n}{2\sigma_0^2} (\mu - \overline{x})^2. \tag{5.7}$$

Differentiating,

$$g(\mu) = \frac{n}{\sigma_0^2} (\mu - \overline{x}), \qquad g'(\mu) = \frac{n}{\sigma_0^2}, \qquad \hat{\mu} = \overline{x}. \tag{5.8}$$

Every feature of the general theory is visible here in exact form. $\mathcal{L}$ is an upward parabola, so it has a unique minimum. Its derivative g is linear, so $g(\mu) = 0$ is solved in one line. The curvature $J(\mu) = g'(\mu) = n/\sigma_0^2$ does not depend on μ or on the data, so observed and expected information coincide, $J = nI$, and both grow linearly with n: doubling the sample doubles the sharpness of the well and halves the variance of $\hat{\mu}$.

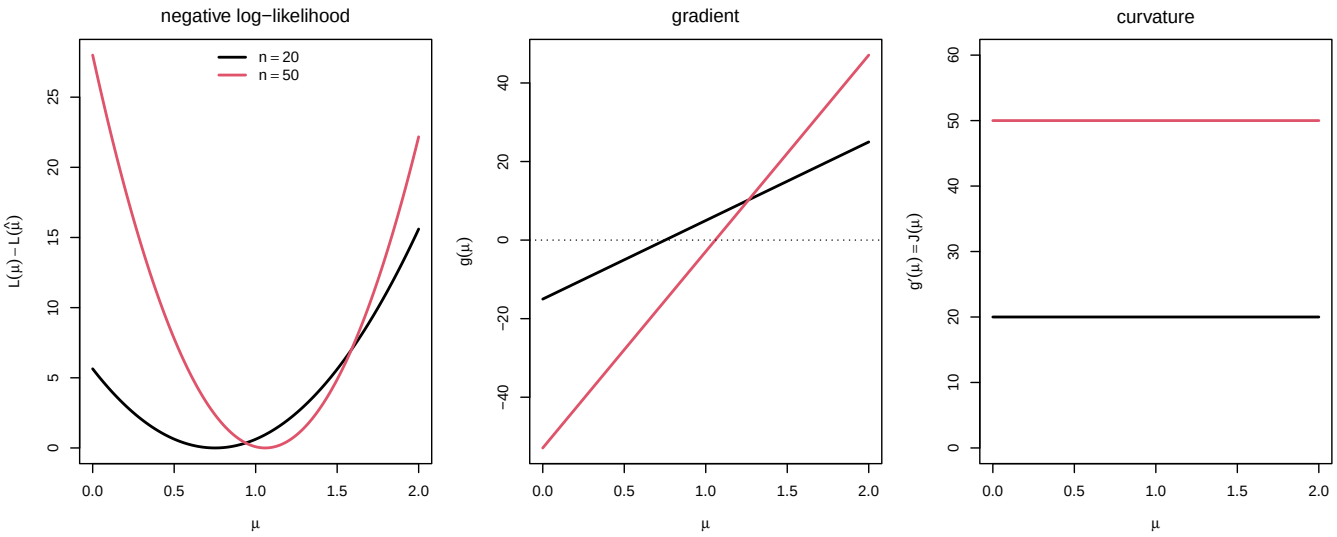


Figure 5.2: Relates the three central objects of maximum likelihood, the negative log-likelihood, its gradient, and its curvature, and shows how each changes with sample size. Negative log-likelihood, its gradient g, and its curvature for a normal mean at $n = 20$ and $n = 50$: the larger sample gives a sharper well, a steeper g, and larger curvature.

The derivative g crosses zero exactly where $\mathcal{L}$ bottoms out, and the larger sample gives a sharper well, a steeper g, and a larger curvature.

5.1.4 Consistency, Asymptotic Normality, and Fisher Information

Under regularity conditions — the support of f not depending on θ, the true value interior to the parameter space, and two derivatives passing under the integral sign — the MLE has three properties as $n \to \infty$, with θ_* the true parameter.

Consistency. $\hat{\theta} \to \theta_*$ in probability. The reason is that $\mathcal{L}(\theta)/n$ converges to $-E_{\theta_*}[\log f(X \mid \theta)]$, which by the information inequality is minimized at $\theta = \theta_*$: the surface being minimized converges to one whose lowest point is at the truth.

Asymptotic normality.

$$\hat{\theta} \dot{\sim} N\left(\theta_*,\ [nI(\theta_*)]^{-1}\right), \qquad I(\theta) = -E_X\left[\frac{\partial^2}{\partial\theta^2}\log f(X \mid \theta)\right] = E_X\left[\left(\frac{\partial}{\partial\theta}\log f(X \mid \theta)\right)^2\right]. \tag{5.9}$$

A sketch: expand $g(\hat{\theta}) = 0$ about θ_* to get $\hat{\theta} - \theta_* \approx -g(\theta_*)/g'(\theta_*)$. The numerator is a sum of iid mean-zero terms — this is the step in which $I(\theta)$ appears as the variance of the score — hence asymptotically normal with variance $nI(\theta_*)$; the denominator converges to $nI(\theta_*)$ by the law of large numbers. Dividing gives variance $[nI(\theta_*)]^{-1}$.

Asymptotic efficiency. No consistent estimator has smaller asymptotic variance — the Cramér–Rao bound is attained.

Since θ_* is unknown, the variance is estimated by substituting $\hat{\theta}$. Two versions are available: the expected information $nI(\hat{\theta})$, which requires the expectation analytically, and the observed information $J(\hat{\theta}) = g'(\hat{\theta})$, which requires only the curvature that a Newton step already computes. By the law of large numbers $J(\hat{\theta})/n \approx I(\hat{\theta})$, and both give

$$\widehat{\mathrm{SE}}(\hat{\theta}) = J(\hat{\theta})^{-1/2} \quad \text{or} \quad \left[nI(\hat{\theta})\right]^{-1/2}, \qquad \hat{\theta} \pm 1.96\,\widehat{\mathrm{SE}}(\hat{\theta}). \tag{5.10}$$

These are asymptotic statements. A flat $\mathcal{L}$ — small curvature — gives a large standard error, and an $\mathcal{L}$ that is far from quadratic gives a standard error that should not be trusted at all; the last section of this chapter shows a case of exactly that.

5.1.5 Demonstration: the Cauchy Location Model

The Cauchy density $f(x \mid \theta) = 1/\{\pi[1 + (x - \theta)^2]\}$ has no closed-form MLE:

$$\mathcal{L}(\theta) = n \log \pi + \sum_{i=1}^{n} \log\left(1 + (x_i - \theta)^2\right), \qquad g(\theta) = -\sum_{i=1}^{n} \frac{2(x_i - \theta)}{1 + (x_i - \theta)^2}. \tag{5.11}$$

Setting $g(\theta) = 0$ gives a polynomial equation of degree $2n - 1$. Plotting $\mathcal{L}$ for samples of increasing size shows the well sharpening around the truth, which is consistency and growing information in one picture.

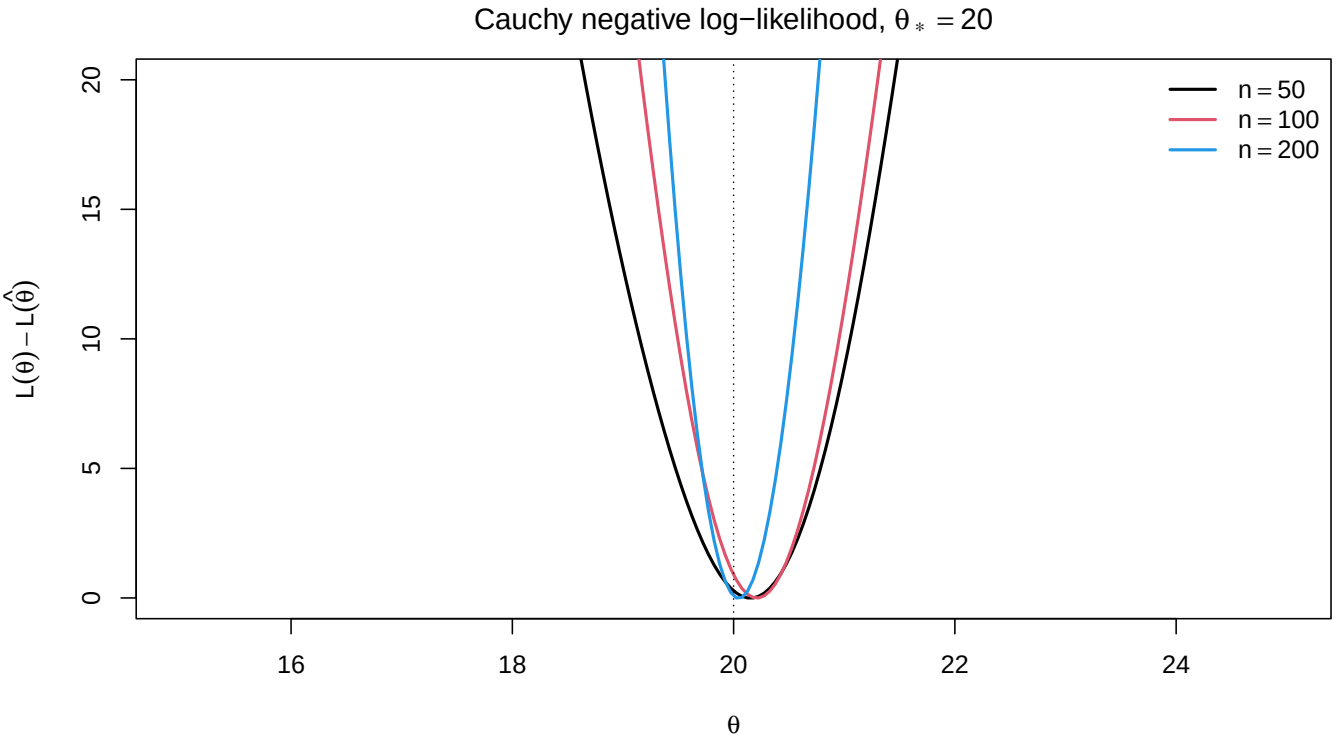


Figure 5.3: Shows how the Cauchy location negative log-likelihood, which is harder to minimize than the normal one, concentrates around the true value as the sample size grows. Cauchy negative log-likelihood for samples of increasing size ($n = 50, 100, 200$) from a true location $\theta_* = 20$: the well sharpens around the truth as n grows.

The variability of the MLE across repeated samples is the sampling distribution the theory describes. Repeating the experiment 50 times at two sample sizes shows the curves clustering more tightly, and their minima concentrating, as n grows.

```
             mean       sd
n = 20  20.05930 0.3843737
n = 200 20.01005 0.0920729
```

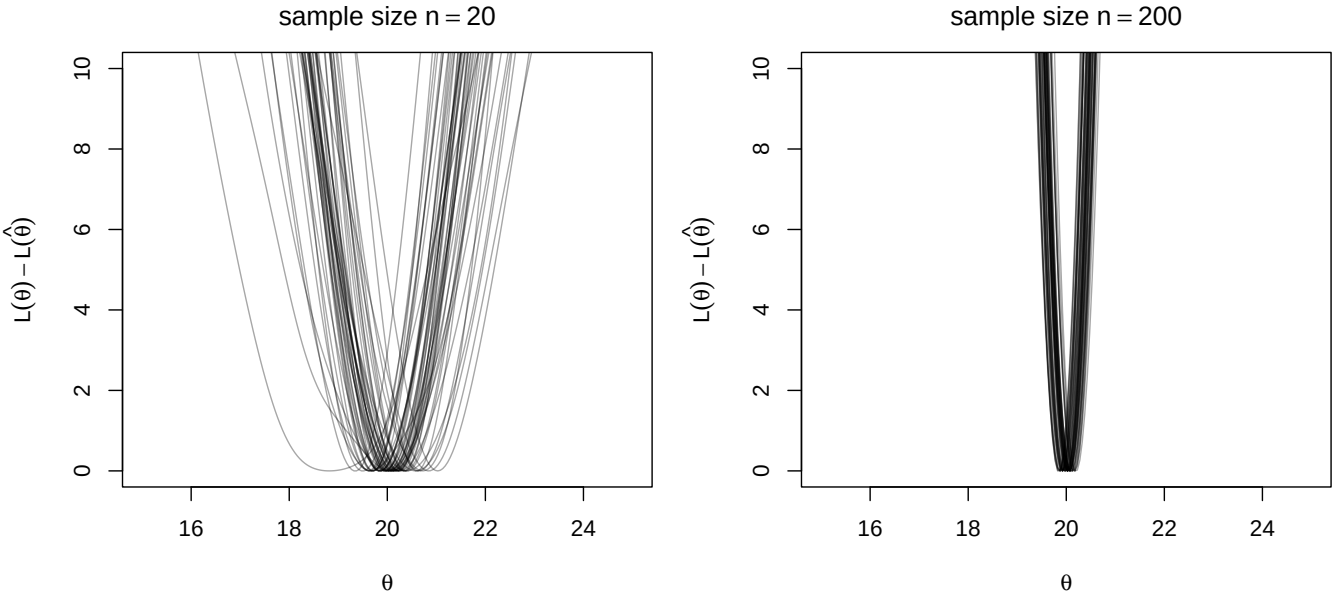


Figure 5.4: Shows the sampling variability of the negative log-likelihood itself: different data sets give different curves, and the minima of those curves, the MLEs, become less variable as n grows. 50 repeated Cauchy negative log-likelihood curves at $n = 20$ versus $n = 200$: the curves cluster more tightly, and their minima concentrate, as n grows.

The standard deviation falls by roughly the factor $\sqrt{10}$ predicted by $[nI(\theta_*)]^{-1/2}$. Two features of these curves also warn us about computation: for small n the negative log-likelihood has several local minima, and its shape away from the well is far from quadratic. Numerical optimization is required, and it must be used with care.

5.2 Univariate Optimization Algorithms

Every algorithm in this section **minimizes** the negative log-likelihood

$$\mathcal{L}(\theta) = -\ell(\theta), \tag{5.12}$$

so that the quantity driving each step is its derivative,

$$g(\theta) = \mathcal{L}'(\theta) = -\ell'(\theta), \tag{5.13}$$

and the quantity measuring curvature is $g'(\theta) = \mathcal{L}''(\theta) = -\ell''(\theta)$. For iid data $\mathcal{L}$ is a sum of per-observation terms, $\mathcal{L}(\theta) = \sum_i \mathcal{L}_i(\theta)$, a decomposition that matters only in the section on stochastic gradients. Maximizing ℓ

and minimizing $-\ell$ are the same problem, but stating it as a minimization matches what `optim`, `nlm` and `optimize` expect, and it is the form the multivariate and gradient-based methods later in the chapter use as well. Note that g' is exactly the observed information $J(\theta)$ of the previous section, so the curvature an optimizer computes on its way to the minimum is the same curvature that supplies the standard error.

5.2.1 The Running Example: Zero-Truncated Poisson Counts

All the methods below are illustrated on one model, so that their pictures and their iterate sequences can be compared directly.

Counts are often recorded only when positive: a study of hospital stays sees admitted patients, a survey of litters sees litters that exist. If $Y \sim \text{Poisson}(\lambda)$ and we observe $X = Y \mid Y > 0$,

$$P(x \mid \lambda) = \frac{e^{-\lambda}\lambda^x}{x!} \cdot \frac{1}{1 - e^{-\lambda}}, \qquad x = 1, 2, \ldots . \tag{5.14}$$

Dropping the factorial terms, which do not involve λ, and dividing by n gives the negative log-likelihood **per observation**,

$$\mathcal{L}(\lambda) = -\overline{x} \log \lambda + \lambda + \log\left(1 - e^{-\lambda}\right), \tag{5.15}$$

whose derivatives are,

$$g(\lambda) = -\ell'(\lambda) = -\frac{\overline{x}}{\lambda} + \frac{1}{1 - e^{-\lambda}}, \qquad g'(\lambda) = -\ell''(\lambda) = \frac{\overline{x}}{\lambda^2} - \frac{e^{-\lambda}}{\left(1 - e^{-\lambda}\right)^2}. \tag{5.16}$$

Two features of this model make it a convenient running example. First, the data enter only through the sample mean $\overline{x}$, so the total negative log-likelihood is exactly n times the per-observation one and the two have the same minimizer; the figures below therefore plot the per-observation version, which does not grow with n, while the standard error at the end uses the total. Second, since $E(X) = \lambda/(1 - e^{-\lambda})$, substituting $E(X)$ for $\overline{x}$ in g' gives the expected information per observation in closed form,

$$I(\lambda) = \frac{1}{\lambda\left(1 - e^{-\lambda}\right)} - \frac{e^{-\lambda}}{\left(1 - e^{-\lambda}\right)^2}, \tag{5.17}$$

which is what Fisher scoring needs. The equation $g(\lambda) = 0$ has no closed-form root, but $\mathcal{L}$ is unimodal — an ideal problem on which to compare all the methods against a single known answer.

```
set.seed(2024)          # the same data underlies every figure in this section

## Negative log-likelihood per observation and its derivatives: nll is L(lambda),
## g is the gradient L'(lambda), dg is the curvature L''(lambda). The data enter
## only through xbar, so the total L is length(xp) times these.
nll <- function(l, xb = xbar) -xb * log(l) + l + log(1 - exp(-l))    # L(lambda)
g   <- function(l, xb = xbar) -xb / l + 1 / (1 - exp(-l))            # gradient  L'
dg  <- function(l, xb = xbar)  xb / l^2 - exp(-l) / (1 - exp(-l))^2  # curvature L''

## expected information per observation, and the fixed-point map used below
info <- function(l) 1 / (l * (1 - exp(-l))) - exp(-l) / (1 - exp(-l))^2
```

```r
fp_map <- function(l, xb = xbar) xb * (1 - exp(-l))

## simulated data: Poisson counts with the zeros discarded
x  <- rpois(100, lambda = 1.5)
xp <- x[x > 0]
xbar <- mean(xp)

lhat <- uniroot(g, c(0.2, 20))$root       # the answer every method must find

c(n_observed = length(xp), sample_mean = xbar, true_lambda = 1.5, mle = lhat)
```

```
 n_observed sample_mean true_lambda         mle
  78.000000    1.897436    1.500000    1.454236
```

The sample mean overshoots 1.5, because the zeros that would have pulled it down were never recorded; the MLE corrects for exactly that.

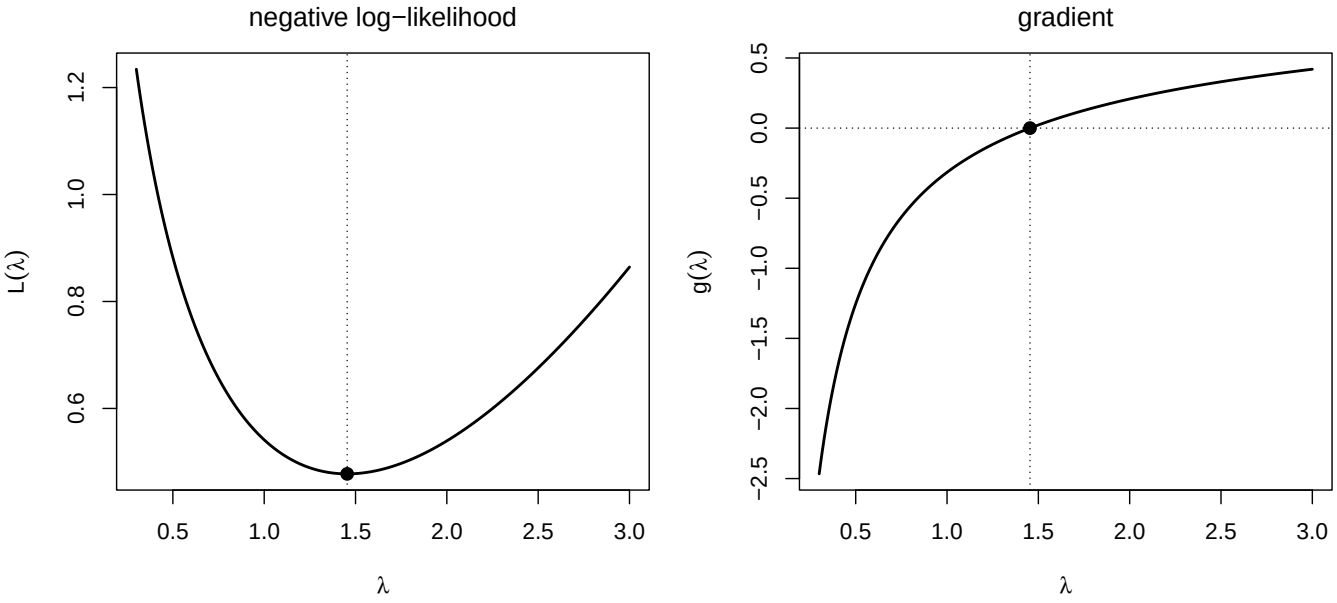


Figure 5.5: Sets up the running example of a zero-truncated Poisson model by showing the negative log-likelihood that every method in this section minimizes, together with its gradient, whose root is the minimizer. Negative log-likelihood per observation and its gradient $g(\lambda)$ for the zero-truncated Poisson model: the minimum of $\mathcal{L}$ sits exactly where g crosses zero.

The minimum of $\mathcal{L}$ and the zero crossing of g are the same point, $\hat{\lambda} \approx 1.454$. Since g is *increasing* through that crossing — the mirror image of the decreasing score $\ell' = -g$ — the curvature $g'(\hat{\lambda})$ is positive, as it must be at a minimum.

5.2.2 Three Equivalent Formulations

The same task can be posed three ways, and the algorithms below attack different versions of it:

- minimize $\mathcal{L}(\theta) = -\ell(\theta)$ directly, using function values only;
- find a root of $g(\theta) = -\ell'(\theta)$;
- rearrange that root equation into a fixed point $\theta = \varphi(\theta)$.

Root-finders are usually faster because they use derivative information, but they locate any stationary point, including maxima and saddle points. Direct minimizers need only function values and always move downhill, but converge more slowly. Which to prefer depends on whether derivatives are available and whether the function is smooth.

Each method below is illustrated with a schematic showing its first few iterations on the truncated-Poisson `nll` just defined. The minimizer $\widehat{\lambda}$ is marked in every figure.

5.2.3 Fixed-Point (Simple) Iteration

Rearrange the stationarity equation $g(\theta) = -\ell'(\theta) = 0$ algebraically into the form $\theta = \varphi(\theta)$ and iterate $\theta_{k+1} = \varphi(\theta_k)$. For the truncated Poisson, $-\overline{x}/\lambda + 1/(1 - e^{-\lambda}) = 0$ rearranges to

$$\lambda = \overline{x}\,(1 - e^{-\lambda}) = \varphi(\lambda), \qquad \varphi'(\lambda) = \overline{x}e^{-\lambda}. \tag{5.18}$$

If φ maps an interval into itself and $|\varphi'(\theta^*)| < 1$ at the fixed point, the mean value theorem gives

$$|\theta_{k+1} - \theta^*| = |\varphi(\theta_k) - \varphi(\theta^*)| \le |\varphi'(\xi)||\theta_k - \theta^*|, \tag{5.19}$$

so the error contracts by roughly the constant factor $|\varphi'(\theta^*)|$ at each step. This is **linear convergence**: a fixed number of digits is gained per iteration, and the closer $|g'|$ is to 1 the slower the progress.

```
fixed_point <- function(g, x0, tol = 1e-10, maxit = 500) {
  for (k in 1:maxit) {
    x1 <- g(x0)
    if (abs(x1 - x0) < tol) return(c(root = x1, iter = k))
    x0 <- x1
  }
  stop("not converged")
}
```

The mechanism is easiest to see as a **cobweb**: from λ_k on the diagonal, move vertically to the curve φ to obtain λ_{k+1}, then horizontally back to the diagonal to start the next step. The iterates walk toward the intersection of g with the 45° line, and the shallower φ is at that crossing — the smaller $|\varphi'|$ — the larger each stride toward it.

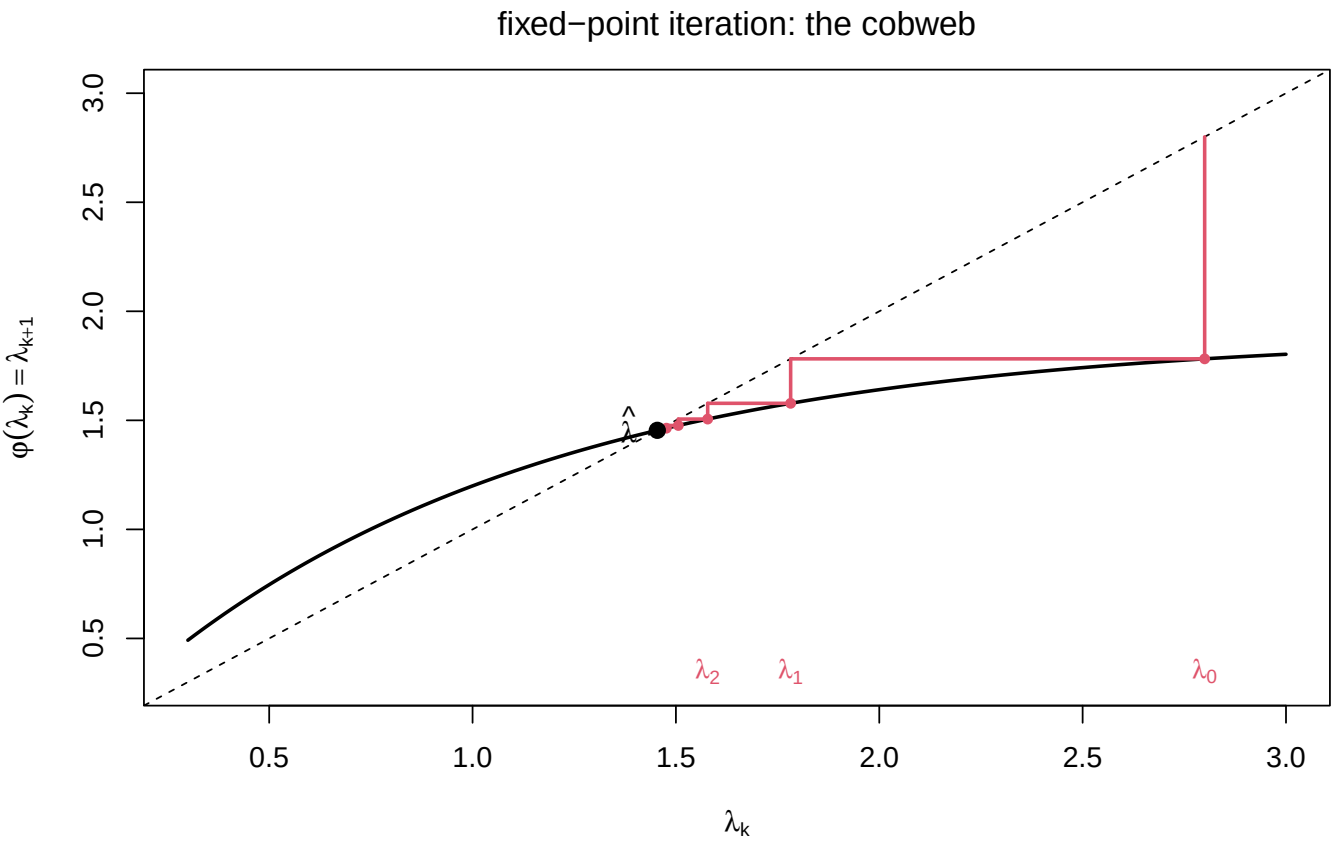


Figure 5.6: Illustrates fixed-point iteration, the simplest scheme for solving the stationarity equation, and how its iterates converge to the solution. Fixed-point iteration as a cobweb diagram: iterates walk toward the intersection of φ with the diagonal.

The method needs no derivatives, but the rearrangement must be found by hand, and different rearrangements of the same equation can converge at different speeds or diverge outright. Geometrically, a rearrangement with $|\varphi'| > 1$ at the crossing produces a cobweb that spirals outward instead of inward.

5.2.4 Bisection

If g is continuous and $g(a)$ and $g(b)$ have opposite signs, a root lies in $[a, b]$. Evaluate g at the midpoint and retain whichever half still changes sign. For the truncated Poisson $-\ell'$ is negative at small λ and positive at large λ, so any bracket straddling $\hat{\lambda}$ works.

```
bisection <- function(d1, a, b, tol = 1e-10) {          # d1 is nll'
  if (d1(a) * d1(b) > 0) stop("endpoints do not bracket a root")
  k <- 0
  while (b - a > tol) {
    m <- (a + b) / 2
    if (d1(a) * d1(m) <= 0) b <- m else a <- m
    k <- k + 1
  }
```

```
  c(root = (a + b) / 2, iter = k)
}
```

The schematic below draws the bracket at each iteration as a bar beneath the curve of g. The dot is the midpoint, and the dotted line carries it up to the curve, where its sign decides which half survives.

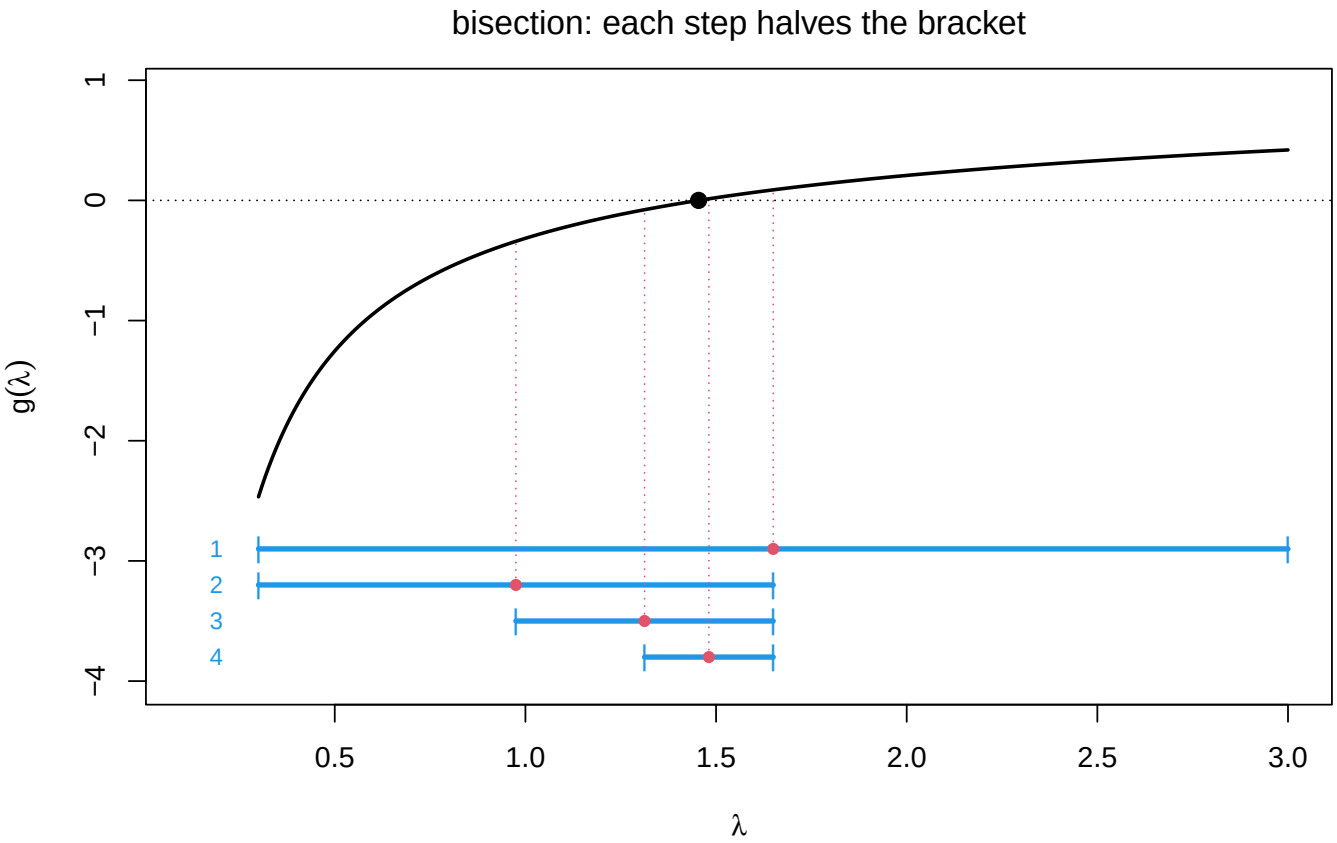


Figure 5.7: Illustrates bisection, a slow but robust root-finding scheme that only needs an interval known to contain the root. Bisection on the gradient g of the negative log-likelihood: each iteration halves the bracket that contains the root.

The bracket halves at every step, so after k iterations the error is at most $(b-a)/2^k$ and the iteration count is known in advance: about $\log_2\{(b-a)/\text{tol}\}$, regardless of the shape of g. Bisection cannot diverge, which makes it the natural fallback when a faster method misbehaves. R's `uniroot` uses Brent's method, which retains the bracketing guarantee while interpolating for speed.

5.2.5 Golden Section Search

Golden section minimizes $\mathcal{L}$ directly, using **no derivative at all**, which makes it the method of choice when ℓ is not differentiable or its derivative is unreliable. Keep a bracket $[x_l, x_u]$ containing the minimum of a unimodal $\mathcal{L}$, with two interior points placed at the golden fraction $\kappa = (3-\sqrt{5})/2 \approx 0.382$ of the way in from each end.

Comparing the two interior values discards one end, and the ratio is chosen precisely so that the surviving interior point is correctly positioned for the next iteration: one new evaluation per step, with the bracket shrinking by the factor 0.618.

```
golden <- function(f, brack.int, eps = 1e-4, ...) {
  kap <- (3 - sqrt(5)) / 2
  xl <- min(brack.int); xu <- max(brack.int)
  tmp <- kap * (xu - xl)
  xmu <- xu - tmp; xml <- xl + tmp
  fl <- f(xml, ...); fu <- f(xmu, ...)
  while (abs(xu - xl) > (1e-5 + abs(xl)) * eps) {
    if (fl < fu) {
      xu <- xmu; xmu <- xml; fu <- fl
      xml <- xl + kap * (xu - xl); fl <- f(xml, ...)
    } else {
      xl <- xml; xml <- xmu; fl <- fu
      xmu <- xu - kap * (xu - xl); fu <- f(xmu, ...)
    }
  }
  if (fl < fu) xml else xmu
}
```

The schematic makes the reuse visible. Each bar is the current bracket with its two interior points; the point that survives into the next iteration is drawn in blue and sits at exactly the right position there, so only the red point has to be evaluated afresh.

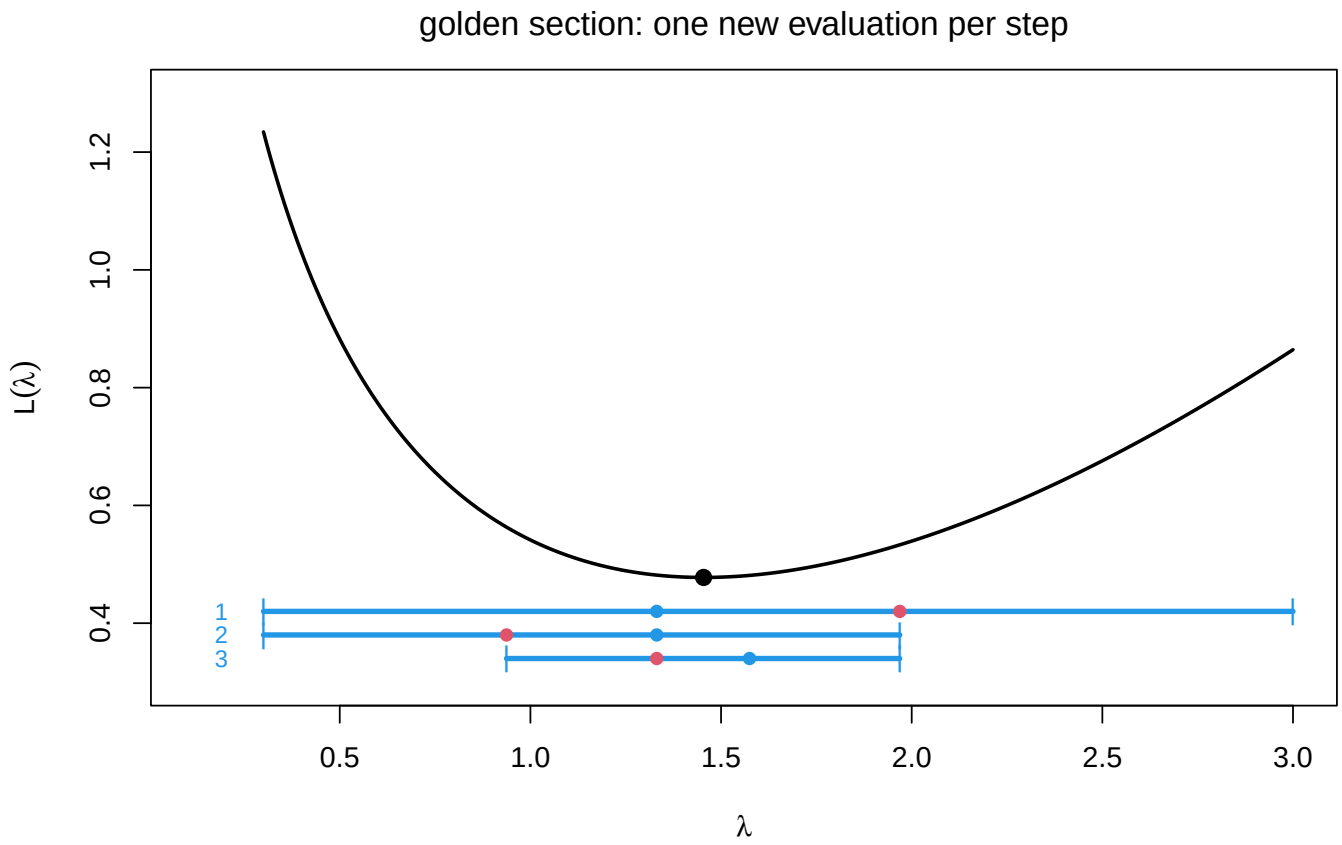


Figure 5.8: Illustrates golden section search, a derivative-free method for optimizing a one-dimensional function, and why it needs only one new function evaluation per step. Golden section search: the interior point that survives (blue) is already correctly positioned for the next iteration, so only one new point (red) is evaluated per step.

The first test is the negative log-likelihood of the classical genetic linkage model, with cell counts $(1997, 906, 904, 32)$ and probabilities $((2+\theta)/4, (1-\theta)/4, (1-\theta)/4, \theta/4)$; it is smooth, so any method would work. The second, $f(x) = |x - 10|$, is not differentiable at its minimum, so no root-finder applies — golden section handles it without modification.

```
f1 <- function(theta) -1997 * log(2 + theta) - 1810 * log(1 - theta) - 32 * log(theta)
f2 <- function(x) abs(x - 10)

par(mfrow = c(1, 2), mar = c(4, 4.5, 2.5, 1))
curve(f1, 0.001, 0.999, n = 400, lwd = 2, xlab = TeX(r"($\theta$)"), ylab =
 ↪  TeX(r"($L(\theta)$)"),
      main = TeX(r"(genetic linkage model)"))
abline(v = golden(f1, c(0.001, 0.999)), col = 2, lty = 2, lwd = 2)

curve(f2, 0, 20, n = 400, lwd = 2, xlab = TeX(r"($x$)"), ylab = TeX(r"($|x - 10|$)"),
      main = TeX(r"(not differentiable at the minimum)"))
abline(v = golden(f2, c(0, 20)), col = 2, lty = 2, lwd = 2)
```

```r
c(linkage = golden(f1, c(0.001, 0.999)), abs_value = golden(f2, c(0, 20)))
```

```
   linkage   abs_value
0.03571247 10.00009645
```

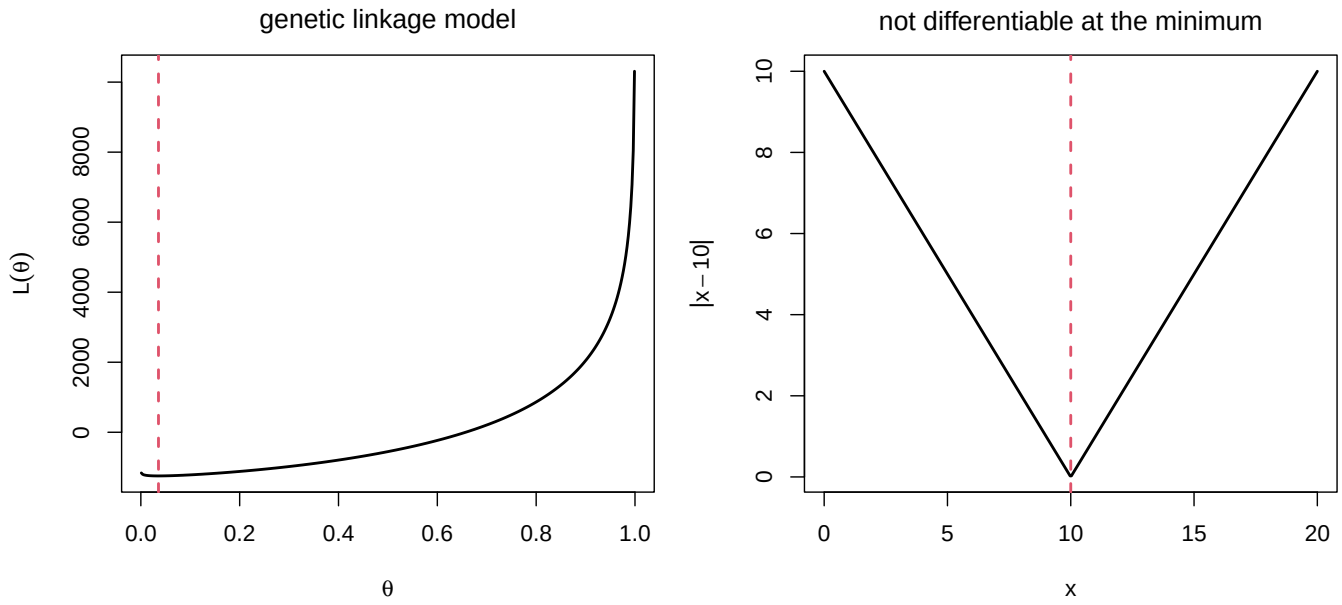


Figure 5.9: Tests golden section search on a smooth likelihood and on a non-differentiable function to show that the method needs no derivatives. Golden section search applied to a smooth genetic-linkage log-likelihood (left) and to a non-differentiable function $|x - 10|$ (right), with the located minimizer marked in red.

5.2.6 Newton–Raphson

5.2.6.1 Derivation

Approximate g by its tangent line at θ_k and solve for where the tangent crosses zero:

$$g(\theta) \approx g(\theta_k) + g'(\theta_k)(\theta - \theta_k) = 0 \implies \theta_{k+1} = \theta_k - \frac{g(\theta_k)}{g'(\theta_k)} = \theta_k - \frac{-\ell'(\theta_k)}{-\ell''(\theta_k)}. \tag{5.20}$$

The two minus signs cancel, so this is numerically the same step a score-based derivation would give; writing it this way keeps the whole section in minimization form. Equivalently, replace $\mathcal{L}$ by its second-order Taylor expansion at θ_k and jump to the minimum of that parabola. For an exactly quadratic $\mathcal{L}(\theta) = a(\theta - u)^2 + C$ with $a > 0$ we have $g = 2a(\theta - u)$ and $g' = 2a$, so

$$\theta_{k+1} = \theta_k - \frac{2a(\theta_k - u)}{2a} = u, \tag{5.21}$$

in one step from any starting value. The normal-mean example of the previous section is exactly of this form: Newton converges immediately.

The left panel below traces three steps. At each iterate, the tangent to g is followed to the horizontal axis, and where it lands is the next iterate. As the curve straightens near the root, the tangent becomes an excellent approximation and the steps land almost exactly on the answer.

The right panel contrasts one Newton step with one step of the method of scoring from the same point. Newton uses the actual slope $g'(\lambda_k)$, the observed information; scoring uses $I(\lambda_k)$, the same curvature averaged over hypothetical datasets. The two lines differ away from the root and coincide as λ_k approaches it.

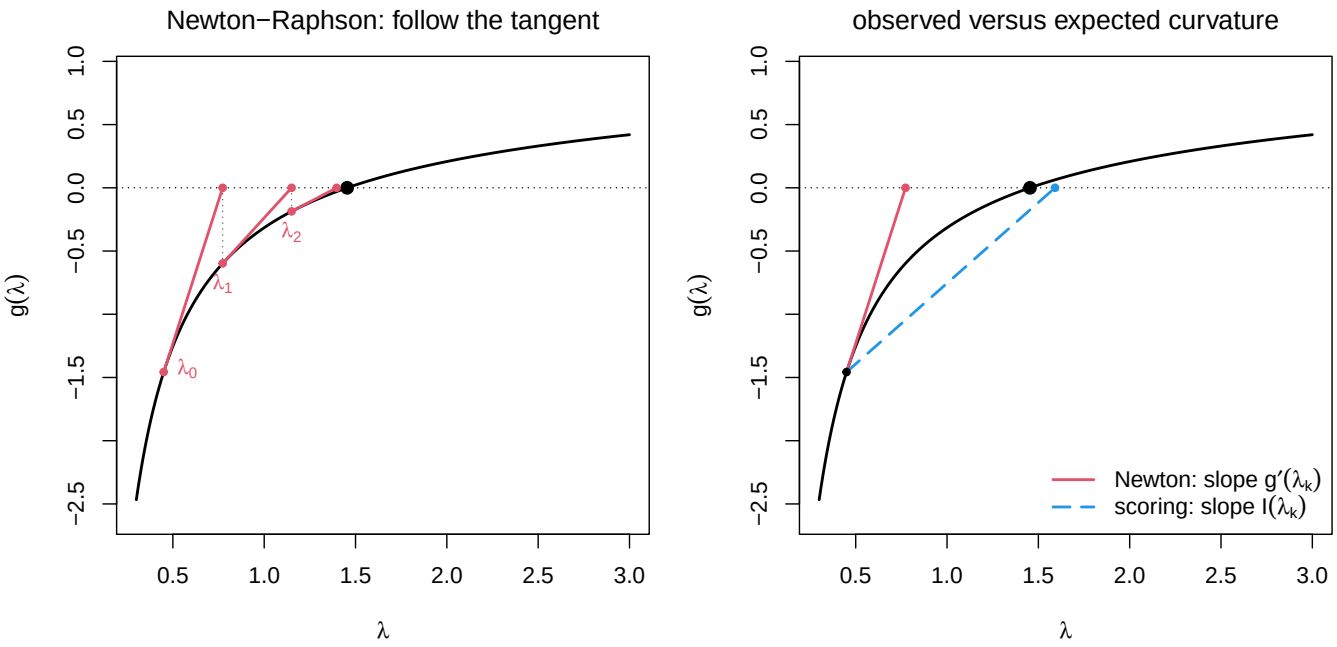


Figure 5.10: Illustrates Newton-Raphson, which replaces the gradient of the negative log-likelihood by its tangent line at each step, and contrasts it with Fisher scoring, which uses the expected rather than the observed curvature. Newton-Raphson following the tangent to $-\ell'$ (left), and one Newton step compared with one Fisher-scoring step from the same point (right).

5.2.7 Advantage: quadratic convergence

Let $e_k = \theta_k - \hat{\theta}$ and expand $g(\hat{\theta}) = 0$ about θ_k:

$$0 = g(\theta_k) - g'(\theta_k)e_k + \tfrac{1}{2}g''(\xi)e_k^2 \implies e_{k+1} = e_k - \frac{g(\theta_k)}{g'(\theta_k)} = \frac{g''(\xi)}{2\,g'(\theta_k)}\,e_k^2. \tag{5.22}$$

The error is squared at each step, so the number of correct digits roughly doubles. Two conditions appear in the constant: $g'(\hat{\theta}) \neq 0$, meaning the minimum is a genuine bowl and not a flat spot, and g'' bounded near the root. Asymptotic normality says that negative log-likelihoods are approximately quadratic near the MLE, which is why Newton is so effective there.

```
newton <- function(d1, d2, x0, tol = 1e-10, maxit = 100, trace = FALSE) {
  for (k in 1:maxit) {                                  # d1 = nll', d2 = nll''
    x1 <- x0 - d1(x0) / d2(x0)
    if (trace) cat(sprintf("iter %2d: %.10f\n", k, x1))
    if (!is.finite(x1)) stop("diverged")
    if (abs(x1 - x0) < tol) return(c(root = x1, iter = k))
    x0 <- x1
  }
  stop("not converged")
}
```

5.2.8 A Classic Illustration: Newton's Method for $\sqrt{a}$

Derivation. To find $\sqrt{a}$, take $f(x) = x^2 - a$, whose positive root is $\sqrt{a}$. Then $f'(x) = 2x$, and the Newton step is,

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} = x_k - \frac{x_k^2 - a}{2x_k} = \frac{1}{2}\left(x_k + \frac{a}{x_k}\right). \tag{5.23}$$

The final form is the Babylonian rule: if x_k is too small then a/x_k is too large, and the root lies between them, so average the pair. Newton's tangent line lands exactly on the midpoint.

The error follows the general quadratic law. Writing $e_k = x_k - \sqrt{a}$,

$$e_{k+1} = \frac{1}{2}\left(x_k + \frac{a}{x_k}\right) - \sqrt{a} = \frac{(x_k - \sqrt{a})^2}{2x_k} = \frac{e_k^2}{2x_k}, \tag{5.24}$$

so the number of correct digits roughly doubles each step.

```
babylonian <- function(a, x0 = 1, iters = 6)
  Reduce(function(x, k) (x + a / x) / 2, 1:iters, init = x0, accumulate = TRUE)

traces <- sapply(c(10, 20, 50), babylonian)
dimnames(traces) <- list(paste("iter", 0:6), sprintf("sqrt(%d)", c(10, 20, 50)))

print(traces, digits = 12)
```

```
           sqrt(10)       sqrt(20)       sqrt(50)
iter 0 1.00000000000  1.00000000000  1.00000000000
iter 1 5.50000000000 10.50000000000 25.50000000000
iter 2 3.65909090909  6.20238095238 13.73039215686
iter 3 3.19600508187  4.71347454529  8.68597437190
iter 4 3.16245562280  4.47831444547  7.22119047433
iter 5 3.16227766518  4.47214021707  7.07262827574
iter 6 3.16227766017  4.47213595500  7.07106798401
```

```
sqrt(c(10, 20, 50))                          # for comparison
```

```
[1] 3.162278 4.472136 7.071068
```

```r
signif(abs(traces - rep(sqrt(c(10, 20, 50)), each = 7)), 3)    # errors
```

```
       sqrt(10) sqrt(20) sqrt(50)
iter 0 2.16e+00 3.47e+00 6.07e+00
iter 1 2.34e+00 6.03e+00 1.84e+01
iter 2 4.97e-01 1.73e+00 6.66e+00
iter 3 3.37e-02 2.41e-01 1.61e+00
iter 4 1.78e-04 6.18e-03 1.50e-01
iter 5 5.01e-09 4.26e-06 1.56e-03
iter 6 4.44e-16 2.03e-12 1.72e-07
```

Starting from $x_0 = 1$ in every case, the first two steps do the coarse work — the larger a is, the further the opening leap — and from the third step on the error squares: roughly 10^{-1}, then 10^{-3}, then 10^{-6}, then machine precision. Six iterations suffice for all three, though $\sqrt{50}$ spends one more step than $\sqrt{10}$ recovering from the poor start.

5.2.9 Problem: divergence

Newton uses only local information, so the quadratic model can be badly wrong away from the maximum. Three failures occur:

- **wrong curvature** — where $g' < 0$ the local model is an upside-down bowl, and the step heads toward a maximum or off to infinity;
- **overshooting** — where g is nearly flat, $g'(\theta_k) \approx 0$ produces an enormous step, possibly outside the parameter space;
- **multiple roots** — the iteration converges to whichever stationary point it happens to reach, which need not be the global minimum.

The single-observation Cauchy likelihood is the standard illustration. With one observation at zero, $\mathcal{L}(\theta) = -\ell(\theta) = \log(1 + \theta^2)$ and $g(\theta) = -\ell'(\theta) = 2\theta/(1 + \theta^2)$, whose only root is $\theta = 0$. That derivative rises, turns over at $\theta = 1$, and decays to zero in the tails — exactly the flat region that defeats Newton.

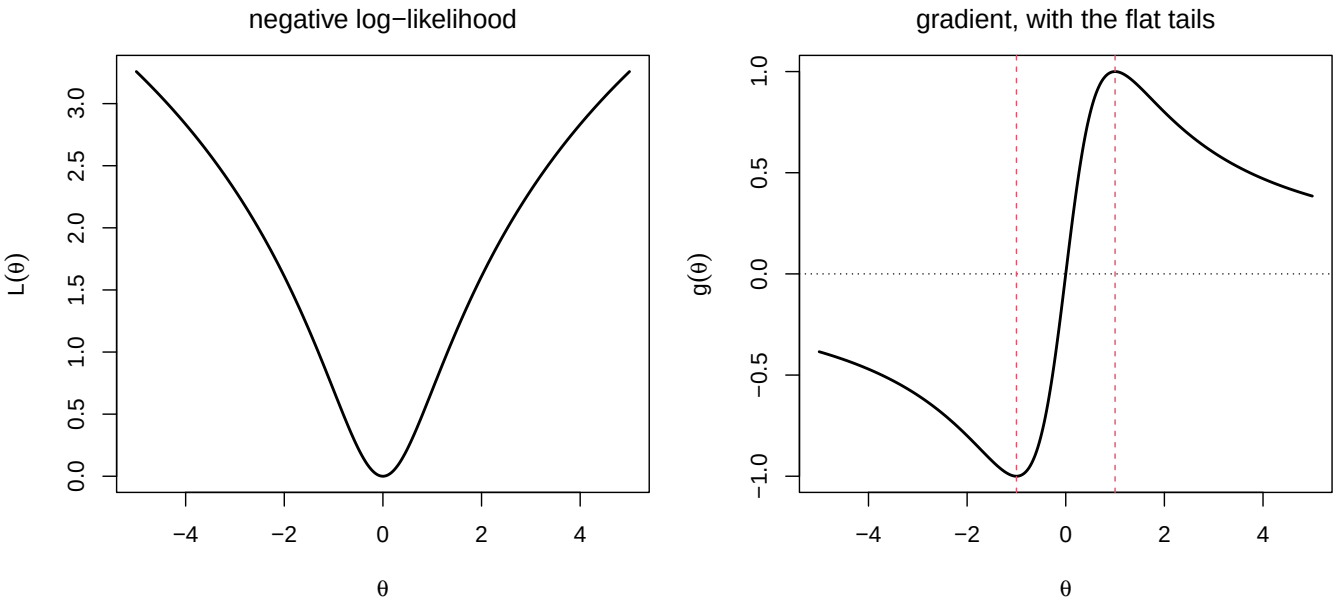


Figure 5.11: Explains why Newton-Raphson can fail on the Cauchy likelihood by showing the shape of the contribution of a single observation. Single-observation Cauchy negative log-likelihood and its gradient: the gradient flattens in the tails, the region that defeats Newton-Raphson.

Tracing the iterates from three starting values shows the boundary at $|\theta_0| = 1$, where g turns over:

```
newton_trace <- function(d1, d2, x0, iters = 8) {
  path <- numeric(iters + 1); path[1] <- x0
  for (k in 1:iters) {
    x0 <- x0 - d1(x0) / d2(x0)
    path[k + 1] <- if (is.finite(x0)) x0 else NA
  }
  path
}

starts <- c(0.5, 0.8, 1.2)
traces <- vapply(starts, function(v) newton_trace(g_c, dg_c, v), numeric(9))
colnames(traces) <- sprintf("start = %.1f", starts)
rownames(traces) <- paste("iter", 0:8)
signif(traces, 5)
```

```
       start = 0.5 start = 0.8 start = 1.2
iter 0  5.0000e-01      0.8000      1.2000
iter 1 -3.3333e-01     -2.8444      7.8545
iter 2  8.3333e-02     -6.4912     15.9680
iter 3 -1.1655e-03    -13.2980     32.0620
iter 4  3.1664e-09    -26.7470     64.1860
iter 5  0.0000e+00    -53.5690    128.4000
iter 6  0.0000e+00   -107.1800    256.8200
iter 7  0.0000e+00   -214.3700    513.6500
iter 8  0.0000e+00   -428.7500   1027.3000
```

From $\theta_0 = 0.5$, inside the region where the quadratic model is adequate, Newton converges quadratically to zero.

From $\theta_0 = 0.8$ the first step already overshoots into the tail, and from $\theta_0 = 1.2$ — beyond the inflection point of $-\ell'$ — the iterates run away, each step landing further out where the derivative is flatter still.

```
## Interactive versions of the same three trajectories (requires the animation
## package and a graphics device that can write animations)
library(animation)
ani.options(interval = 1, nmax = 50)
newton.method(FUN = g_c, init = 0.5, rg = c(-1.5, 1.5))
newton.method(FUN = g_c, init = 0.8, rg = c(-8, 1.5))
newton.method(FUN = g_c, init = 1.2, rg = c(-2, 20))
```

5.2.9.1 Shinylive App for Newton-Raphson on the Cauchy Likelihood

There is a shinylive app to show how the Newton-Raphson step diverges on the Cauchy likelihood from a starting value chosen by clicking, and how step halving and a robustified Hessian each repair it.

5.2.10 Univariate Fisher Scoring

Newton's step divides by the *observed* curvature $g'(\theta_k) = -\ell''(\theta_k)$, computed from the data at hand, which can be negative — or nearly zero — far from the minimum. Replacing it by the *expected* curvature $nI(\theta_k)$ gives the **method of scoring**:

$$\theta_{k+1} = \theta_k - \frac{g(\theta_k)}{nI(\theta_k)} = \theta_k + \frac{\ell'(\theta_k)}{nI(\theta_k)}. \tag{5.25}$$

Because $nI(\theta) > 0$ for every θ by construction, the denominator never changes sign: every step of scoring is a descent step on $\mathcal{L}$, even at a point where Newton's step would point uphill or blow up. Near the minimum $nI(\widehat{\theta}) \approx g'(\widehat{\theta})$, so scoring and Newton coincide and both converge quickly; the two methods differ only in how they behave away from the root.

```
fisher_scoring <- function(d1, nI, x0, tol = 1e-10, maxit = 100) {
  for (k in 1:maxit) {                              # d1 = nll', nI = expected info
    x1 <- x0 - d1(x0) / nI(x0)
    if (abs(x1 - x0) < tol) return(c(root = x1, iter = k))
    x0 <- x1
  }
  stop("not converged")
}

## a poor start: at lambda = 3.5 the curvature nll'' is nearly flat
c(newton_step_from_3.5 = 3.5 - g(3.5) / dg(3.5))  # overshoots to an infeasible negative lambda
```

```
newton_step_from_3.5
          -0.4826613
```

```
rbind(newton  = tryCatch(newton(g, dg, 3.5), error = function(e) c(root = NA, iter = NA)),
      scoring = fisher_scoring(g, info, 3.5))
```

```
            root iter
newton        NA   NA
scoring 1.454235    5
```

From this poor start, Newton's step lands on a negative — and hence infeasible — value of λ and never recovers, while scoring's positive denominator lands its very first step near $\lambda \approx 1.64$, already close to the root, and converges normally from there. The price of that safety is that $nI(\theta)$ must be available in closed form; where it is not, Newton with the safeguards below is the fallback.

5.2.11 Safeguards

Three remedies cover most cases. Start from a robust estimate — the sample median for a location model. Apply **step halving**: accept the full Newton step only if it decreases $\mathcal{L}$, otherwise halve it until it does. Or replace the observed curvature by the expected information, giving the **method of scoring**,

$$\theta_{k+1} = \theta_k - \frac{g(\theta_k)}{nI(\theta_k)}, \tag{5.26}$$

in which the divisor is positive by construction, so every step descends. Near the MLE $nI(\hat\theta) \approx g'(\hat\theta)$, so scoring inherits Newton's speed while being far more stable away from the optimum. Its cost is that the expectation must be available in closed form.

```
safe_newton <- function(f, d1, d2, x0, lower = -Inf, upper = Inf,
                        tol = 1e-10, maxit = 100, max_halve = 40) {
  f0 <- f(x0)                                   # f = nll, d1 = nll', d2 = nll''
  for (k in 1:maxit) {
    step <- -d1(x0) / d2(x0)
    if (d2(x0) < 0) step <- -sign(d1(x0))       # wrong curvature: just move downhill
    for (h in 0:max_halve) {                    # halve until feasible and improving
      x1 <- x0 + step / 2^h
      if (x1 > lower && x1 < upper && is.finite(f(x1)) && f(x1) <= f0) break
    }
    if (abs(x1 - x0) < tol) return(c(root = x1, iter = k, halvings = h))
    x0 <- x1; f0 <- f(x1)
  }
  stop("not converged")
}
```

5.2.12 Numerical Derivatives

When g or g' is tedious or error-prone to derive by hand, both can be approximated by finite differences:

$$g(\theta) \approx \frac{\mathcal{L}(\theta+h) - \mathcal{L}(\theta-h)}{2h}, \qquad g'(\theta) \approx \frac{\mathcal{L}(\theta+h) - 2\,\mathcal{L}(\theta) + \mathcal{L}(\theta-h)}{h^2}. \tag{5.27}$$

The step h trades off two sources of error: too large and the approximation suffers **truncation error** from the curvature of $\mathcal{L}$ itself; too small and it suffers **rounding error**, because the numerator subtracts two nearly equal

floating-point numbers. A step of $h \approx \sqrt{\varepsilon}\,(1 + |\theta|) \approx 10^{-8}$ balances the two for the first derivative, and $h \approx \varepsilon^{1/3}$ for the second, where $\varepsilon \approx 2 \times 10^{-16}$ is double-precision machine epsilon.

```
h <- 1e-5
c(analytic = dg(lhat),
  numeric  = (nll(lhat + h) - 2 * nll(lhat) + nll(lhat - h)) / h^2)
```

```
 analytic   numeric
0.4995693 0.4995676
```

The two agree to several digits, which is reassuring but also the point: `nlm()` and `optim(method = "BFGS")` fall back on exactly this kind of finite differencing whenever an analytic gradient is not supplied, and `numDeriv::grad()` / `numDeriv::hessian()` implement more accurate variants (Richardson extrapolation) for when a single finite difference is not precise enough.

5.2.13 Summary of the Methods

method	needs	convergence	fails when
fixed point	a rearrangement $\theta = \varphi(\theta)$	linear, factor $\lvert\varphi'\rvert$	$\lvert\varphi'\rvert \geq 1$
bisection	g and a sign-changing bracket	linear, factor $1/2$	no bracket available
Newton–Raphson	g and g'	quadratic near the root	flat or wrong curvature
method of scoring	g and $I(\theta)$	linear to quadratic	$E[\cdot]$ unavailable
golden section	$\mathcal{L}$ only, and a bracket	linear, factor 0.618	$\mathcal{L}$ not unimodal

5.2.14 Putting the Methods Together on the Truncated Poisson

Each method above was drawn on this model one step at a time. Running them to convergence side by side shows how the iterate sequences themselves differ.

5.2.14.1 The Three Iterations

Each method below returns the whole sequence of iterates together with the value of $\mathcal{L}$ attained, so the trajectories can be compared directly. All three use the `nll`, `g`, `dg` and `info` — that is, $\mathcal{L}$, g, g' and I — defined at the start of this section, with **simple iteration** using the rearrangement $\lambda = \overline{x}(1 - e^{-\lambda}) = \varphi(\lambda)$ derived earlier.

```
## Each returns the iterate sequence together with the nll attained.
iterate_path <- function(step, n_iter, lambda0) {
  lambda <- numeric(n_iter + 1); lambda[1] <- lambda0
  for (i in 1:n_iter) lambda[i + 1] <- step(lambda[i])
  data.frame(lambda = lambda, nll = nll(lambda))
}

nzp.simple.iteration <- function(n_iter, lambda0)
```

```
  iterate_path(function(l) fp_map(l), n_iter, lambda0)

nzp.newton.raphson <- function(n_iter, lambda0)
  iterate_path(function(l) l - g(l) / dg(l), n_iter, lambda0)

nzp.method.of.scoring <- function(n_iter, lambda0)
  iterate_path(function(l) l - g(l) / info(l), n_iter, lambda0)
```

5.2.14.2 Comparison From Several Starting Values

The three methods are run for 15 iterations from a sensible start, a start far below, and a start far above.

```
starts <- c(xbar, 0.1, 100)

trace_tab <- function(fun, n_iter = 15)
  vapply(starts, function(s) fun(n_iter, s)$lambda, numeric(n_iter + 1))

paths <- list(simple = trace_tab(nzp.simple.iteration),
              newton = trace_tab(nzp.newton.raphson),
              scoring = trace_tab(nzp.method.of.scoring))
```

```
Warning in log(l): NaNs produced

Warning in log(1 - exp(-l)): NaNs produced
```

```
### first six iterates from each starting value
for (m in names(paths)) {
  cat("\n", m, "\n")
  tab <- paths[[m]][1:6, ]
  dimnames(tab) <- list(paste("iter", 0:5), sprintf("start = %.4g", starts))
  print(signif(tab, 6))
}
```

```
 simple
       start = 1.897 start = 0.1 start = 100
iter 0       1.89744    0.100000   100.00000
iter 1       1.61291    0.180565     1.89744
iter 2       1.51926    0.313459     1.61291
iter 3       1.48214    0.510573     1.51926
iter 4       1.46643    0.758687     1.48214
iter 5       1.45961    1.008900     1.46643

 newton
       start = 1.897 start = 0.1 start = 100
iter 0       1.89744    0.100000      100.00
iter 1       1.34531    0.194248    -5070.27
iter 2       1.44706    0.366193         NaN
iter 3       1.45420    0.649619         NaN
iter 4       1.45423    1.024460         NaN
iter 5       1.45423    1.337120         NaN

 scoring
```

```
       start = 1.897 start = 0.1 start = 100
iter 0       1.89744     0.10000   100.00000
iter 1       1.46976     1.73860     1.89744
iter 2       1.45426     1.46102     1.46976
iter 3       1.45423     1.45424     1.45426
iter 4       1.45423     1.45423     1.45423
iter 5       1.45423     1.45423     1.45423
```

The pattern is the one the theory predicts. Newton–Raphson takes the fewest iterations from a good start, and its digits lock in from the left as the error is squared each step. Simple iteration converges from every start, including the extreme ones, but slowly and at a steady rate. The method of scoring is intermediate: always descending on $\mathcal{L}$, and nearly as fast as Newton once close.

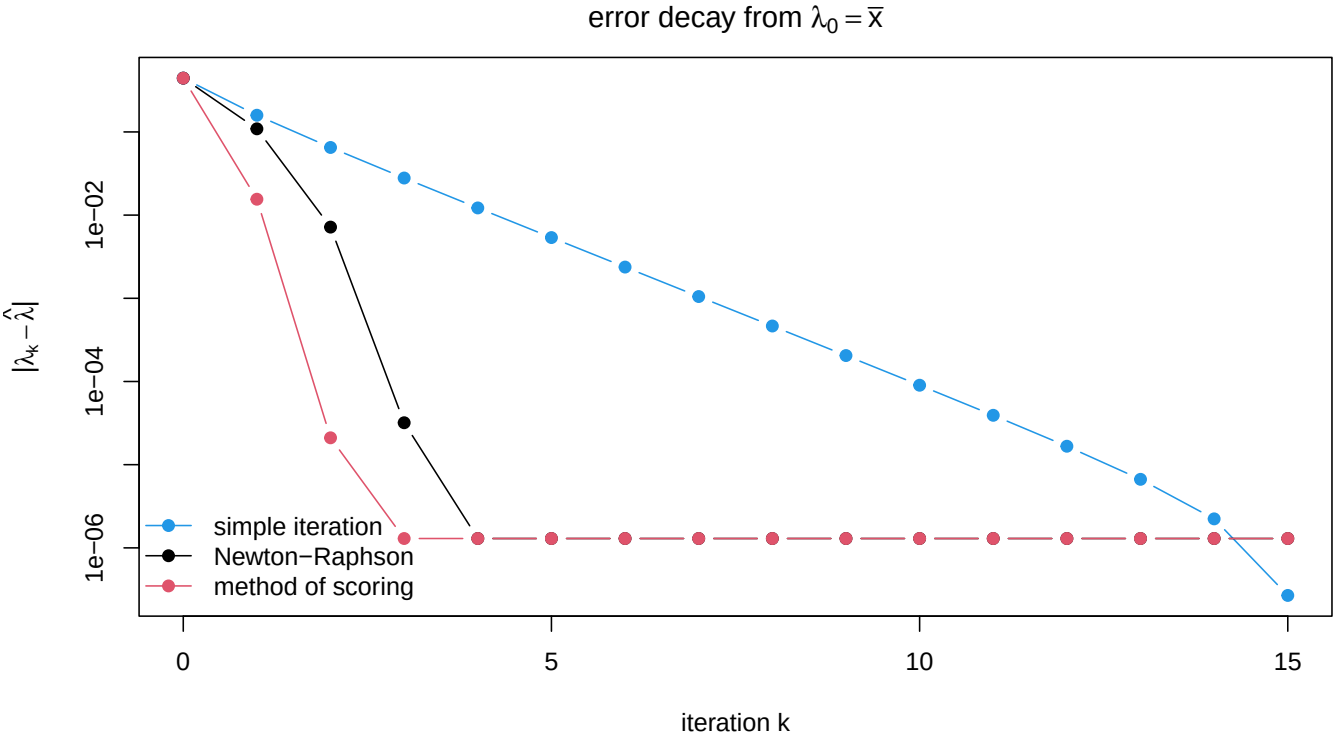


Figure 5.12: Compares the convergence speed of three iterative algorithms on the same problem by showing how quickly each reduces its error. Error decay (log scale) of simple iteration, Newton-Raphson, and the method of scoring for the zero-truncated Poisson MLE: Newton's error curve bends downward, reflecting quadratic convergence.

On a log scale a linear method is a straight line; Newton's curve bends downward until it reaches the floor set by machine precision.

```
c(newton      = unname(newton(g, dg, xbar)["root"]),
  fixed_point = unname(fixed_point(fp_map, xbar)["root"]),
  bisection   = unname(bisection(g, 0.1, 20)["root"]),
  golden      = golden(nll, c(0.1, 20), eps = 1e-8),
  uniroot     = uniroot(g, c(0.1, 20), tol = 1e-10)$root,
  optimize    = optimize(nll, c(0.1, 20), tol = 1e-10)$minimum)
```

```
     newton fixed_point   bisection      golden     uniroot    optimize
   1.454235    1.454235    1.454235    1.454235    1.454235    1.454235
```

All six agree, which is the point of running them together: the methods differ in cost and robustness, not in the answer.

5.2.14.3 Standard Error From the Curvature

The optimizer has already produced everything inference needs: the curvature $g'(\hat{\lambda})$ that Newton divided by *is* the observed information per observation, so the total is $J(\hat{\lambda}) = n\, g'(\hat{\lambda})$.

```
n_obs  <- length(xp)
se_obs <- 1 / sqrt(n_obs * dg(lhat))      # observed information
se_exp <- 1 / sqrt(n_obs * info(lhat))        # expected information

c(mle = lhat, se_observed = se_obs, se_expected = se_exp)
```

```
       mle se_observed se_expected
  1.4542358   0.1601972   0.1601971
```

```
lhat + c(-1.96, 1.96) * se_obs                      # 95% Wald interval
```

```
[1] 1.140249 1.768222
```

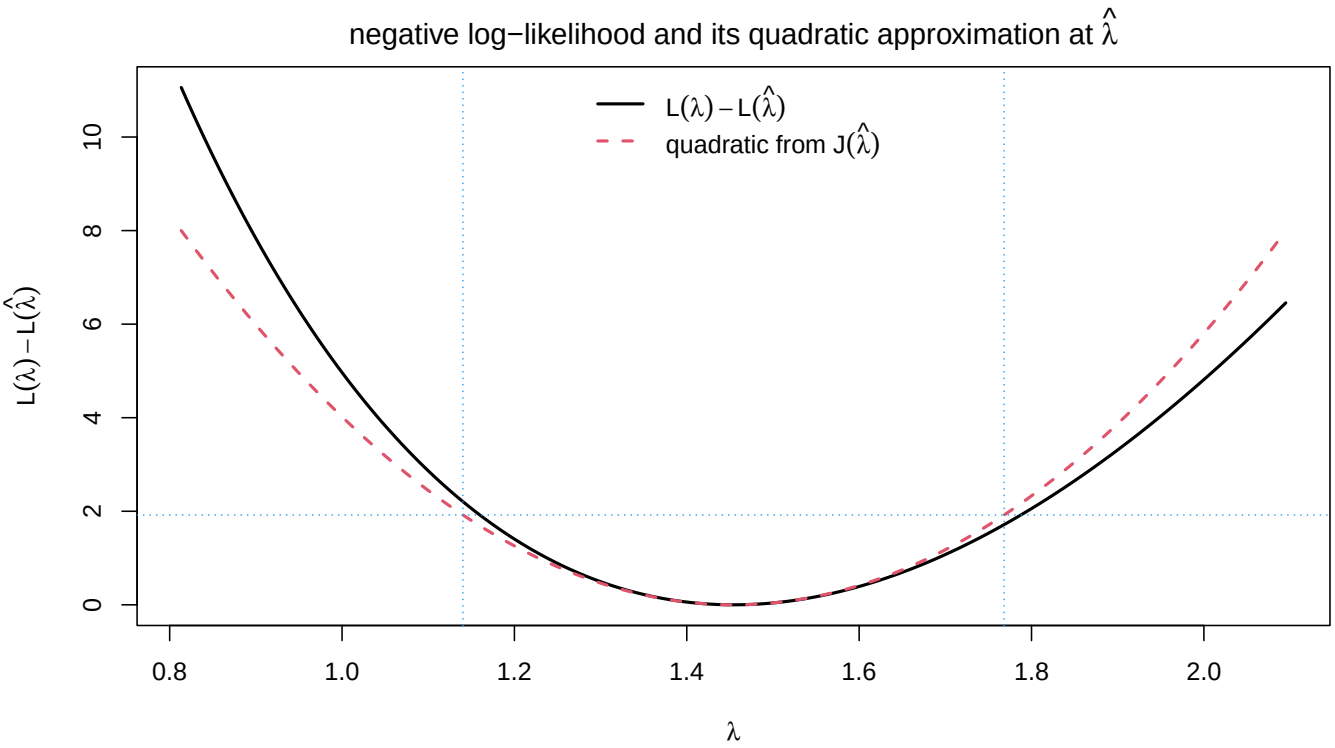


Figure 5.13: Shows how the curvature of the negative log-likelihood at the MLE yields a standard error and a Wald confidence interval, by approximating it with a parabola. Zero-truncated Poisson negative log-likelihood versus its quadratic approximation at $\hat{\lambda}$, with the resulting Wald 95% interval marked.

The parabola implied by the observed information tracks the negative log-likelihood closely over the width of the interval, so the Wald interval is trustworthy here.

5.2.14.4 Evaluating the Naive Estimator

All of this machinery is worth the effort only if the MLE beats the easy alternative. The naive estimator ignores the truncation and reports $\overline{x}$. Simulating many truncated samples gives the sampling distributions of both estimators.

```
mle_nzp <- function(n)
  uniroot(function(l) g(l, xb = mean(n)), c(1e-6, 50), tol = 1e-10)$root

sim_nzp <- function(lambda, n_gen = 100, B = 500)
  vapply(1:B, function(b) {
    y <- rpois(n_gen, lambda); n <- y[y > 0]
    c(naive = mean(n), mle = mle_nzp(n))
  }, c(naive = 0, mle = 0))

for (lambda_true in c(0.5, 1.5)) {
  est <- sim_nzp(lambda_true)
  tab <- t(apply(est, 1, function(e)
    c(bias = mean(e) - lambda_true, sd = sd(e), rmse = sqrt(mean((e - lambda_true)^2)))))
```

```
  cat("\nlambda =", lambda_true, "\n"); print(round(tab, 4))
}
```

```
lambda = 0.5
         bias     sd   rmse
naive  0.7720 0.0906 0.7773
mle   -0.0012 0.1549 0.1547

lambda = 1.5
         bias     sd   rmse
naive  0.4248 0.1220 0.4420
mle   -0.0107 0.1666 0.1668
```

```
est <- sim_nzp(1.5)
par(mfrow = c(1, 2), mar = c(4, 4.5, 2.5, 1))
for (m in c("naive", "mle")) {
  hist(est[m, ], breaks = 25, col = "grey90",
       xlim = range(est), xlab = TeX(r"($\hat{\lambda}$)"),
       main = TeX(sprintf(r"(%s, bias $= %.3f$)", m, mean(est[m, ]) - 1.5)))
  abline(v = 1.5, col = 2, lwd = 2, lty = 2)
}
```

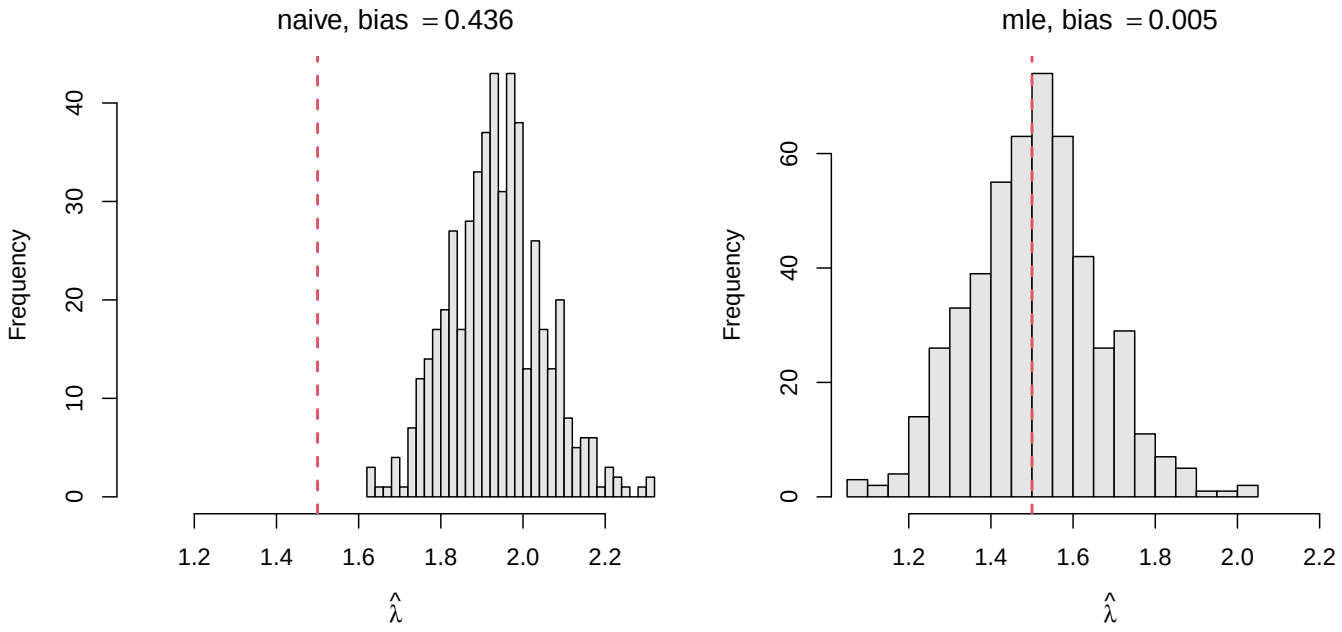


Figure 5.14: Uses simulation to show why the MLE is worth computing: ignoring the truncation gives a biased estimator, whereas the MLE is centred on the truth. Sampling distributions of the naive estimator $\bar{x}$ versus the MLE for a zero-truncated Poisson mean, $\lambda_{true} = 1.5$: the naive estimator is badly biased upward, the MLE is centred on the truth.

The naive estimator is biased upward, and severely so when λ is small, since that is when most of the probability mass sits on the discarded zero. Its bias does not shrink with sample size: it reflects the wrong model, not too little

data. The MLE is centred on the truth with a standard deviation matching the information-based standard error of the previous section. The optimization was the means; the correct estimator was the end.

5.3 Nelder–Mead Method for Multivariate Optimization

Golden section search needs a bracket, which exists only in one dimension. Nelder–Mead generalizes the idea: it keeps $p + 1$ points, a **simplex**, in $\mathbb{R}^p$ (a segment in 1D, a triangle in 2D, a tetrahedron in 3D) and, at each iteration, replaces the worst point by a better one, using **only function values**: no gradient, no Hessian, no bracket. The simplex then crawls downhill, stretching where the objective falls steadily and shrinking around a minimum, so that it adapts its shape to the local contours of $\mathcal{L}$ without ever computing a derivative.

5.3.1 One iteration

Write $f = \mathcal{L}$ for the objective. Order the $p + 1$ vertices so that

$$f(v_1) \leq f(v_2) \leq \cdots \leq f(v_{p+1}), \tag{5.28}$$

so that v_1 is the best and v_{p+1} the worst, and let $c = \frac{1}{p}\sum_{j=1}^{p} v_j$ be the centroid of all vertices except the worst. The worst vertex is reflected through c, and, depending on how good the new point is, the simplex is expanded, contracted or shrunk. The trial points are

$$x_r = c + \alpha\,(c - v_{p+1}), \qquad x_e = c + \gamma\,(x_r - c), \qquad x_c = c + \rho\,(v_{p+1} - c), \tag{5.29}$$

the **reflection**, the **expansion** and the (inside) **contraction**, with coefficients $\alpha = 1$, $\gamma = 2$, $\rho = \frac{1}{2}$ and shrink factor $\sigma = \frac{1}{2}$, the values used by `optim`.

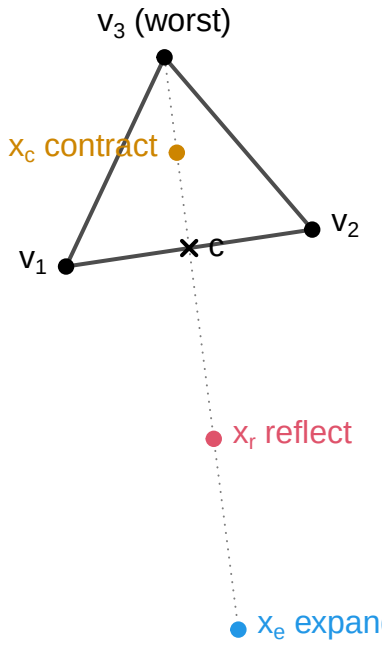


Figure 5.15: Shows the trial points of one Nelder-Mead iteration, which are all on the line through the worst vertex and the centroid of the other two. A triangle with vertices v_1 (best), v_2 and v_3 (worst): the reflection x_r, the expansion x_e beyond it, and the contraction x_c between c and the worst vertex.

The decision rule compares $f(x_r)$ with the current values:

Table 5.1: Lists the actions of one Nelder–Mead iteration according to how good the reflected point is. Decision rule of the Nelder–Mead method.

Outcome of the reflection	Action
$f(x_r) < f(v_1)$, better than the best	Try the **expansion** x_e; keep the better of x_e and x_r in place of v_{p+1}
$f(v_1) \le f(x_r) < f(v_p)$, better than the second worst	Accept x_r in place of v_{p+1}
$f(v_p) \le f(x_r) < f(v_{p+1})$, only slightly better than the worst	**Outside contraction** $x_c = c + \rho(x_r - c)$; accept it if $f(x_c) \le f(x_r)$, otherwise shrink
$f(x_r) \ge f(v_{p+1})$, no better than the worst	**Inside contraction** $x_c = c + \rho(v_{p+1} - c)$; accept it if $f(x_c) < f(v_{p+1})$, otherwise shrink
Contraction failed	**Shrink**: $v_j \leftarrow v_1 + \sigma(v_j - v_1)$ for $j = 2, \dots, p+1$

A typical iteration costs one or two function evaluations; only a shrink costs p more. The simplex needs a starting point and a size: `optim` builds the initial simplex from the starting value θ_0 by moving each coordinate in turn by a step proportional to the scale of the parameter. The iteration stops when the spread of the function values over the simplex, $|f(v_{p+1}) - f(v_1)|$, falls below a tolerance (`reltol`, default 10^{-8} relative), which is a criterion on f rather than on the gradient: it does not certify that a stationary point has been reached.

5.3.2 Implementation

The whole algorithm fits in a few lines. The function below returns the final point and also all the simplices visited, so that the path can be drawn.

```
## Nelder-Mead for f(theta); x0 is the start, step the size of the initial simplex
nelder_mead <- function(f, x0, step = 0.5, maxit = 500, tol = 1e-10,
                        alpha = 1, gamma = 2, rho = 0.5, sigma = 0.5) {
  p <- length(x0)
  V <- rbind(x0, t(sapply(1:p, function(j) x0 + step * (1:p == j))))   # (p+1) x p simplex
  fv <- apply(V, 1, f); nev <- p + 1; hist <- list(V)
  for (it in 1:maxit) {
    o <- order(fv); V <- V[o, , drop = FALSE]; fv <- fv[o]            # v_1 best, v_{p+1} worst
    if (abs(fv[p + 1] - fv[1]) < tol * (abs(fv[1]) + tol)) break      # spread of f over the
     ↪ simplex
    cen <- colMeans(V[1:p, , drop = FALSE])                           # centroid of all but the
↪ worst
    xr <- cen + alpha * (cen - V[p + 1, ]); fr <- f(xr); nev <- nev + 1
    if (fr < fv[1]) {                                                 # better than the best:
     ↪ expand
      xe <- cen + gamma * (xr - cen); fe <- f(xe); nev <- nev + 1
      if (fe < fr) { V[p + 1, ] <- xe; fv[p + 1] <- fe } else { V[p + 1, ] <- xr; fv[p + 1] <-
       ↪ fr }
    } else if (fr < fv[p]) {                                          # better than the second
↪ worst
      V[p + 1, ] <- xr; fv[p + 1] <- fr
    } else {                                                          # poor: contract, outside
↪ or inside
      xc <- if (fr < fv[p + 1]) cen + rho * (xr - cen) else cen + rho * (V[p + 1, ] - cen)
      fc <- f(xc); nev <- nev + 1
      if (fc < min(fr, fv[p + 1])) {
        V[p + 1, ] <- xc; fv[p + 1] <- fc
      } else {                                                        # contraction failed:
↪ shrink
        for (i in 2:(p + 1)) V[i, ] <- V[1, ] + sigma * (V[i, ] - V[1, ])
        fv[2:(p + 1)] <- apply(V[2:(p + 1), , drop = FALSE], 1, f); nev <- nev + p
      }
    }
    hist[[length(hist) + 1]] <- V
  }
  o <- order(fv)
  list(par = V[o[1], ], value = unname(fv[o[1]]), iterations = it, evals = nev, hist = hist)
}
```

5.3.3 Example: the Rosenbrock banana

The Rosenbrock function $f(\theta) = (1-\theta_1)^2 + 100(\theta_2 - \theta_1^2)^2$ has its minimum at $(1,1)$, at the bottom of a long, curved, narrow valley: easy to find, hard to follow. It is the standard test of an optimizer.

```
rosen <- function(v) (1 - v[1])^2 + 100 * (v[2] - v[1]^2)^2       # Rosenbrock "banana", min at
 ↪  (1, 1)
rosen_gr <- function(v) c(-2 * (1 - v[1]) - 400 * v[1] * (v[2] - v[1]^2), 200 * (v[2] -
 ↪  v[1]^2))

nm <- nelder_mead(rosen, c(-1.2, 1))
c(nm$par, f = nm$value, iterations = nm$iterations, evals = nm$evals)
```

```
                                 f   iterations       evals
1.000000e+00 1.000000e+00 1.238327e-20 1.240000e+02 2.350000e+02
```

```
## the same method in optim, and gradient-based methods on the same problem
cmp <- rbind("Nelder-Mead (optim)" = optim(c(-1.2, 1), rosen, method = "Nelder-Mead")$counts,
             "BFGS, numerical gradient" = optim(c(-1.2, 1), rosen, method = "BFGS")$counts,
             "BFGS, analytic gradient" = optim(c(-1.2, 1), rosen, rosen_gr, method =
              ↪  "BFGS")$counts)
cmp
```

```
                         function gradient
Nelder-Mead (optim)           195       NA
BFGS, numerical gradient      120       38
BFGS, analytic gradient       110       43
```

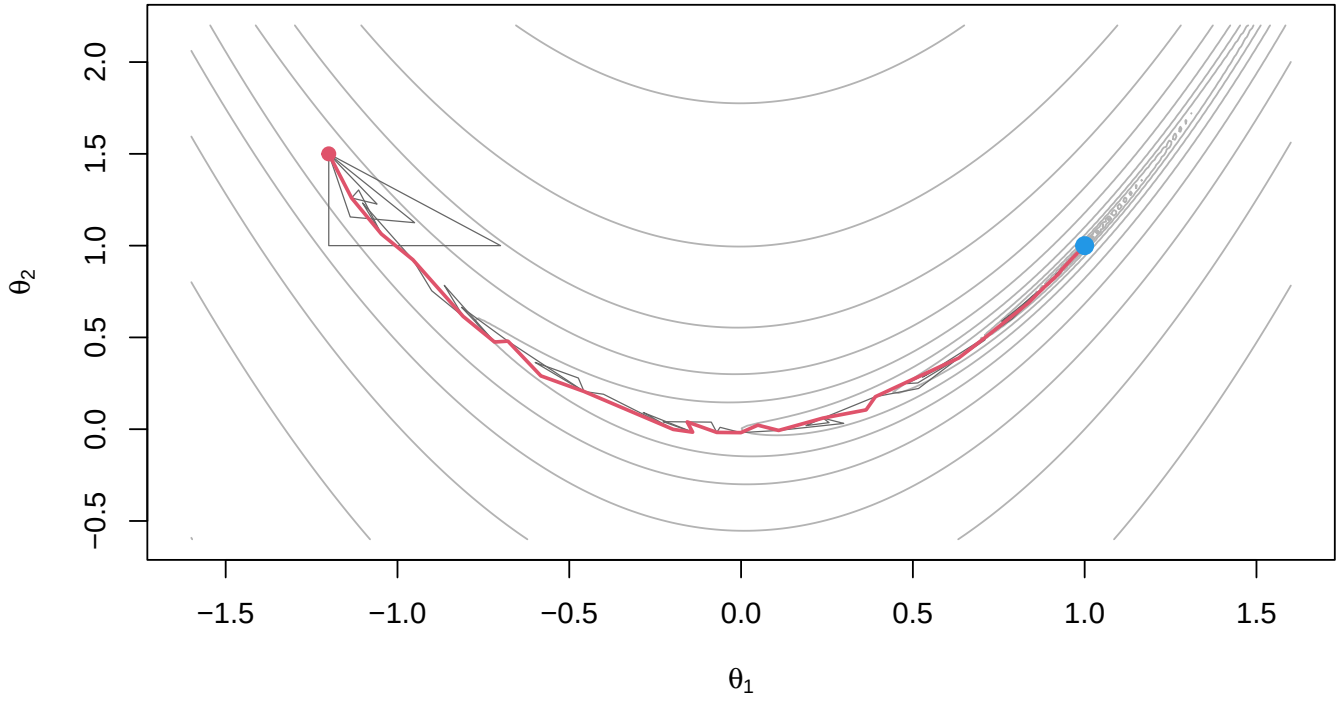


Figure 5.16: Shows how the Nelder-Mead simplex follows the curved valley of the Rosenbrock function using only function values. Contours of the Rosenbrock function with the successive simplices of Nelder-Mead (grey triangles, every second iteration) and the path of the best vertex (red); the method starts at $(-1.2, 1)$ and ends at the minimum $(1, 1)$ (blue).

Starting from $(-1.2, 1)$ the simplex first stretches and tumbles down into the valley, then follows it round the bend, shrinking as it goes, and finally collapses onto $(1, 1)$. The counts in the table compare the cost: Nelder–Mead needs somewhat more function evaluations than BFGS does here (195 against 110 to 120), but no gradient evaluations at all. In two parameters the difference is small; it grows quickly with p, since BFGS needs about the same number of iterations whereas the simplex has more vertices to improve.

5.3.4 Strengths and limits

- **No derivatives.** It needs only f, so it applies to noisy, non-differentiable or awkward objectives, such as those from simulations or a likelihood with an integral that has no usable derivative, and when coding a gradient is not worth the effort.
- **Robust to a poor start.** The simplex is large at first and does not rely on a quadratic model, so it rarely diverges as Newton–Raphson can.
- **Slow and unreliable in high dimension.** The simplex has $p + 1$ vertices to improve, convergence is only sublinear, and beyond about ten parameters it becomes impractical. It can also stall on a ridge or in a valley, giving a point that is not a stationary point; restarting from the returned value, with a new simplex, is the usual remedy.
- **No standard errors.** It produces no curvature. After Nelder–Mead the Hessian must be obtained by numerical differentiation at the solution, as `optim(..., hessian = TRUE)` does.

This makes it, as `optim`'s default method, a reasonable first look at an unfamiliar objective with a few parameters, but a poor choice for a smooth likelihood that is fitted many times or has many parameters, where the gradient-based methods of the next sections are far more efficient.

5.3.5 Shinylive App for the Nelder–Mead Algorithm

There is a shinylive app to show how the Nelder-Mead method finds a minimum without derivatives, by repeatedly reflecting, expanding, contracting, and shrinking a simplex.

5.4 Multivariate Newton-Raphson Optimzation Methods

5.4.1 Gradient and Hessian: A Geometric Illustration

The two objects entering the multivariate Newton step have a direct geometric reading. The negative gradient $-g(\theta)$ is the direction of steepest descent and is perpendicular to the contour of $\mathcal{L}$ through θ; the Hessian describes how that direction curves, and at a strict local minimum $J(\theta) = \nabla g(\theta)$ is positive definite. Near $\hat{\theta}$, $\mathcal{L}$ is well approximated by the quadratic

$$\mathcal{L}(\theta) \approx \mathcal{L}(\hat{\theta}) + \tfrac{1}{2}(\theta - \hat{\theta})^{\top} J(\hat{\theta})(\theta - \hat{\theta}), \tag{5.30}$$

whose contours are ellipses centred at $\hat{\theta}$. Writing the eigendecomposition $J(\hat{\theta}) = Q\Lambda Q^{\top}$, the axes of these ellipses point along the eigenvectors in Q, and the half-length along axis j is proportional to $1/\sqrt{\lambda_j}$: a large eigenvalue means sharp curvature and a short, well-determined direction, while $\lambda_{\max}/\lambda_{\min}$ measures how elongated — and, once the axes are tilted, how correlated the parameters are.

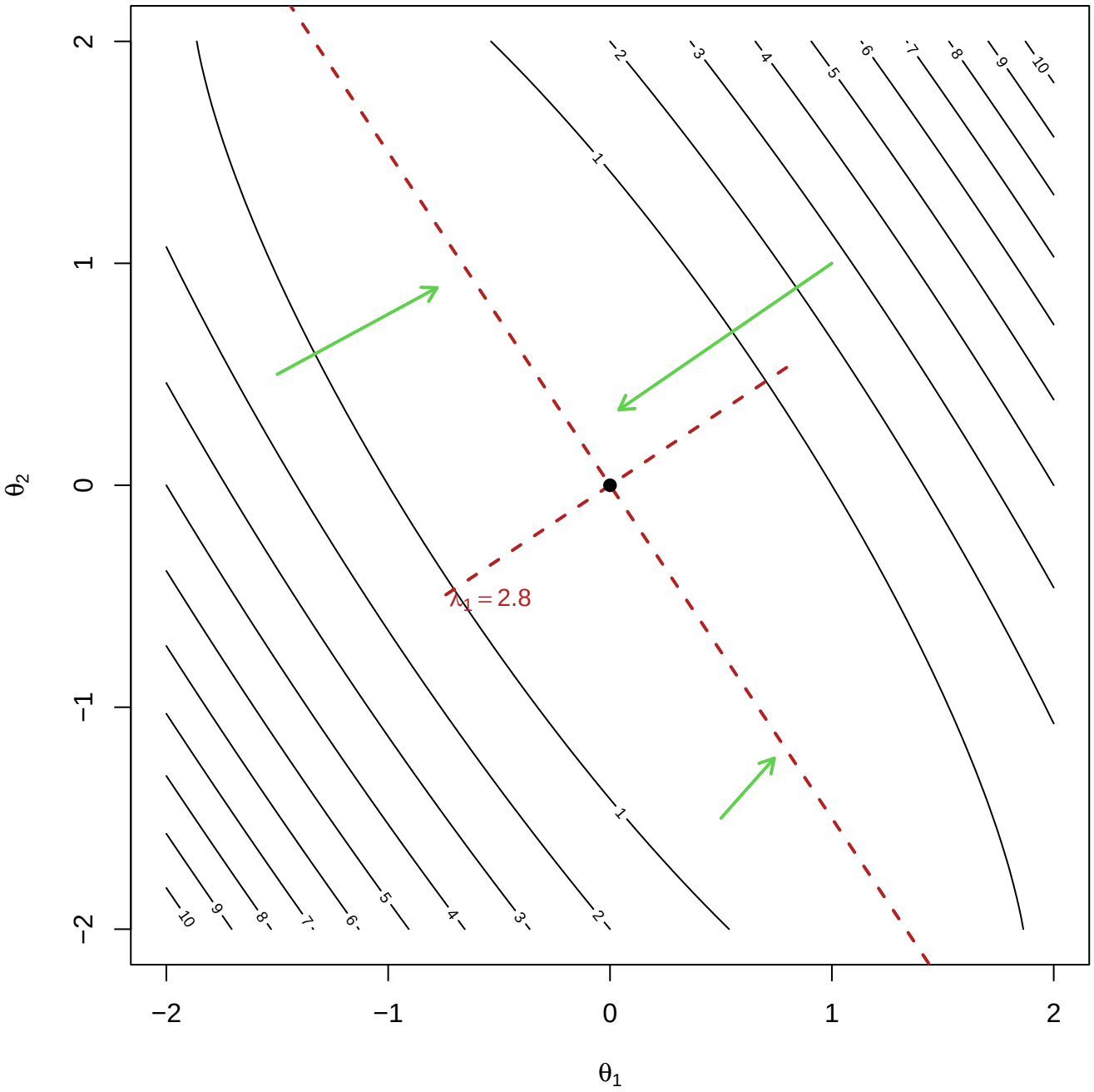


Figure 5.17: Gives a geometric picture of the gradient and the Hessian in two dimensions, explaining why the direction of steepest descent generally does not point at the minimum. Contours of a quadratic negative log-likelihood: steepest-descent arrows (green) run perpendicular to the contours but do not point at the minimum unless the ellipses are circular; the dashed axes are the eigenvectors of $J(\hat{\theta})$, each with half-length $1/\sqrt{\lambda_j}$.

Every arrow in the figure is perpendicular to the contour it starts on and points generally downhill toward the minimum at the origin, but only an arrow started exactly on a principal axis points straight at it. This is why Newton's step, which multiplies the gradient by $J(\hat{\theta})^{-1}$ rather than simply following it, converges directly to $\hat{\theta}$: it rescales and

rotates the descent direction to undo exactly this distortion. The plain gradient-descent methods of a later section have no such correction, and zig-zag across elongated contours like these.

5.4.2 Naive Newton-Raphson Iterations

In the previous section the parameter was a scalar, and the log-likelihood could be inspected by plotting it. Once $\theta \in \mathbb{R}^p$ neither is true: we cannot see the surface, and the notion of "bracketing a root" that underlies bisection and golden section has no direct multivariate analogue. What survives is the local quadratic approximation, and the Newton–Raphson algorithm is its direct consequence.

This section develops the multivariate Newton–Raphson method and its statistical variant, Fisher scoring, applies both to logistic regression, examines interactively how and why the iteration can diverge, and closes with a survey of the derivative-free and gradient-only alternatives.

Write the log-likelihood as $\ell(\theta)$ for $\theta \in \mathbb{R}^p$, and define.

$$g(\theta) = \nabla\mathcal{L}(\theta) = -\frac{\partial \ell(\theta)}{\partial \theta} \quad (p\times 1,\ \text{the gradient}), \qquad \nabla g(\theta) = -\frac{\partial^2 \ell(\theta)}{\partial \theta\, \partial \theta^\top} \quad (p\times p,\ \text{the Hessian of } \mathcal{L}). \tag{5.31}$$

The univariate $g = \mathcal{L}'$ becomes the gradient $g = \nabla\mathcal{L}$ with no change of meaning. As before, the Hessian of $\mathcal{L}$ *is* the **observed information matrix**, so we write $J(\theta) = \nabla g(\theta)$ throughout.

Expand $\mathcal{L}$ to second order about the current iterate $\theta^{(k)}$:

$$\mathcal{L}(\theta) \;\approx\; \mathcal{L}(\theta^{(k)}) + g(\theta^{(k)})^\top(\theta - \theta^{(k)}) + \tfrac{1}{2}\,(\theta - \theta^{(k)})^\top J(\theta^{(k)})\,(\theta - \theta^{(k)}). \tag{5.32}$$

This is a quadratic in θ. If $J(\theta^{(k)})$ is positive definite the quadratic has a unique minimum, found by setting its derivative to zero:

$$g(\theta^{(k)}) + J(\theta^{(k)})(\theta - \theta^{(k)}) = 0. \tag{5.33}$$

Solving for θ gives the **Newton–Raphson update**:

$$\boxed{\theta^{(k+1)} = \theta^{(k)} - J(\theta^{(k)})^{-1} g(\theta^{(k)})}. \tag{5.34}$$

The algorithm therefore replaces the negative log-likelihood by the quadratic that matches its value, slope and curvature at the current point, and jumps to the minimum of that quadratic. Everything that follows — the fast convergence, and the modes of failure — comes from how good or bad that replacement is.

Three remarks are worth making explicit.

The inverse is never formed. Write the update as $\theta^{(k+1)} = \theta^{(k)} + \delta^{(k)}$ where $\delta^{(k)}$ solves the linear system $J(\theta^{(k)})\,\delta = -g(\theta^{(k)})$. In R this is `solve(J, -g)`, not `solve(J) %*% (-g)`. The former is faster and numerically more stable; the latter computes p^2 numbers in order to use p of them.

The step is not the steepest-descent direction. $\delta^{(k)} = -J^{-1}g$ points along $-g$ only when J is a multiple of the identity. The information matrix rescales and rotates the descent direction to account for differing curvature and correlation among the parameters, which is exactly why Newton–Raphson is insensitive to the scaling of the parameters while gradient descent is not.

Positive definiteness is an assumption, not a guarantee. Away from the minimum $J(\theta^{(k)})$ can be indefinite or singular, in which case the "minimum of the quadratic" is a saddle point or does not exist, and $\delta^{(k)}$ may point uphill. Section 5.4.5 examines this.

5.4.2.1 Fisher scoring

The observed information $J(\theta)$ depends on the data. Its expectation under the model,

$$I(\theta) = E_\theta\{J(\theta)\} = -E_\theta\left\{\frac{\partial^2 \ell(\theta)}{\partial\theta\,\partial\theta^\top}\right\}, \tag{5.35}$$

is the **expected (Fisher) information matrix**. Replacing J by I in the update gives the **method of scoring**:

$$\theta^{(k+1)} = \theta^{(k)} - I(\theta^{(k)})^{-1} g(\theta^{(k)}). \tag{5.36}$$

The two algorithms have different iterates but the same fixed point, since both stop where $g(\theta) = 0$. The practical differences:

Table 5.2: Contrasts Newton-Raphson with Fisher scoring, the two choices of curvature matrix in a second-order update, to help decide between them. Newton–Raphson compared with Fisher scoring

	Newton–Raphson	Fisher scoring
Curvature matrix	Observed information $J(\theta)$	Expected information $I(\theta)$
Derivation	Second derivative of $\mathcal{L}$	Expectation of that second derivative
Positive definite?	Only guaranteed near the minimum	Always, wherever the model is identified
Algebra	Can be messy	Often simpler, as data-dependent terms average out
Local convergence	Quadratic	Linear, but typically with a very small rate constant
Stability from poor starts	Less stable	More stable

The guaranteed positive definiteness of $I(\theta)$ is the main practical argument for scoring: the update direction is always a descent direction on $\mathcal{L}$, so the iteration cannot be sent uphill by an ill-behaved Hessian.

For generalised linear models with the **canonical link** the two coincide exactly, because the second derivative of the log-likelihood does not involve the response. Logistic regression uses the canonical logit link, so the example below is simultaneously Newton–Raphson, Fisher scoring, and — as we note at the end of the section — iteratively reweighted least squares.

5.4.3 Convergence properties

Near a minimum $\hat{\theta}$ of $\mathcal{L}$ at which $J(\hat{\theta})$ is positive definite, Newton–Raphson converges **quadratically**: the error satisfies

$$\|\theta^{(k+1)} - \hat{\theta}\| \;\le\; C\,\|\theta^{(k)} - \hat{\theta}\|^2, \tag{5.37}$$

for some constant C depending on the third derivatives. Informally, the number of correct digits doubles at each iteration, which is why five or six iterations are usually enough and why a well-started run appears to converge instantly.

The qualification "near" is essential. The theorem is local: it guarantees nothing about a start far from $\hat{\theta}$, and nothing at all if J fails to be positive definite along the way.

5.4.4 Standard errors as a by-product

At convergence the same matrix that drove the iteration supplies the asymptotic variance. Under the usual regularity conditions,

$$\hat{\theta} \;\dot{\sim}\; N\big(\theta,\; J(\hat{\theta})^{-1}\big)\,, \tag{5.38}$$

so $\mathrm{SE}(\hat{\theta}_j) = \sqrt{[J(\hat{\theta})^{-1}]_{jj}}$. This is a genuine practical advantage of Newton-type methods over derivative-free ones: the curvature information needed for inference has already been computed. Methods that never form a Hessian require a separate numerical differentiation step, which is what `hessian = TRUE` requests from `nlm` and `optim`.

5.4.5 Enhancements to Multivariate Newton-Raphson Methods

Newton–Raphson can diverge even on a well-behaved problem. Far from $\hat{\theta}$ the quadratic approximation is unreliable, and $J_k = J(\theta^{(k)})$ may be nearly singular or not positive definite, so the raw step $\delta^{(k)} = -J_k^{-1} g_k$, with $g_k = g(\theta^{(k)})$, can be enormous or fail to point downhill at all. The standard enhancements all keep the Newton idea and add a guard.

- **Step halving (line search).** Move $\theta^{(k+1)} = \theta^{(k)} + s\,\delta^{(k)}$, starting at $s = 1$ and halving until $\mathcal{L}$ decreases by enough — the **Armijo condition** $\mathcal{L}(\theta^{(k)} + s\,\delta^{(k)}) \le \mathcal{L}(\theta^{(k)}) + c_1 s\, g_k^\top \delta^{(k)}$ with a tiny c_1 (say 10^{-4}). Near the minimum $s = 1$ passes on the first try, so fast local convergence is untouched.

  The condition asks for more than a lower value of $\mathcal{L}$. Written as

  $$\underbrace{\mathcal{L}(\theta^{(k)} + s\,\delta^{(k)}) - \mathcal{L}(\theta^{(k)})}_{\text{actual decrease}} \;\le\; c_1 \cdot \underbrace{s\, g_k^\top \delta^{(k)}}_{\text{decrease predicted by the tangent plane}}, \tag{5.39}$$

  it says that the decrease must be at least a fixed fraction of what the slope promises for a step of that length. (When J_k is positive definite, $g_k^\top \delta^{(k)} = -g_k^\top J_k^{-1} g_k < 0$, so the right-hand side is a genuine decrease.) Asking only for *some* decrease is not enough, because the gains can shrink so fast that the iterates stall at a point that is not a minimum. For $\mathcal{L}(\theta) = \theta^2$, the sequence $\theta^{(k)} = (-1)^k(1 + 2^{-k})$ lowers $\mathcal{L}$ at every step and moves downhill each time, yet it bounces towards ± 1, where the gradient is ± 2, never 0. With $\delta = -g = -2\theta$ the Armijo condition reduces to $s \le 1 - c_1$, and the overshooting steps need $s \to 1$, so they are rejected. Armijo only forbids steps that are too long for the gain they deliver. Starting at $s = 1$ and halving only when the test fails is what keeps the accepted steps from being needlessly short.

- **Ridging (Levenberg–Marquardt).** Replace J_k by $J_k + \tau\mathbb{I}$, with $\tau \geq 0$ just large enough to make it positive definite. This repairs indefiniteness and shortens the step; as τ grows the direction rotates from Newton toward plain gradient descent, and $\tau = 0$ wherever J_k is already well behaved. This is roughly what `nlm` does internally.

  A simple rule takes $\tau = \max\{0,\ \epsilon - \lambda_{\min}(J_k)\}$, which lifts the smallest eigenvalue of J_k to a small positive ϵ. A modified Cholesky factorisation achieves the same without an eigendecomposition. Since $\delta^{(k)}(\tau) = -(J_k + \tau\mathbb{I})^{-1} g_k$ has length decreasing in τ, ridging also acts as a step-length control, and with a positive definite matrix the direction satisfies $g_k^\top \delta^{(k)} < 0$, so a line search along it is guaranteed to find a decrease. A related guard for GLMs is **Fisher scoring**: the expected information $I(\theta) = X^\top W X$ is positive semi-definite at every θ, and positive definite whenever X has full column rank and all weights are positive, so the ridge is rarely needed at all.

- **Trust regions.** Minimise the quadratic approximation only within $\|\theta - \theta^{(k)}\| \leq \Delta_k$, growing or shrinking Δ_k according to how well the quadratic predicted the realised change. This is `nlminb` and the `PORT` routines.

  The quality of the prediction is measured by the ratio

  $$\rho_k = \frac{\mathcal{L}(\theta^{(k)} + \delta) - \mathcal{L}(\theta^{(k)})}{g_k^\top \delta + \frac{1}{2}\,\delta^\top J_k\,\delta}, \tag{5.40}$$

  actual decrease over the decrease predicted by the quadratic model. A typical rule rejects the step if ρ_k is small or negative and shrinks Δ_k (say by a factor of four); it accepts the step and keeps Δ_k when ρ_k is moderate; and it doubles Δ_k when $\rho_k \geq 3/4$ and the step reached the boundary. The constrained subproblem has the solution $\delta = -(J_k + \tau\mathbb{I})^{-1} g_k$ for some $\tau \geq 0$, so a trust region is ridging with τ chosen by the step length rather than the other way round. Unlike a line search, it first fixes how far to go and then chooses the direction. That makes it robust when J_k is indefinite: at a saddle point, where the Newton step points the wrong way, it still moves downhill.

- **Better parameterisation and starting values.** Often the cheapest fix. Centring and scaling the covariates improves the conditioning of J. Parameters with natural bounds are better optimised on an unconstrained scale — $\log\sigma$ rather than σ, logit p rather than p — which removes steps that leave the parameter space and usually makes ℓ closer to quadratic. Good starting values come from a simpler fit: method of moments estimates, a least-squares fit, or the null model with an intercept only. `glm` sidesteps the choice altogether: it starts from fitted means rather than coefficients, taking $\mu_i^{(0)} = (y_i + 0.5)/2$ for binary data, which are always inside $(0, 1)$, and it therefore never begins in the flat region where the weights $p_i(1 - p_i)$ vanish.

Combining the first two gives **safeguarded Newton**: at each iteration form a positive definite $\tilde{J}_k$, solve $\tilde{J}_k\,\delta^{(k)} = -g_k$, then backtrack on s. Both guards switch themselves off near a well-behaved maximum, so the method keeps Newton's fast local convergence while still making progress from remote starting values.

5.4.5.1 Shinylive App for Safeguarded Newton-Raphson

There is a shinylive app to show how the full Newton-Raphson step diverges on a logistic-regression likelihood started far from the maximum, and how backtracking on the step length rescues the same starting value.

5.5 Example of Fitting Logistic Regression with Safeguarded Multivariate Newton-Raphson

5.5.1 The model and its derivatives

With a single covariate x_i and binary response y_i, let $u_i = \beta_0 + \beta_1 x_i$ and $p_i = e^{u_i}/(1 + e^{u_i})$. The log-likelihood is

$$\ell(\beta) = \sum_{i=1}^{n} \{y_i u_i - \log(1 + e^{u_i})\}. \tag{5.41}$$

Writing X for the $n \times 2$ design matrix with rows $(1, x_i)$, the gradient of $\mathcal{L}(\beta) = -\ell(\beta)$ and the information are,

$$g(\beta) = -X^\top (y - p), \qquad J(\beta) = X^\top W X, \qquad W = \mathrm{diag}\{p_i(1 - p_i)\}. \tag{5.42}$$

Note that J does not involve y. This is the canonical-link property mentioned above, and it means $J(\beta) = I(\beta)$ identically.

Two numerical points, both carried over from the chapter on computer arithmetic. First, $\log(1 + e^u)$ overflows for $u \gtrsim 710$; the stable form is $\log(1 + e^u) = \max(u, 0) + \log(1 + e^{-|u|})$. Second, $p_i(1 - p_i) \to 0$ when $|u_i|$ is large, so W — and hence J — degenerates toward singularity when the fitted probabilities are extreme. That observation is the whole explanation of the divergence we are about to see.

5.5.1.1 The model as a computation graph

Before differentiating anything it helps to fix the object being differentiated. Write the model as a **computation graph**: one node for every intermediate quantity — the inputs, each linear unit, each activation, the linear predictor, the loss — and one directed edge $u \to z$ whenever z is computed directly from u. The graph is acyclic, values flow forward along its edges, and the whole of backpropagation is two rules stated on it.

The first is the **node rule**. Write

$$\delta_z = \frac{\partial \mathcal{L}_i}{\partial z} \tag{5.43}$$

for the sensitivity of the loss to node z. Since z can influence the loss only through the nodes it feeds, the multivariate chain rule gives one term per outgoing edge,

$$\delta_z = \sum_{p \in \mathrm{consumers}(z)} \delta_p \frac{\partial p}{\partial z}, \tag{5.44}$$

and the recursion starts at the loss itself with $\delta_{\mathcal{L}_i} = 1$.

The second is the **edge rule**. Every parameter sits on an edge that forms a linear combination, and it reaches z only through the single term $\omega\, u$ it contributes, so $\partial z/\partial \omega = u$ and

$$\frac{\partial \mathcal{L}_i}{\partial \omega_{u \to z}} = \delta_z \cdot u : \tag{5.45}$$

the sensitivity at the arrow's head times the value at its tail.

That is the entire algorithm. Everything below — logistic regression here, a hidden layer later in this chapter, a hundred layers in a modern network — is these two rules applied to a larger graph. Note that the graph is neither a chain nor a tree: a unit typically feeds *several* others, so most nodes have more than one consumer, and the sum in the node rule is what gathers those contributions. Section 5.6.5 writes the two rules out for a network of arbitrary depth.

5.5.2 The gradient by the chain rule

The gradient above can be written down directly, but it is worth deriving it the way an automatic-differentiation system would, because the same mechanics carry over unchanged to models with many more layers. Logistic regression *is* a neural network — the smallest interesting one: two inputs, one linear unit, a sigmoid activation, and a cross-entropy loss.

Take a single observation and follow it forward through the model:

$$u_i = \beta_0 \cdot 1 + \beta_1 x_i, \qquad p_i = \sigma(u_i) = \frac{1}{1+e^{-u_i}}, \qquad \mathcal{L}_i = -\left\{ y_i \log p_i + (1-y_i)\log(1-p_i) \right\}. \tag{5.46}$$

Each stage is a simple function of the one before it, so $\partial \mathcal{L}_i / \partial \beta$ is a product of three local derivatives. Working backwards:

Loss with respect to the probability. Differentiating the cross-entropy,

$$\frac{\partial \mathcal{L}_i}{\partial p_i} = -\frac{y_i}{p_i} + \frac{1-y_i}{1-p_i} = \frac{p_i - y_i}{p_i(1-p_i)}. \tag{5.47}$$

Probability with respect to the linear unit. The sigmoid has the convenient derivative

$$\frac{\partial p_i}{\partial u_i} = \sigma'(u_i) = p_i(1-p_i). \tag{5.48}$$

Linear unit with respect to each parameter. Every weight sees only the input it multiplies, so there is one local derivative per edge entering u_i:

$$\frac{\partial u_i}{\partial \beta_0} = 1 \quad \text{(the edge from the constant node 1)}, \qquad \frac{\partial u_i}{\partial \beta_1} = x_i \quad \text{(the edge from } x_i\text{)}. \tag{5.49}$$

Multiplying the first two gives the quantity that backpropagation carries between layers — the sensitivity of the loss to the linear unit, written δ_i:

$$\delta_i \;=\; \frac{\partial \mathcal{L}_i}{\partial u_i} = \frac{\partial \mathcal{L}_i}{\partial p_i} \cdot \frac{\partial p_i}{\partial u_i} = \frac{p_i - y_i}{p_i(1-p_i)} \cdot p_i(1-p_i) = p_i - y_i. \tag{5.50}$$

The factor $p_i(1-p_i)$ cancels exactly. This cancellation is the reason cross-entropy is paired with a sigmoid rather than, say, squared error: the awkward σ' term disappears and the signal sent backwards is simply the **prediction error** $p_i - y_i$. Multiplying by the last local derivative and summing over observations,

$$g(\beta) = \nabla \mathcal{L}(\beta) = \sum_{i=1}^{n} \delta_i \begin{pmatrix} \partial u_i/\partial \beta_0 \\ \partial u_i/\partial \beta_1 \end{pmatrix} = \sum_{i=1}^{n} (p_i - y_i) \begin{pmatrix} 1 \\ x_i \end{pmatrix} = X^\top (p - y) = -X^\top (y - p), \tag{5.51}$$

which is the expression used above and coded below. Note that δ_i is also exactly the per-observation gradient contribution that stochastic gradient descent samples later in this chapter.

The diagram below lays the same computation out as a network, drawn the same way as the two-layer picture later in this chapter: values flow left to right in the top panel (the **forward pass**), and derivatives accumulate right to left in the bottom panel (the **backward pass**), each arrow carrying exactly one local derivative from the list above. The intercept is drawn as the weight β_0 on an edge from a square **constant node** holding the value 1, placed directly above u_i so that the space beneath the row is free: there the sensitivity δ of each node sits directly under that node, and the two parameter gradients are collected in the box beneath the fan of β arrows. A constant node has no sensitivity of its own, since nothing it holds depends on the parameters, so in the backward panel its edge carries the weight but no arrow.

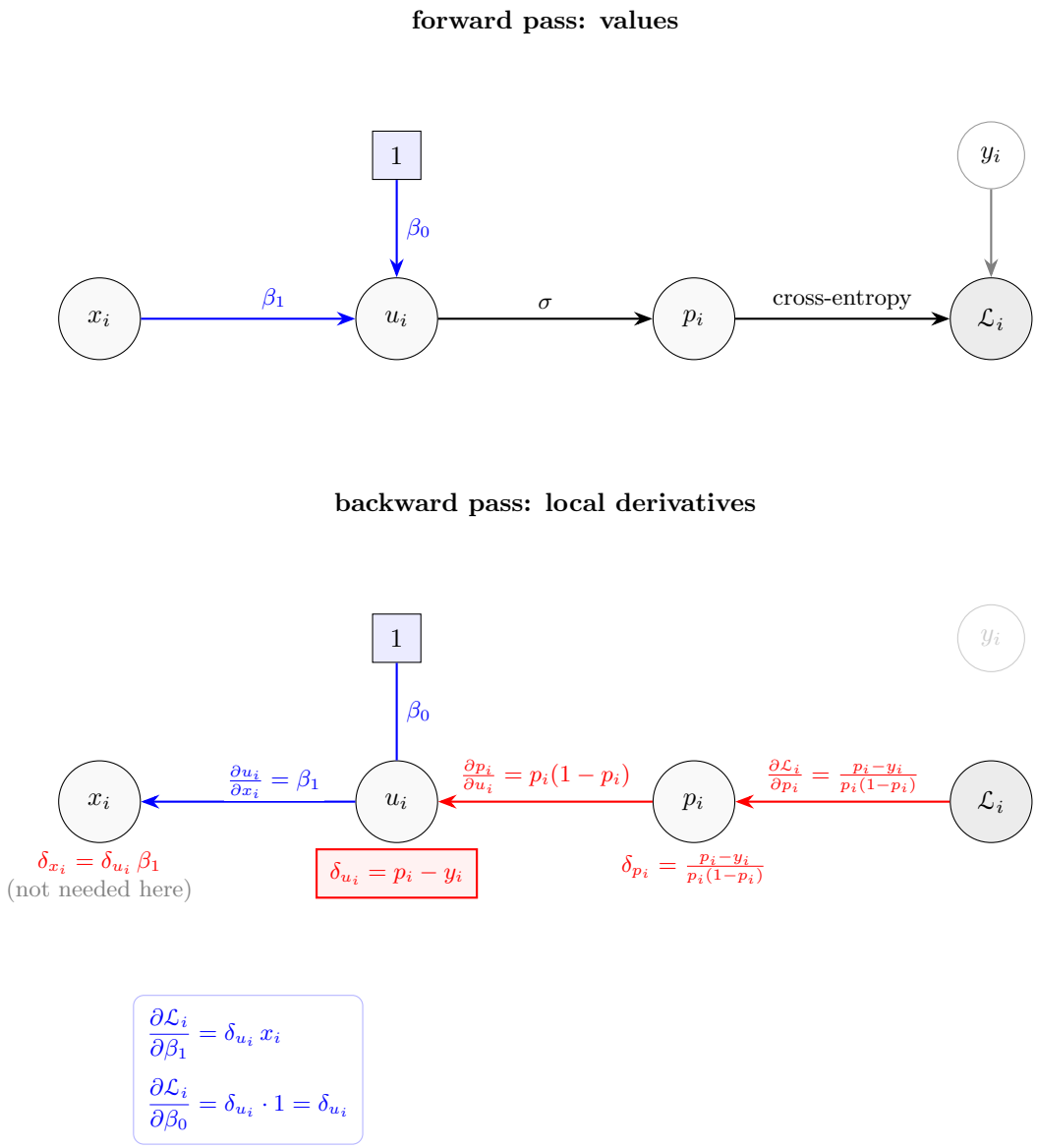


Figure 5.18: Shows how the gradient of the negative log-likelihood is assembled by the chain rule, by drawing logistic regression as the one-unit neural network it is. Logistic regression as a one-unit network: the forward pass (top) maps inputs to a linear unit, a sigmoid and a loss; the backward pass (bottom) multiplies the local derivatives, and the sigmoid factor cancels so that the signal reaching the linear unit is the prediction error $p_i - y_i$. As in the two-layer picture, each arrow carries only the local derivative of its head with respect to its tail, the sensitivity δ of each node sits beneath that node, and the parameter gradients are collected in the box beneath the fan of weight arrows.

The red box marks the cancellation: whatever the sigmoid contributes on the way forward is undone on the way back, leaving $\delta_i = p_i - y_i$. In a deeper network the same δ would be passed further left and multiplied by the next layer's local derivative; here there is only one layer, so the recursion stops immediately and $g(\beta) = \sum_i \delta_i\,(1, x_i)^\top$.

5.5.3 Implementation in R

Gradient of Logistic Regression Model

```
### numerically stable log(1 + exp(u))
log1pexp <- function(u) pmax(u, 0) + log1p(exp(-abs(u)))

### negative log-likelihood, its gradient g, and the information matrix
nll_grad_info <- function(b, y, x) {
    u <- b[1] + b[2] * x
    p <- plogis(u)
    w <- p * (1 - p)
    list(
        neg_loglik = -sum(u * y - log1pexp(u)),
        grad       = -c(sum(y - p), sum(x * (y - p))),   # g(beta)
        inf        = matrix(c(sum(w), sum(x * w),
                              sum(x * w), sum(x * x * w)), 2, 2)
    )
}

neg_logp_logistic <- function(b, x, y) {
    u <- b[1] + b[2] * x
    -sum(u * y - log1pexp(u))
}
```

Because we minimise the *negative* log-likelihood, the update $\beta^{(k+1)} = \beta^{(k)} + J^{-1}S$ becomes $\beta^{(k+1)} = \beta^{(k)} - J^{-1}g(\beta^{(k)})$, which is the form coded below.

Newton-Raphson Iterations for Logistic Regression

```
mle_logistic_nr <- function(b0, no_iter, y, x, debug = FALSE) {
    out <- matrix(NA_real_, no_iter + 1, 3,
                  dimnames = list(NULL, c("beta0", "beta1", "neg_loglike")))
    b <- b0
    for (i in seq_len(no_iter + 1)) {
        q <- nll_grad_info(b, y, x)
        out[i, ] <- c(b, q$neg_loglik)
        if (debug) print(out[i, ])
        if (i > no_iter) break
        step <- tryCatch(solve(q$inf, q$grad),
                         error = function(e) rep(NA_real_, 2))
        if (any(!is.finite(step))) break
        b <- b - step
    }
    out
}
```

```
gen_logistic_data <- function(b, n) {
    x <- sort(runif(n, -2, 2))
```

```
    p <- plogis(b[1] + b[2] * x)
    y <- rbinom(n, 1, p)
    plot(x, p, type = "l", ylim = c(0, 1), ylab = "P(y = 1)")
    points(x, y, col = y + 1)
    list(x = x, y = y)
}

set.seed(1)
data <- gen_logistic_data(c(0, 1.5), 200)
```

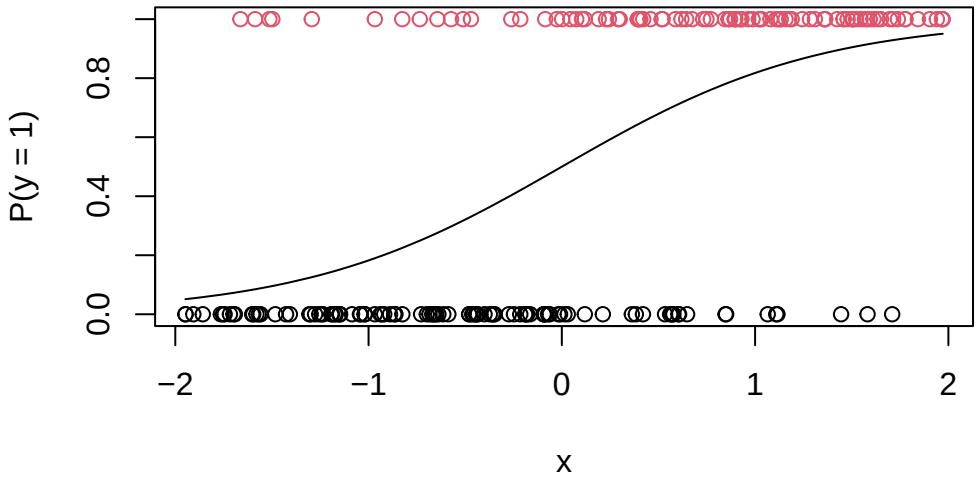


Figure 5.19: Shows the simulated logistic-regression data used as a running example for multivariate optimization. Simulated logistic data: true success probability and observed responses.

A start close to the truth converges in a handful of iterations:

```
round(mle_logistic_nr(c(0, 3), 8, data$y, data$x), 6)
```

```
          beta0     beta1 neg_loglike
 [1,]  0.000000  3.000000   117.03598
 [2,] -0.505055 -0.306054   164.86061
 [3,] -0.040097  1.281039    99.03765
 [4,] -0.251124  1.470987    97.74624
 [5,] -0.272205  1.502838    97.72964
 [6,] -0.272532  1.503424    97.72964
 [7,] -0.272532  1.503424    97.72964
 [8,] -0.272532  1.503424    97.72964
 [9,] -0.272532  1.503424    97.72964
```

The doubling of correct digits predicted by the quadratic-convergence result is visible in the final rows.

5.5.4 Comparison with `nlm` and `glm`

```
logit_nlm <- nlm(neg_logp_logistic, c(0, 0),
                 x = data$x, y = data$y, hessian = TRUE)
logit_nlm$estimate
```

```
[1] -0.2725325  1.5034237
```

```
sqrt(diag(solve(logit_nlm$hessian)))   # asymptotic standard errors
```

```
[1] 0.1797390 0.2089674
```

```
summary(glm(y ~ x, family = binomial(), data = data))$coefficients
```

```
             Estimate Std. Error   z value     Pr(>|z|)
(Intercept) -0.2725322  0.1797371 -1.516283 1.294479e-01
x            1.5034244  0.2089473  7.195234 6.235374e-13
```

The three routes agree to printing accuracy. `glm` is doing the same arithmetic as `mle_logistic_nr`: for a GLM the Newton–Raphson step can be rearranged as a weighted least-squares fit of the working response $u_i + (y_i - p_i)/\{p_i(1-p_i)\}$ on X with weights $p_i(1-p_i)$, giving **iteratively reweighted least squares (IRLS)**. This is not a different algorithm but a different arrangement of the same one, and it explains why `glm` reports "Fisher Scoring iterations" in its output.

5.6 Backpropagation Algorithm for Neural Networks

5.6.1 Example of Poisson Neural Network

Logistic regression was a one-unit network, and its Hessian was small enough to invert. Adding a single hidden layer breaks that comfort, and shows why the methods in this section exist.

Let each observation carry $p = 3$ covariates $x_i = (x_{i1}, x_{i2}, x_{i3})$ and a count response y_i. Put $J = 3$ hidden units h_{i1}, h_{i2}, h_{i3} between the covariates and the mean, and use a log link so that the mean stays positive:

$$a_{ij} = \sum_{k=0}^{p} w_{jk}\, x_{ik}, \qquad h_{ij} = \tanh(a_{ij}), \qquad \eta_i = \sum_{j=0}^{J} v_j\, h_{ij}, \qquad \lambda_i = e^{\eta_i}, \tag{5.52}$$

where $x_{i0} \equiv 1$ and $h_{i0} \equiv 1$ are constant inputs, so that the intercept of each hidden unit is the weight w_{j0} and the intercept of the linear predictor is the weight v_0, with $y_i \mid x_i \sim \text{Poisson}(\lambda_i)$. Dropping the $\log y_i!$ term, which carries no parameters, the negative log-likelihood is

$$\mathcal{L}(\theta) = \sum_{i=1}^{n} \mathcal{L}_i(\theta), \qquad \mathcal{L}_i(\theta) = \lambda_i - y_i \eta_i, \tag{5.53}$$

where θ collects the $J(p+1) = 12$ input weights w_{jk}, $k = 0, \dots, p$, and the $J + 1 = 4$ output weights v_j, $j = 0, \dots, J$ — sixteen parameters in all.

The intercepts matter: without w_{j0} every hidden unit would pass through the origin, and without v_0 the level of λ could not be shifted freely. Carrying them as ordinary weights on constant inputs means they need no separate

symbol and no separate rule: w_{j0} and v_0 are differentiated exactly like every other weight, and their local derivative happens to be the tail value 1.

Three features make this a natural home for gradient-only methods. The score equations have no closed-form solution. $\mathcal{L}$ is **not convex** — permuting the hidden units leaves it unchanged, so every minimum is one of $J!$ equivalent copies, and the Hessian is indefinite over much of the space, which is exactly where a raw Newton step misbehaves. And while a 12×12 Hessian is still cheap here, its size grows as $(Jp + J)^2$: widen the layer or the covariate set and the $O(p^3)$ solve becomes the bottleneck long before the gradient does.

5.6.2 What the model can represent

With a single covariate the model is easy to see. Each `tanh` unit switches on at a location set by its intercept w_{j0} and at a rate set by its weight w_{j1}, and enters the linear predictor scaled by its output weight v_j. It is those *scaled* units $v_j h_j(x)$ that are worth plotting: they add up, with v_0, to $\eta(x)$, and exponentiating η gives a $\lambda(x)$ with a hump — a shape a log-linear Poisson regression, whose log λ is a straight line, cannot produce at all.

```
set.seed(7)

w_true  <- c(3.0, 3.0, 1.0)      # input weight of each hidden unit
w0_true <- c(1.5, -1.5, 0.0)     # hidden intercepts: where each unit switches on
v_true  <- c(1.2, -1.2, 0.3)     # output weights
v0_true <- -0.3                  # output intercept

lambda_nn <- function(x, w, w0, v, v0)
  as.vector(exp(v0 + tanh(sweep(outer(x, w), 2, w0, "+")) %*% v))

xg <- seq(-3, 3, length.out = 400)
H  <- tanh(sweep(outer(xg, w_true), 2, w0_true, "+"))
VH <- sweep(H, 2, v_true, "*")             # the scaled units v_j h_j(x)
eg <- v0_true + rowSums(VH)                # eta(x), what they add up to

par(mfrow = c(1, 2), mar = c(4, 4.5, 2.5, 1))

matplot(xg, VH, type = "l", lty = 1, lwd = 2, col = c(1, 2, 4),
        ylim = range(VH, eg) + c(0, 1.3),   # headroom for the legend
        xlab = TeX(r"($x$)"), ylab = TeX(r"($v_j h_j(x)$)"),
        main = TeX(r"(the three scaled hidden units)"))
lines(xg, eg, lwd = 2.5, lty = 2, col = "grey30")
abline(h = 0, lty = 3)
legend("topleft", bty = "n", lty = c(1, 1, 1, 2), lwd = 2,
       col = c(1, 2, 4, "grey30"), cex = 0.8,
       legend = TeX(c(sprintf(r"($v_%d h_%d$: $w = %.1f$, $w_0 = %.1f$, $v = %.1f$)",
                              1:3, 1:3, w_true, w0_true, v_true),
                      r"($\eta(x) = v_0 + \sum_j v_j h_j(x)$)")))

## simulate counts from the curve
n  <- 150
xi <- runif(n, -3, 3)
yi <- rpois(n, lambda_nn(xi, w_true, w0_true, v_true, v0_true))
```

```
plot(xg, lambda_nn(xg, w_true, w0_true, v_true, v0_true), type = "l", lwd = 2,
     ylim = c(0, max(yi) + 0.5),
     xlab = TeX(r"($x$)"), ylab = TeX(r"($\lambda(x)$ and $y$)"),
     main = TeX(r"(mean curve and simulated counts)"))
points(xi, yi, pch = 19, col = adjustcolor(2, 0.45), cex = 0.7)
legend("topright", bty = "n", lwd = c(2, NA), pch = c(NA, 19), col = c(1, adjustcolor(2, 0.6)),
       legend = TeX(c(r"($\lambda(x)$)", r"($y_i \sim Poisson(\lambda(x_i))$)")))
```

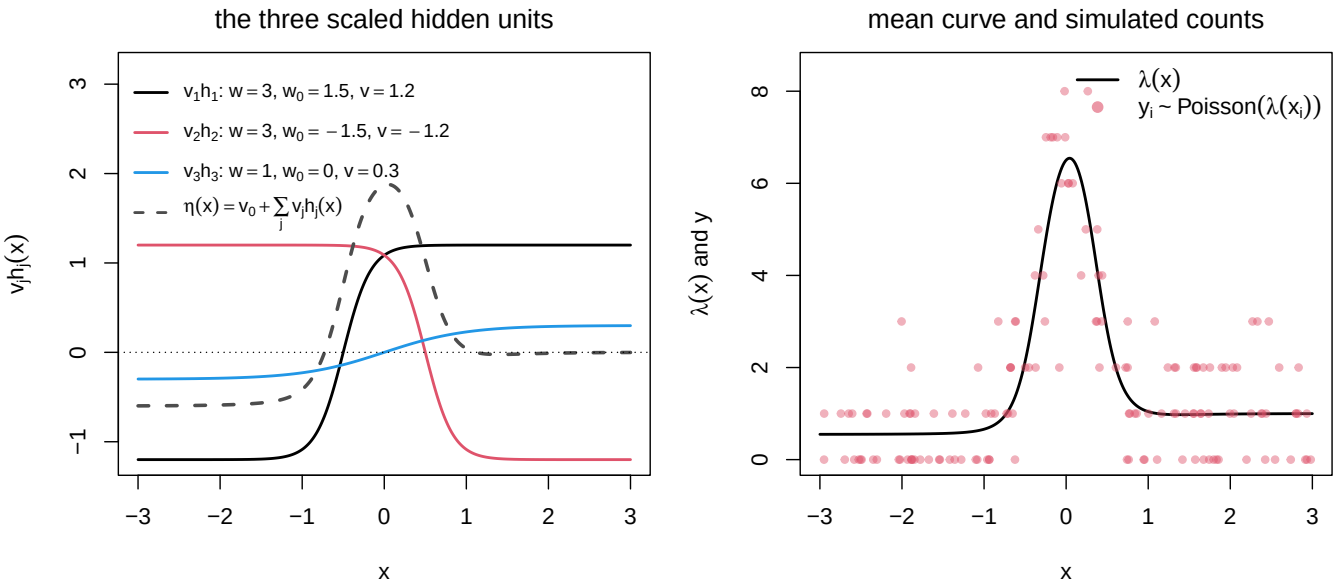


Figure 5.20: Shows what a hidden layer buys in a Poisson regression, by displaying the three scaled hidden units and the non-monotone mean curve they combine into, together with counts simulated from it. A one-covariate Poisson network with three tanh units: the scaled activations $v_j h_j(x)$ and the linear predictor $\eta(x)$ they sum to (left), and the resulting mean $\lambda(x) = e^{\eta(x)}$ with simulated counts (right), a shape no log-linear model could produce.

The counts scatter around the curve with variance equal to the mean, so they bunch near zero in the flat regions and spread out under the peak.

5.6.3 The gradient by backpropagation

The chain rule applies exactly as in the logistic case, only with one more layer to traverse. Working backwards from the loss for a single observation:

Loss to the linear predictor. In terms of λ_i, the loss of one observation is

$$\mathcal{L}_i = \lambda_i - y_i \log \lambda_i, \qquad \lambda_i = e^{\eta_i}, \tag{5.54}$$

which is $\lambda_i - y_i\eta_i$ once $\log \lambda_i = \eta_i$ is substituted. The two local derivatives at the output are

$$\frac{\partial \mathcal{L}_i}{\partial \lambda_i} = 1 - \frac{y_i}{\lambda_i}, \qquad \frac{\partial \lambda_i}{\partial \eta_i} = \lambda_i, \tag{5.55}$$

whose product is again a bare prediction error, with the λ_i cancelling just as $p_i(1-p_i)$ did for the sigmoid:

$$\delta_i \;=\; \frac{\partial \mathcal{L}_i}{\partial \eta_i} = \left(1 - \frac{y_i}{\lambda_i}\right)\lambda_i = \lambda_i - y_i. \tag{5.56}$$

Output layer. Since $\partial \eta_i / \partial v_j = h_{ij}$ and $\partial \eta_i / \partial v_0 = h_{i0} = 1$,

$$\frac{\partial \mathcal{L}_i}{\partial v_j} = \delta_i \, h_{ij}, \qquad \frac{\partial \mathcal{L}_i}{\partial v_0} = \delta_i. \tag{5.57}$$

Into the hidden layer. The path from a_{ij} to the loss runs through h_{ij} and then η_i, contributing $\partial \eta_i / \partial h_{ij} = v_j$ and $\partial h_{ij} / \partial a_{ij} = 1 - h_{ij}^2$ (the `tanh` derivative). Taking them one at a time gives a sensitivity at each node on the way,

$$\delta^h_{ij} \;=\; \frac{\partial \mathcal{L}_i}{\partial h_{ij}} = \delta_i \, v_j, \qquad \delta^a_{ij} \;=\; \frac{\partial \mathcal{L}_i}{\partial a_{ij}} = \delta^h_{ij}\left(1 - h_{ij}^2\right). \tag{5.58}$$

Input layer. Finally $\partial a_{ij} / \partial w_{jk} = x_{ik}$ for every k, including $\partial a_{ij} / \partial w_{j0} = x_{i0} = 1$, so

$$\frac{\partial \mathcal{L}_i}{\partial w_{jk}} = \delta^a_{ij} \, x_{ik}, \qquad \frac{\partial \mathcal{L}_i}{\partial w_{j0}} = \delta^a_{ij}. \tag{5.59}$$

Each intercept derivative is the sensitivity at its own unit, unmultiplied. Splitting at a_{ij} is also what makes the computation efficient: δ^a_{ij} is formed once and reused for all p weights entering unit j.

5.6.4 Two kinds of arrow

Every arrow in the network is one of two kinds, and the distinction organises the whole computation.

Arrows that carry a weight, drawn blue below, are the ones forming a linear combination: $x_{ik} \to a_{ij}$ carries w_{jk}, $h_{ij} \to \eta_i$ carries v_j, and the edges from the constant nodes holding $x_{i0} = h_{i0} = 1$ into a_{ij} and η_i carry the intercepts w_{j0} and v_0. *Every parameter of the model sits on one of these arrows*, and these are the only derivatives we ultimately want.

Arrows that carry no weight are the fixed transformations: $a_{ij} \to h_{ij}$ is `tanh`, $\eta_i \to \lambda_i$ is exp, $\lambda_i \to \mathcal{L}_i$ is the Poisson loss. Nothing is estimated on them, so they contribute no gradient component of their own; they only pass sensitivity along.

Backpropagation computes one number per node — the sensitivity of the loss to that node,

$$\delta_z = \frac{\partial \mathcal{L}_i}{\partial z}, \tag{5.60}$$

obtained by multiplying the local derivatives along the arrows from the loss back to z. Once δ is known at a node, the edge rule of Section 5.5.1.1 makes the weights feeding it free:

The gradient of the weight on a blue arrow $u \to z$ is δ_z times the value at the arrow's tail: .

$$> \frac{\partial \mathcal{L}_i}{\partial \omega_{u \to z}} = \delta_z \cdot u. > . \tag{5.61}$$

The reason is immediate: z is a linear combination, so it depends on ω only through the single term $\omega\, u$, giving $\partial z/\partial \omega = u$. For this network, with $\delta_\eta = \lambda_i - y_i$, $\delta_{h_j} = \delta_\eta\, v_j$ and $\delta_{a_j} = \delta_{h_j}(1 - h_{ij}^2)$,

$$\frac{\partial \mathcal{L}_i}{\partial v_j} = \delta_\eta\, h_{ij}, \qquad \frac{\partial \mathcal{L}_i}{\partial v_0} = \delta_\eta \cdot 1, \qquad \frac{\partial \mathcal{L}_i}{\partial w_{jk}} = \delta_{a_j}\, x_{ik}, \qquad \frac{\partial \mathcal{L}_i}{\partial w_{j0}} = \delta_{a_j} \cdot 1. \tag{5.62}$$

The intercept gradients are the same expression with a tail value of 1, which is the whole content of "an intercept is a weight on a constant input".

This also explains the one label that might look out of place. Along the blue arrow $h_{ij} \to \eta_i$ the local derivative is

$$\frac{\partial \eta_i}{\partial h_{ij}} = v_j, \tag{5.63}$$

and it plays **no part in v_j's own gradient** — a weight's gradient never involves that weight. Its job is to carry δ from η_i back to h_{ij}, so v_j appears in the gradient of every weight *behind* unit j and in none at or after it. In a deeper network this is the only kind of factor that accumulates: a path from the loss to an early weight is a long product of such node-to-node derivatives, closed off at the end by one tail value.

Summing over observations gives the full gradient $g(\theta)$. This is backpropagation in miniature: one forward sweep stores h, η and λ; one backward sweep turns δ_i into δ_{ij}^h, then δ_{ij}^a, reading off every partial derivative on the way. The cost is a small multiple of one function evaluation, no matter how many parameters there are — which is precisely why first-order methods scale where Newton does not.

The diagram below is the logistic picture with a layer inserted, and it is drawn twice: once for the forward pass and once for the backward pass, so that neither set of arrows has to share an edge with the other. The intercepts are drawn as in the logistic picture: each unit that has an intercept gets its own square constant node holding the value 1, directly above it, and the intercept — w_{j0} for hidden unit j, v_0 for η — is the weight on the edge from that square. Every parameter is then a w or a v sitting on an ordinary edge, the intercepts included.

The pre-activation a_j is drawn as a node of its own, so that every arrow joins two quantities that actually appear in the picture and the `tanh` step gets an edge to sit on. The backward panel traces one **continuous path** from the loss to a single weight, $L \to \lambda \to \eta \to h_j \to a_j \to w_{jk}$, in bold. Multiplying the four local derivatives along it gives that weight's partial derivative; every other weight is reached by a path of the same shape, drawn faint. Only the highlighted edges are labelled — the rest carry the same formulas with different indices.

Each arrow in the backward panel carries **only** its local derivative. The sensitivity δ of each node is written directly beneath that node (including δ_{x_k} at the input layer, which is well defined but never needed, since no parameter hangs off an input). A constant node, by contrast, has no δ at all, because the value 1 it holds does not depend on the parameters, so its edge carries the intercept but no arrow. The parameter gradients — each one a δ at an arrow's head times the value at its tail — are collected in the boxes beneath the fan of arrows whose weights they belong to.

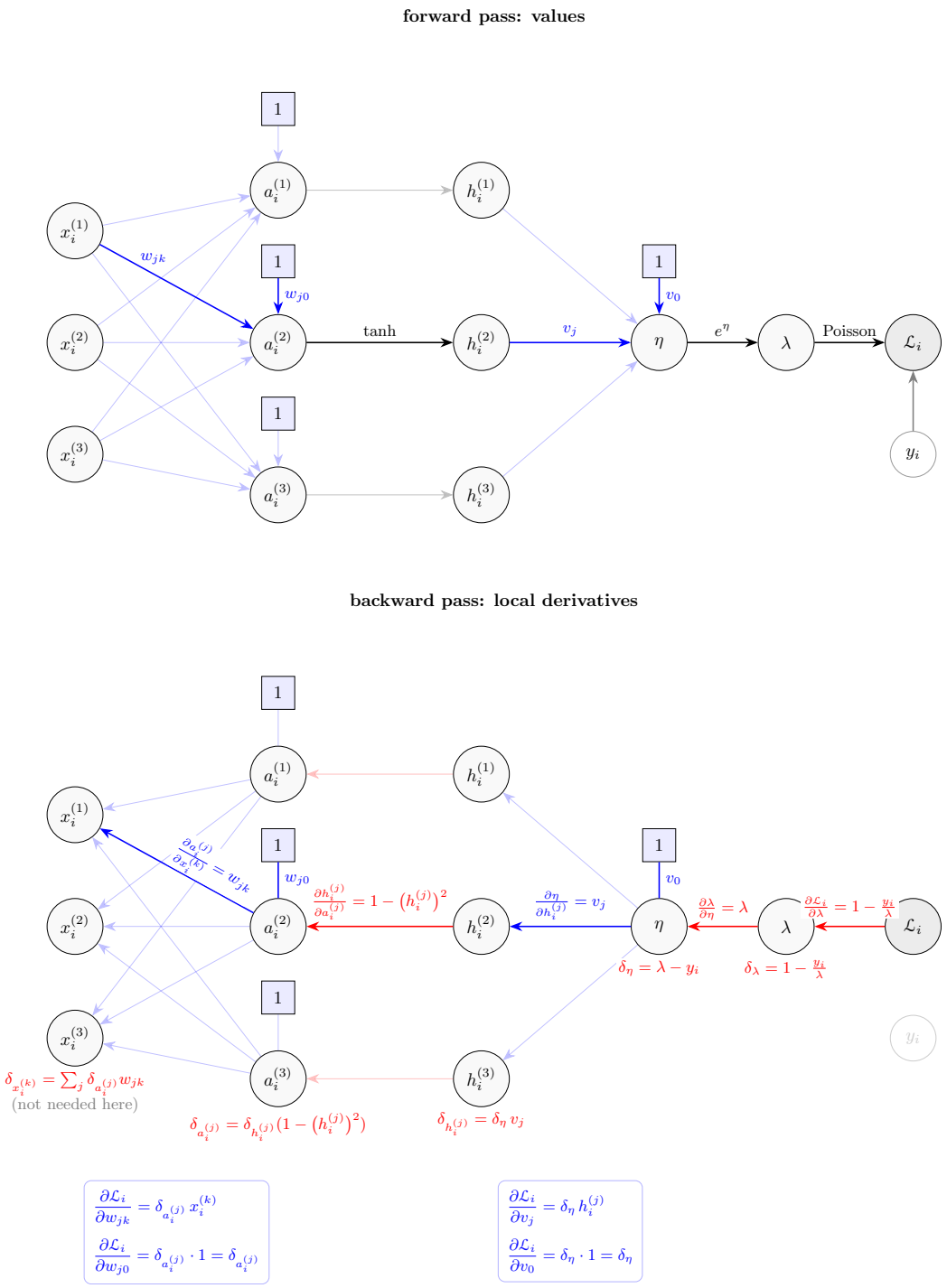


Figure 5.21: This figure shows the way that backpropagation assembles the gradient of a two-layer Poisson network, by labelling each edge with its local derivative. The network is oriented left-to-right, with the nodes of each layer stacked vertically: the forward pass (top) carries values rightward, and the backward pass (bottom) carries local derivatives leftward. Each arrow carries only the local derivative of its head with respect to its tail; the sensitivity δ of each node sits directly beneath that node, and the parameter gradients — obtained by multiplying the δ at the arrow's head by the value at its tail — are collected in the boxes beneath the corresponding fan of weight arrows. Each intercept is the weight on an edge from a square constant node holding the value 1; a constant node has no δ, so in the backward pass its edge carries no arrow.

R function for Computing and Illustrating the Gradient of Single-layer Neural Networks

```
library(gt)
J <- 3                                      # number of hidden units

## math column header for gt tables
m <- function(tex) html(paste0("\\(", tex, "\\)"))

## theta = (W[J x p, by column], w0, v, v0); X holds x_1..x_p (no constant column)
## workingtable = TRUE also returns three gt tables (first nshow observations):
##   forward values of the nodes, backward deltas, and the gradient as inner products
nn_pois <- function(theta, X, y, workingtable = FALSE, nshow = 6) {
  p <- ncol(X)

  ## ---- extract weights from theta ----
  W  <- matrix(theta[1:(J * p)], J, p)      # input weights w_jk, k = 1..p
  w0 <- theta[J * p + 1:J]                  # hidden intercepts w_j0
  v  <- theta[J * p + J + 1:J]              # output weights v_j
  v0 <- theta[J * p + 2 * J + 1]            # output intercept v_0

  ## ---- forward pass: store a, h, eta, lambda (needed by the backward pass) ----
  A   <- sweep(X %*% t(W), 2, w0, "+")      # n x J  a_ij = w_j0 + sum_k w_jk x_ik
  H   <- tanh(A)                            # n x J  h_ij = tanh(a_ij)
  eta <- as.vector(v0 + H %*% v)            # n      eta_i = v_0 + sum_j v_j h_ij
  lam <- exp(eta)                           # n      lambda_i = exp(eta_i)

  ## ---- loss: Poisson negative log-likelihood ----
  nll <- sum(lam - y * eta)                 # dropping log(y!)

  ## ---- backward pass: sensitivities delta, from the loss back to a ----
  delta_eta <- lam - y                      # n      dL_i/d eta_i  (seed)
  delta_h   <- matrix(0, nrow(X), J)        # n x J  dL_i/d h_ij = delta_eta_i * v_j
  for (j in 1:J) delta_h[, j] <- delta_eta * v[j]
  delta_a   <- delta_h * (1 - H^2)          # n x J  dL_i/d a_ij = delta_h * tanh'(a_ij)

  ## ---- gradients of the weights: delta at the arrow's head x value at its tail,
  ##      summed over observations (the intercepts have tail value 1) ----
  d_v  <- as.vector(t(H) %*% delta_eta)     # J      dL/dv_j  = sum_i delta_eta_i h_ij
  d_v0 <- sum(delta_eta)                    # 1      dL/dv_0
  d_W  <- t(delta_a) %*% X                  # J x p  dL/dw_jk = sum_i delta_a_ij x_ik
  d_w0 <- colSums(delta_a)                  # J      dL/dw_j0

  ## ---- pack the gradient in the same order as theta, with names ----
  grad <- c(as.vector(d_W), d_w0, d_v, d_v0)
  names(grad) <- c(paste0("w", rep(1:J, p), rep(1:p, each = J)),   # w11, w21, ..., w3p
                   paste0("w", 1:J, "0"), paste0("v", 1:J), "v0")
  out <- list(nll = nll, grad = grad)
  if (!workingtable) return(out)

  ## ---- working tables: one row per observation i ----
  wk <- data.frame(i = seq_along(y), y = y, X, A, H, eta = eta, lam = lam,
                   delta_eta = delta_eta, delta_h, delta_a)
```

```
names(wk) <- c("i", "y", paste0("x", 1:p), paste0("a", 1:J), paste0("h", 1:J),
               "eta", "lam", "delta_eta", paste0("delta_h", 1:J), paste0("delta_a", 1:J))
top <- head(wk, nshow)

## forward values
forward <- top |>
  gt() |>
  cols_hide(starts_with("delta")) |>
  cols_label(i = m("i"), y = m("y_i"),
             x1 = m("x_i^{(1)}"), x2 = m("x_i^{(2)}"), x3 = m("x_i^{(3)}"),
             a1 = m("a_i^{(1)}"), a2 = m("a_i^{(2)}"), a3 = m("a_i^{(3)}"),
             h1 = m("h_i^{(1)}"), h2 = m("h_i^{(2)}"), h3 = m("h_i^{(3)}"),
             eta = m("\\eta_i"), lam = m("\\lambda_i")) |>
  tab_spanner(m("\\text{covariates}"), c(x1, x2, x3)) |>
  tab_spanner(m("a_i^{(j)} = w_{j0} + \\sum_k w_{jk} x_i^{(k)}"), c(a1, a2, a3)) |>
  tab_spanner(m("h_i^{(j)} = \\tanh(a_i^{(j)})"), c(h1, h2, h3)) |>
  tab_spanner(m("\\eta_i = v_0 + \\sum_j v_j h_i^{(j)}, \\; \\lambda_i = e^{\\eta_i}"),
    ↪  c(eta, lam)) |>
  fmt_number(-c(i, y), decimals = 3) |>
  tab_header(title = "Forward pass: values of the nodes")

## backward deltas
backward <- top |>
  gt() |>
  cols_hide(c(y, starts_with("x"), starts_with("a"), starts_with("h"), eta, lam)) |>
  cols_label(i = m("i"), delta_eta = m("\\delta_{\\eta_i} = \\lambda_i - y_i"),
             delta_h1 = m("\\delta_{h_i^{(1)}}"), delta_h2 = m("\\delta_{h_i^{(2)}}"),
             delta_h3 = m("\\delta_{h_i^{(3)}}"),
             delta_a1 = m("\\delta_{a_i^{(1)}}"), delta_a2 = m("\\delta_{a_i^{(2)}}"),
             delta_a3 = m("\\delta_{a_i^{(3)}}")) |>
  tab_spanner(m("\\delta_{h_i^{(j)}} = \\delta_{\\eta_i}\\, v_j"), starts_with("delta_h")) |>
  tab_spanner(m("\\delta_{a_i^{(j)}} = \\delta_{h_i^{(j)}} \\bigl(1 - (h_i^{(j)})^2\\bigr)"),
    ↪  starts_with("delta_a")) |>
  fmt_number(-i, decimals = 3) |>
  tab_header(title = "Backward pass: sensitivities")

## gradient: inner product of two columns of the full working table (all n rows)
ip <- function(u, w) sum(wk[[u]] * w)
tex <- val <- character(0)
for (k in 1:p) for (j in 1:J) {
  tex <- c(tex, sprintf("\\sum_i \\delta_{a_i^{(%d)}}\\, x_i^{(%d)}", j, k))
  val <- c(val, ip(paste0("delta_a", j), wk[[paste0("x", k)]]))
}
for (j in 1:J) {
  tex <- c(tex, sprintf("\\sum_i \\delta_{a_i^{(%d)}}\\cdot 1", j))
  val <- c(val, ip(paste0("delta_a", j), 1))
}
for (j in 1:J) {
  tex <- c(tex, sprintf("\\sum_i \\delta_{\\eta_i}\\, h_i^{(%d)}", j))
  val <- c(val, ip("delta_eta", wk[[paste0("h", j)]]))
}
```

```
  tex <- c(tex, "\\sum_i \\delta_{\\eta_i}\\cdot 1"); val <- c(val, ip("delta_eta", 1))

  par_tex <- c(sprintf("w_{%d%d}", rep(1:J, p), rep(1:p, each = J)),
               sprintf("w_{%d0}", 1:J), sprintf("v_{%d}", 1:J), "v_0")
  gradient <- data.frame(par = paste0("\\(", par_tex, "\\)"),
                         inner = paste0("\\(", tex, "\\)"),
                         value = as.numeric(val), back = unname(grad)) |>
    gt() |>
    fmt_passthrough(c(par, inner), escape = FALSE) |>
    cols_label(par = m("\\text{weight}"),
               inner = m("\\text{inner product of two columns}"),
               value = m("\\text{value}"), back = m("\\text{gradient}")) |>
    fmt_number(c(value, back), decimals = 4) |>
    tab_header(title = "Gradient: inner products over all observations")

  c(out, list(tables = list(forward = forward, backward = backward, gradient = gradient)))
}

set.seed(3)
n <- 40; p <- 3
X <- matrix(rnorm(n * p), n, p)
theta0 <- rnorm(J * p + 2 * J + 1, sd = 0.5)
## counts simulated from the network at theta0
ytrue <- rpois(n, local({
  W <- matrix(theta0[1:(J * p)], J, p); w0 <- theta0[J * p + 1:J]
  v <- theta0[J * p + J + 1:J];        v0 <- theta0[J * p + 2 * J + 1]
  exp(v0 + tanh(sweep(X %*% t(W), 2, w0, "+")) %*% v)
}))

fit  <- nn_pois(theta0, X, ytrue, workingtable = TRUE)
back <- fit$grad                          # named gradient

## central differences, one coordinate at a time
h <- 1e-6
numeric_grad <- vapply(seq_along(theta0), function(k) {
  e <- numeric(length(theta0)); e[k] <- h
  (nn_pois(theta0 + e, X, ytrue)$nll - nn_pois(theta0 - e, X, ytrue)$nll) / (2 * h)
}, numeric(1))
```

Check the Gradient Ouput

```
cbind(backprop = back, numeric = numeric_grad)
```

```
        backprop       numeric
w11  0.4737615235  0.4737615296
w21 -1.0509837992 -1.0509837960
w31  0.0001707534  0.0001707647
w12 -1.3588962590 -1.3588962613
w22  4.1853262037  4.1853262047
w32 -0.0881340975 -0.0881340956
w13  0.3705859680  0.3705859690
w23  0.8368418239  0.8368418190
```

```
w33  0.0974066121  0.0974066090
w10 -0.9574699509 -0.9574699522
w20  1.3058687313  1.3058687340
w30 -0.0693719994 -0.0693719926
v1  -0.6236882995 -0.6236883063
v2  -1.8296636752 -1.8296636757
v3   0.8358353706  0.8358353725
v0  -1.3255493620 -1.3255493627
```

```
max(abs(back - numeric_grad))              # agreement across all 16 coordinates
```

```
[1] 1.130205e-08
```

The working tables. Calling the function with `workingtable = TRUE` also returns three tables (the first six of the $n = 40$ observations are shown). The forward pass fills in the first, from the covariates to λ_i.

```
fit$tables$forward
```

Forward pass: values of the nodes

		covariates			$a_i^{(j)} = w_{j0} + \sum_k w_{jk} x_i^{(k)}$			$h_i^{(j)} = \tanh(a_i^{(j)})$			$\eta_i = v_0 + \sum_j v_j h_i^{(j)},\ \lambda_i = e^{\eta_i}$	
i	y_i	$x_i^{(1)}$	$x_i^{(2)}$	$x_i^{(3)}$	$a_i^{(1)}$	$a_i^{(2)}$	$a_i^{(3)}$	$h_i^{(1)}$	$h_i^{(2)}$	$h_i^{(3)}$	η_i	λ_i
1	2	-0.962	0.787	1.736	1.442	-0.141	-1.973	0.894	-0.140	-0.962	-0.482	0.618
2	1	-0.293	-0.310	-0.352	0.804	0.305	0.418	0.666	0.296	0.395	-0.921	0.398
3	0	0.259	1.699	0.689	0.106	0.739	-1.103	0.106	0.629	-0.802	-1.687	0.185
4	0	-1.152	-0.795	1.224	1.868	-0.540	-0.979	0.953	-0.493	-0.753	-0.136	0.873
5	0	0.196	0.348	0.794	0.501	0.237	-0.695	0.463	0.232	-0.601	-1.085	0.338
6	0	0.030	-2.265	-0.006	1.103	-0.410	0.949	0.802	-0.388	0.739	-0.239	0.787

The backward pass fills in the second, from δ_{η_i} back to $\delta_{a_i^{(j)}}$.

```
fit$tables$backward
```

Backward pass: sensitivities

		$\delta_{h_i^{(j)}} = \delta_{\eta_i} v_j$			$\delta_{a_i^{(j)}} = \delta_{h_i^{(j)}}(1 - (h_i^{(j)})^2)$		
i	$\delta_{\eta_i} = \lambda_i - y_i$	$\delta_{h_i^{(1)}}$	$\delta_{h_i^{(2)}}$	$\delta_{h_i^{(3)}}$	$\delta_{a_i^{(1)}}$	$\delta_{a_i^{(2)}}$	$\delta_{a_i^{(3)}}$
1	-1.382	-1.032	1.128	-0.088	-0.207	1.106	-0.007
2	-0.602	-0.449	0.491	-0.038	-0.250	0.448	-0.032
3	0.185	0.138	-0.151	0.012	0.137	-0.091	0.004
4	0.873	0.651	-0.712	0.056	0.059	-0.539	0.024
5	0.338	0.252	-0.276	0.022	0.198	-0.261	0.014
6	0.787	0.588	-0.642	0.050	0.210	-0.545	0.023

Each component of the gradient is then the **inner product of two columns** of the working table over all n observations — a δ column against the column at the tail of the weight's arrow, or against a column of ones for an intercept. The last column is the named gradient returned by the function.

```
fit$tables$gradient
```

Gradient: inner products over all observations

weight	inner product of two columns	value	gradient
w_{11}	$\sum_i \delta_{a_i^{(1)}} \, x_i^{(1)}$	0.4738	0.4738
w_{21}	$\sum_i \delta_{a_i^{(2)}} \, x_i^{(1)}$	-1.0510	-1.0510
w_{31}	$\sum_i \delta_{a_i^{(3)}} \, x_i^{(1)}$	0.0002	0.0002
w_{12}	$\sum_i \delta_{a_i^{(1)}} \, x_i^{(2)}$	-1.3589	-1.3589
w_{22}	$\sum_i \delta_{a_i^{(2)}} \, x_i^{(2)}$	4.1853	4.1853
w_{32}	$\sum_i \delta_{a_i^{(3)}} \, x_i^{(2)}$	-0.0881	-0.0881
w_{13}	$\sum_i \delta_{a_i^{(1)}} \, x_i^{(3)}$	0.3706	0.3706
w_{23}	$\sum_i \delta_{a_i^{(2)}} \, x_i^{(3)}$	0.8368	0.8368
w_{33}	$\sum_i \delta_{a_i^{(3)}} \, x_i^{(3)}$	0.0974	0.0974
w_{10}	$\sum_i \delta_{a_i^{(1)}} \cdot 1$	-0.9575	-0.9575
w_{20}	$\sum_i \delta_{a_i^{(2)}} \cdot 1$	1.3059	1.3059
w_{30}	$\sum_i \delta_{a_i^{(3)}} \cdot 1$	-0.0694	-0.0694
v_1	$\sum_i \delta_{\eta_i} \, h_i^{(1)}$	-0.6237	-0.6237
v_2	$\sum_i \delta_{\eta_i} \, h_i^{(2)}$	-1.8297	-1.8297
v_3	$\sum_i \delta_{\eta_i} \, h_i^{(3)}$	0.8358	0.8358
v_0	$\sum_i \delta_{\eta_i} \cdot 1$	-1.3255	-1.3255

The working graph. The same numbers can be written on the computational graph itself, which is how backpropagation is usually done by hand. In Figure 5.22 each node holds its δ for the first three observations (the input nodes hold the covariates, the tail values that the input weights are multiplied by), and each edge shows its weight at θ_0 together with the gradient of that weight in green. As in Figure 5.21, the intercepts are the weights on the edges from the square constant nodes holding 1. The gradients are inner products over all $n = 40$ observations, so they are the values in the last column of the table above, not sums of the three numbers shown.

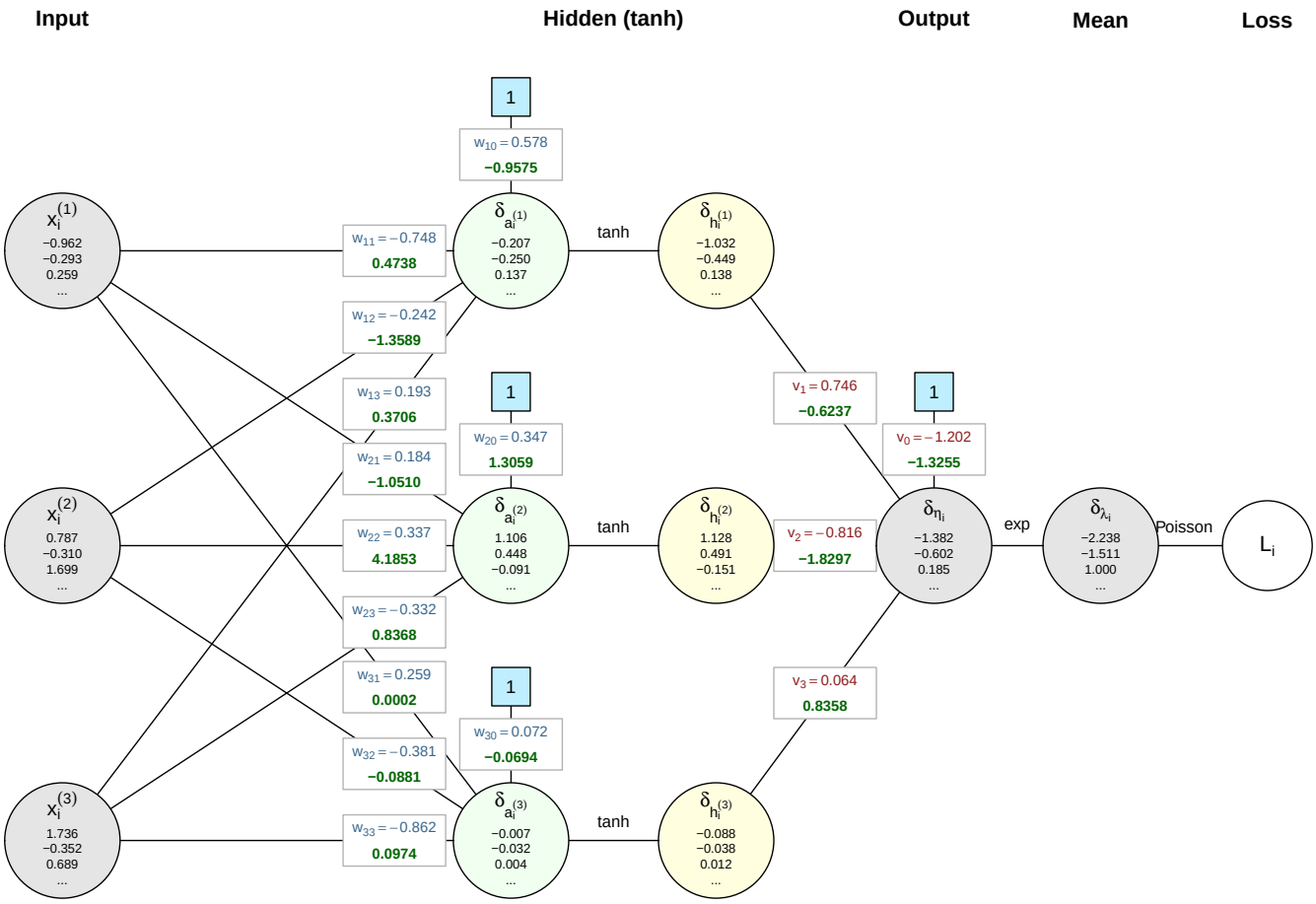


Figure 5.22: Shows the backward pass of the Poisson network on actual numbers, so that each gradient can be traced to the δ at the head of its edge and the value at its tail. The computational graph at θ_0: each node shows its sensitivity δ for observations $i = 1, 2, 3$ of the $n = 40$ (the input nodes show the covariates), and each edge shows its weight and, in green, the gradient of the negative log-likelihood with respect to that weight, summed over all 40 observations. The squares are constant nodes holding 1, whose edges carry the intercepts.

5.6.5 A General Description of Gradient of Neural Networks

Neither rule of Section 5.5.1.1 mentions depth, so the same pair describes a network of any size. Write a feed-forward network of L layers as

$$h^{(0)} = x, \qquad a^{(l)} = W^{(l)} h^{(l-1)}, \qquad h^{(l)} = f\big(a^{(l)}\big), \qquad l = 1, \dots, L, \tag{5.64}$$

with a constant entry folded into each $h^{(l-1)}$ so that the intercepts are ordinary columns of $W^{(l)}$, exactly as the constant nodes holding 1 supply the intercepts in Figure 5.21.

The backward pass has the same shape as the forward pass. Applying the node rule to a whole layer at once turns the sum over consumers into a matrix–vector product with the transposed weight matrix:

$$\delta_{h^{(l-1)}} = W^{(l)\top} \delta_{a^{(l)}}, \qquad \delta_{a^{(l)}} = \delta_{h^{(l)}} \odot f'\big(a^{(l)}\big), \tag{5.65}$$

where $\odot$ is the elementwise product. Set the two sweeps side by side and they are the same two operations:

	forward	backward
combine	$a^{(l)} = W^{(l)} h^{(l-1)}$	$\delta_{h^{(l-1)}} = W^{(l)\top} \delta_{a^{(l)}}$
transform	$h^{(l)} = f\big(a^{(l)}\big)$	$\delta_{a^{(l)}} = \delta_{h^{(l)}} \odot f'\big(a^{(l)}\big)$

A layer of the backward pass takes a linear combination of the quantities in the layer just processed and then rescales each entry — the same two steps the forward pass performs on values. Three things differ. The matrix is transposed, so the linear combination runs across the arrows *entering* a unit rather than those leaving it. The elementwise step multiplies by $f'(a^{(l)})$ instead of applying f, which makes the backward pass **linear** in δ once the activations are fixed. And the sweep runs from $l = L$ down to $l = 1$.

Starting the recursion. The sweep needs one number to begin with, the sensitivity at the output. For the Poisson network of this section that is

$$\delta_{\eta_i} = \frac{\partial \mathcal{L}_i}{\partial \eta_i} = \lambda_i - y_i, \tag{5.66}$$

and for the logistic model of Section 5.5.1.1 it is $p_i - y_i$. In both cases the derivative of the output activation cancels against the derivative of the loss, which is why the seed is a bare prediction error.

Reading off the gradients. Once $\delta_{a^{(l)}}$ is known, the edge rule delivers the whole weight matrix at once as an outer product,

$$\frac{\partial \mathcal{L}_i}{\partial W^{(l)}} = \delta_{a^{(l)}}\, h^{(l-1)\top}, \tag{5.67}$$

one entry $\delta_{a_j^{(l)}} h_k^{(l-1)}$ per weight $w_{jk}^{(l)}$: the sensitivity at the arrow's head times the value at its tail. The label $\delta_{x_k} = \sum_j \delta_{a_j} w_{jk}$ in Figure 5.21 is the node rule at $l = 1$. It is marked unneeded there because no parameter hangs off an input, but in a deeper network it is precisely the δ handed to the layer below.

The complete gradient for one observation is therefore four steps: sweep forward storing $a^{(l)}$ and $h^{(l)}$; seed δ at the output; sweep backward through the pair of recursions above; and form one outer product per layer along the way. Summing over observations gives $g(\theta)$.

Cost. The two sweeps consist of the same L matrix–vector products, one per layer, with the same matrices — $W^{(l)}$ going up and $W^{(l)\top}$ coming down. A gradient therefore costs a small constant multiple of one evaluation of $\mathcal{L}_i$, about $O(LW^2)$ for width W, or one multiply–add per parameter per sweep, *independent of how many parameters there are*. This is worth contrasting with the form in which the chain rule is usually written. Expanding $\partial \mathcal{L}_i / \partial w$ as a sum over paths from the loss to w produces about W^L terms, exponential in depth. The recursion computes the same quantity by visiting each *edge* once: a node's δ is formed once and reused by every node that feeds it, so shared sub-paths are never recomputed. In the language of algorithms the graph is a DAG and the backward sweep is dynamic programming over it in reverse topological order, which stands to path enumeration as a memoized recursion stands to a naive one.

Memory. The backward sweep consumes what the forward sweep produced — $h^{(l-1)}$ for the outer product, $f'(a^{(l)})$ for the elementwise step — so the activations have to be stored, at a cost proportional to the total number of nodes. That is why the diagrams above have a forward panel at all, and it is the cost that gradient checkpointing trades back for recomputation in very deep models.

Two extensions. Weight sharing needs no new machinery: when one parameter sits on many edges, as in a convolutional or recurrent network, its gradient is the sum of the edge rule over all of them, which is the node rule again. And the direction of the sweep is a choice. Propagating $\partial z/\partial\theta_k$ *forward* is equally correct but costs one sweep per parameter, whereas sweeping backward costs one sweep per output and there is a single scalar output here. That asymmetry is what makes a full gradient affordable for models with millions of parameters, and it is the property the rest of this chapter relies on.

Backpropagation is easy to get subtly wrong, so it is worth testing against finite differences before handing the gradient to an optimizer — the same comparison made for the univariate case earlier in this chapter.

The two agree to the accuracy of the difference quotient. With a gradient this cheap and this reliable, BFGS or conjugate gradient can fit the model without ever forming a Hessian — which is what the rest of this section is about.

5.7 Gradient-Based Methods

Newton–Raphson is the method of choice when the second derivatives are available and p is modest. Outside that regime, three considerations push toward alternatives: derivatives may be unavailable or painful to code; the $O(p^3)$ cost of solving the Newton system and the $O(p^2)$ storage of J become prohibitive for large p; and the objective may be non-smooth, noisy or multimodal.

As everywhere in this chapter, the objective is the negative log-likelihood $\mathcal{L}(\theta) = -\ell(\theta)$, whose minimum is exactly the maximum of ℓ. This is also the convention that R's `optim()`, `nlm()`, and `nlminb()` use internally, so the code below matches what those functions do under the hood. The methods here apply to any smooth objective, not only a likelihood, so this section writes f for the function being minimized and keeps $g = \nabla f$ for its gradient — the same g used everywhere else in the chapter, where $f = \mathcal{L}$ and $g = \mathcal{L}'(\theta)$ in the univariate case.

5.7.1 Classification by derivative information

Methods are most usefully organised by what they require of the objective.

Table 5.3: Organizes the optimization methods of this chapter by how much derivative information each one requires. Optimization methods classified by the derivative information required

Order	Requires	Methods	Typical use
0	Function values only	Nelder–Mead, simulated annealing, genetic algorithms	Non-smooth, discontinuous, or awkward-to-differentiate objectives; small p
1	Gradient	Steepest descent, conjugate gradient, quasi-Newton (BFGS, L-BFGS), BHHH, stochastic gradient descent (with momentum, Adam)	The workhorse class; large p; gradients cheap by automatic differentiation

Order	Requires	Methods	Typical use
2	Gradient and Hessian	Newton–Raphson, Fisher scoring, IRLS, trust-region Newton	Statistical models of modest dimension where the Hessian is also wanted for inference

5.7.2 Steepest Descent with Line Search

Move along $-g(\theta_k)$, the direction of steepest descent, choosing the distance by a one-dimensional search:

$$\theta_{k+1} = \theta_k - \alpha_k\, g(\theta_k), \qquad \alpha_k = \arg\min_{\alpha>0} f(\theta_k - \alpha\, g(\theta_k))\,. \tag{5.68}$$

Writing g_k for $g(\theta_k)$, the step is simply $\theta_{k+1} = \theta_k - \alpha_k g_k$: the search direction *is* the negative gradient, so it needs no name of its own.

The step length α_k can come from an **exact** line search — a univariate minimization by golden section or `optimize()` along the ray — or an **inexact** one: start at $\alpha = 1$ and halve until the Armijo sufficient-decrease condition $f(\theta_k - \alpha g_k) \le f(\theta_k) - c\,\alpha\, g_k' g_k$ holds. This is conceptually the simplest gradient method, and the one that best illustrates the role of a line search, but its convergence is only linear, and the rate deteriorates with the condition number of the Hessian: on an elongated valley the iterates zig-zag across it rather than travelling along it, as the next figure shows. It is rarely used unmodified.

```
grad_descent <- function(f, g, x0, maxit = 1000, tol = 1e-8) {
  path <- x0
  for (k in 1:maxit) {
    gk <- g(x0)
    a <- optimize(function(a) f(x0 - a * gk), c(0, 10))$minimum   # exact line search
    x1 <- x0 - a * gk; path <- rbind(path, x1)
    if (sqrt(sum((x1 - x0)^2)) < tol) break
    x0 <- x1 }
  list(par = x1, iter = k, path = path)
}
```

5.7.3 Why Gradient Descent Zig-Zags

With an exact line search, consecutive gradients are **orthogonal** — $g_{k+1}' g_k = 0$, since the search stops exactly where the directional derivative along $-g_k$ vanishes. On the elongated contours of the quadratic from Figure 5.17, that orthogonality forces the path to zig-zag back and forth across the valley instead of running down its length:

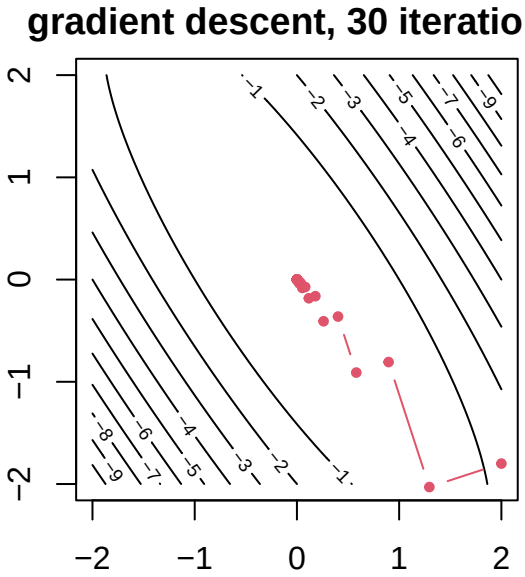


Figure 5.23: Illustrates the main weakness of gradient descent: on an elongated objective it makes slow zig-zag progress. Gradient descent with an exact line search zig-zags across an elongated valley: each step is orthogonal to the next.

The number of iterations needed grows with the **condition number** $\lambda_{\max}/\lambda_{\min}$ of the Hessian — precisely the elongation of the ellipses in the earlier figure.

5.7.4 Conjugate Gradient

The zig-zag is fixed by choosing each new direction to be **conjugate** to the previous ones with respect to the Hessian Q: $d_i' Q d_j = 0$ for $i \neq j$. For a quadratic $f(\theta) = \frac{1}{2}\theta' Q\theta - b'\theta$, minimizing along p such directions in succession reaches the exact minimum in **at most** p **steps**, without ever forming or storing Q — storage is $O(p)$, its defining advantage over Newton.

Here a separate symbol is warranted: the search direction d_k is no longer the negative gradient, but a correction of it. The nonlinear version builds these directions from gradients only:

$$d_0 = -g_0, \qquad d_{k+1} = -g_{k+1} + \beta_k d_k, \tag{5.69}$$

$$\beta_k^{\mathrm{FR}} = \frac{g_{k+1}' g_{k+1}}{g_k' g_k} \text{ (Fletcher–Reeves)}, \qquad \beta_k^{\mathrm{PR}} = \frac{g_{k+1}'(g_{k+1} - g_k)}{g_k' g_k} \text{ (Polak–Ribière)}, \tag{5.70}$$

with $\theta_{k+1} = \theta_k + \alpha_k d_k$ from a line search and $g_k = g(\theta_k)$ as before. Fletcher–Reeves and Polak–Ribière differ only in this correction coefficient. The method is sensitive to the accuracy of the line search, and conjugacy degrades on non-quadratic f, so it is usually restarted with $d = -g$ every p iterations.

```
conj_grad <- function(f, g, x0, maxit = 1000, tol = 1e-8, restart = length(x0)) {
  g0 <- g(x0); d <- -g0; path <- x0            # d: a conjugate direction, not -g
  for (k in 1:maxit) {
```

```
    a <- optimize(function(a) f(x0 + a * d), c(0, 10))$minimum
    x1 <- x0 + a * d; g1 <- g(x1); path <- rbind(path, x1)
    if (sqrt(sum(g1^2)) < tol) break
    beta <- if (k %% restart == 0) 0 else max(0, sum(g1 * (g1 - g0)) / sum(g0^2))   #
 ↪  Polak-Ribiere+
    d <- -g1 + beta * d; x0 <- x1; g0 <- g1 }
  list(par = x1, iter = k, path = path)
}
cg <- conj_grad(fq, gq, c(2, -1.8))
par(pty = "s", mar = c(4, 4, 2, 1))
contour(s, s, -z, nlevels = 15, main = sprintf("conjugate gradient, %d iterations", cg$iter))
lines(gd$path, type = "b", col = 2, pch = 19, cex = 0.6); lines(cg$path, type = "b", col = 4,
 ↪  pch = 19, lwd = 2)
legend("topleft", c("gradient descent", "conjugate gradient"), col = c(2, 4), lwd = 2, bty =
 ↪  "n")
```

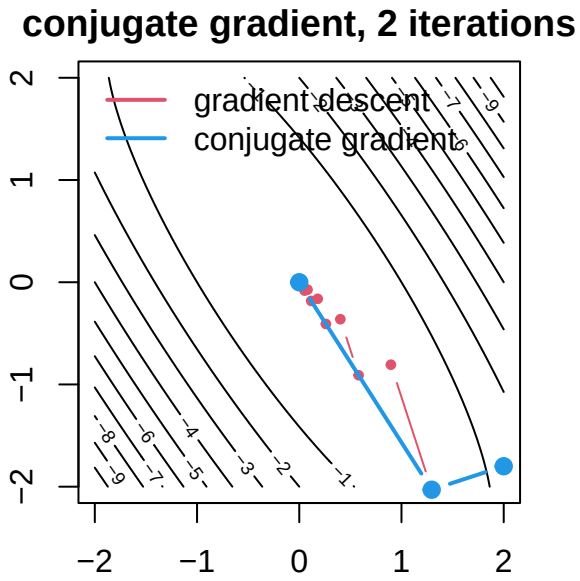


Figure 5.24: Shows how conjugate gradient repairs the zig-zag behaviour of gradient descent by choosing search directions that do not undo earlier progress. Conjugate gradient (blue) reaches the minimum of the same elongated quadratic in far fewer iterations than gradient descent (red).

5.7.5 Quasi-Newton: BFGS

BFGS sits between the gradient methods above and Newton's method: it builds an approximation $B_k \approx J(\theta_k)$ from successive gradient differences $s_k = \theta_{k+1} - \theta_k$, $y_k = g_{k+1} - g_k$, via a rank-two update that keeps B_k positive definite by construction (which is exactly why the Wolfe curvature condition of an earlier section matters: it is what guarantees $s_k' y_k > 0$, and hence a well-defined update). Steps are $\theta_{k+1} = \theta_k - \alpha_k B_k^{-1} g_k$ with a line search, giving **superlinear** convergence from only first derivatives — again a rescaling of g_k rather than g_k itself. BFGS itself stores a full $p \times p$ approximation; **L-BFGS** keeps only the last few update vectors, reducing storage to $O(mp)$ and making it the default choice for high-dimensional smooth problems. For most likelihood problems where the Hessian is awkward to derive, BFGS with an analytic gradient is the best practical compromise.

5.7.5.1 Example: Nelder–Mead, CG and BFGS on the Logistic Regression

Returning to the logistic-regression example, `optim()` can fit it with no Hessian at all — Nelder–Mead needs only the log-likelihood, CG and BFGS also take the analytic gradient already coded for Newton–Raphson:

```
negll_fn  <- function(b) neg_logp_logistic(b, data$x, data$y)
neggr_fn  <- function(b) nll_grad_info(b, data$y, data$x)$grad

fits <- list(NM   = optim(c(0, 0), negll_fn, method = "Nelder-Mead"),
             CG   = optim(c(0, 0), negll_fn, neggr_fn, method = "CG"),
             BFGS = optim(c(0, 0), negll_fn, neggr_fn, method = "BFGS", hessian = TRUE))
t(sapply(fits, function(f) c(f$par, f_evals = f$counts[1], g_evals = f$counts[2])))
```

```
                     f_evals.function g_evals.gradient
NM   -0.2726011 1.503158               65              NA
CG   -0.2725322 1.503424               25               9
BFGS -0.2725326 1.503425               31               9
```

```
sqrt(diag(solve(fits$BFGS$hessian)))     # SEs from the numerically computed Hessian at the
 ↪  optimum
```

```
[1] 0.1797399 0.2089537
```

All three land on the same estimate as `mle_logistic_nr` and `glm` above, and `optim(..., hessian = TRUE)` returns a finite-difference Hessian at the solution, so standard errors are available even for a method that never formed one during the iterations.

5.7.6 Stochastic Gradient Descent

5.7.6.1 Motivation: Large n

For iid data the negative log-likelihood and its gradient are **sums over observations**:

$$\mathcal{L}(\theta) = \sum_{i=1}^{n} \mathcal{L}_i(\theta), \qquad g(\theta) = \sum_{i=1}^{n} g^{(i)}(\theta), \qquad g^{(i)} = \nabla \mathcal{L}_i. \tag{5.71}$$

Every derivative-based method above needs the **full** gradient — an $O(np)$ pass through all the data — at every single iteration, and Newton needs $O(np^2)$ for the Hessian on top of that. With n in the millions and p in the millions, as in a large neural network, even one full gradient evaluation is expensive.

The idea behind stochastic gradient descent is that $g^{(i)}(\theta)$ for a randomly chosen i is an **unbiased estimate** of the average gradient,

$$E_i\big[g^{(i)}(\theta)\big] = \frac{1}{n} g(\theta), \tag{5.72}$$

so a noisy but cheap gradient estimate is often enough to make progress — the same Monte Carlo idea used elsewhere in this book, now applied to optimization rather than integration.

Notation. From here on a superscript in parentheses indexes cases or mini-batches and a subscript indexes iterations, as for θ_k: $g^{(i)}$ is the gradient of case i; $g^{(B)} = \frac{1}{m}\sum_{i \in B} g^{(i)}$ is the average over a mini-batch B of m cases; and g_k is the gradient used at iteration k, either the full gradient $g(\theta_k)$ or a mini-batch gradient $g^{(B_k)}(\theta_k)$.

5.7.6.2 The SGD Algorithm

For $t = 1, 2, \ldots$:

1. Sample an index i_t uniformly from $\{1, \ldots, n\}$ — or shuffle the data once and cycle through it, so that one full pass is called an **epoch**.
2. Update in the descent direction of that single observation's negative log-likelihood:

$$\theta_{t+1} = \theta_t - \gamma_t \, g^{(i_t)}(\theta_t). \tag{5.73}$$

Mini-batch SGD averages the gradient over a batch B_t of m observations (typically $m = 32$–512 in machine-learning practice, though far smaller batches suffice for the simple models in this book):

$$\theta_{t+1} = \theta_t - \gamma_t \, g^{(B_t)}(\theta_t), \qquad g^{(B_t)} = \frac{1}{m} \sum_{i \in B_t} g^{(i)}, \tag{5.74}$$

which reduces the variance of the gradient estimate by a factor of $1/m$ at m times the per-step cost, and is what lets the computation vectorize efficiently or run on a GPU. The **learning rate** γ_t plays the role that a line search played for the deterministic methods above — a line search itself would need a full pass over the data to evaluate $\mathcal{L}$, defeating the purpose.

5.7.6.3 Learning Rate and Convergence

Because each update uses a noisy gradient, a constant learning rate γ makes θ_t **bounce around** the optimum indefinitely, with a variance proportional to γ. The classical result of Robbins and Monro (1951) is that θ_t converges to the optimum provided

$$\sum_t \gamma_t = \infty \quad \text{and} \quad \sum_t \gamma_t^2 < \infty, \qquad \text{e.g. } \gamma_t = \frac{\gamma_0}{1 + t/t_0}, \tag{5.75}$$

the first condition allowing the iterate to travel arbitrarily far from its start, and the second forcing the noise to die out fast enough for it to settle.

A learning rate that is too large causes divergence or persistent noise; one that is too small is painfully slow. Convergence is only **sublinear** — far slower than Newton's quadratic rate *per iteration* — but each SGD iteration costs $O(p)$ or $O(mp)$ rather than $O(np)$ for a full gradient or $O(np^2)$ for a Hessian, and early progress is often fast even when final convergence is not. Common refinements are the **momentum** and **Adam** modifications of the gradient described next, which were developed for, or are now mostly used with, mini-batch gradients, and **Polyak–Ruppert averaging** of the iterates, $\overline{\theta}_T = \frac{1}{T} \sum_{t \le T} \theta_t$, which restores the $\sqrt{n}$-efficiency of the MLE that the noisy path alone does not have.

5.7.6.4 Why SGD?

Two different arguments are made for SGD, and they are not equally firm.

Cost per unit of progress. This is the established reason, and it holds for any loss that is a sum over many cases, convex or not. Large data sets are redundant: a mini-batch of a few hundred cases usually points in nearly the same direction as all n, so SGD takes many useful steps in the time a full-gradient method takes one. Since the MLE itself is accurate only to $O(n^{-1/2})$, there is little value in optimizing far below that statistical error, and for large n SGD reaches that accuracy with less computation than methods that need the full gradient (Bottou and Bousquet, 2008, "The Tradeoffs of Large Scale Learning").

The noise itself. The mini-batch gradient is the average gradient plus noise whose covariance, for m cases drawn with replacement, is

$$\operatorname{Var}\left(g^{(B)}\right) = \frac{1}{m}\Big(\frac{1}{n}\sum_{i=1}^{n} g^{(i)}g^{(i)\prime} - \bar{g}\,\bar{g}'\Big), \qquad \bar{g} = \frac{1}{n}\,g(\theta). \tag{5.76}$$

Near a minimum $\bar{g} \approx 0$, and the covariance is $\frac{1}{nm}$ times the empirical Fisher information of Section 5.7.6.7, which under a correctly specified model approximates the Hessian. The noise is therefore largest along the directions of highest curvature, and its overall size is set by the ratio γ/m of learning rate to batch size. For non-convex losses this noise has been argued to help in three ways:

- it moves the iterate away from saddle points, where exact gradient descent can stall (Ge, Huang, Jin and Yuan, 2015; Jin et al., 2017);
- SGD behaves roughly like gradient descent on a smoothed version of the loss, in which small, sharp local minima disappear (Kleinberg, Li and Yuan, 2018);
- for neural networks, small-batch training tends to end in wide, flat minima, which generalize better than the sharp minima reached with large batches (Keskar et al., 2017).

The first is proved, under conditions on the saddle points. The other two are supported by analyses under simplifying assumptions and by experiments, but they remain debated: large-batch training with a rescaled learning rate (Goyal et al., 2017), or full-batch gradient descent with explicit regularization (Geiping et al., 2022), can match small-batch SGD, and how flat a minimum is depends on how the network is parameterized (Dinh et al., 2017). The safest summary is that in neural networks SGD's noise often acts as an implicit regularizer, by a mechanism that is not yet settled.

For a concave log-likelihood such as the logistic regression fitted above by Newton–Raphson, there is a single optimum, the MLE, and only the first argument applies. The noise then adds variance around the MLE rather than helping to find it, and Newton–Raphson or BFGS remain the better choices at such sizes. SGD earns its place on neural networks, where the loss is non-convex and n and p are large.

5.7.6.5 Momentum

At iteration k let B_k be the mini-batch and $g^{(B_k)} = g^{(B_k)}(\theta_k)$ its gradient at the current iterate, the average of the case gradients $g^{(i)}$, $i \in B_k$ (superscript: which batch; subscript: which iteration). This gradient is noisy, and the iterates also zig-zag across valleys. **Momentum** (the heavy-ball method) modifies the gradient by *averaging* it over

past iterations, which damps both the noise and the zig-zag: it keeps an exponentially weighted sum m_k of past gradients and steps along it,

$$m_k = \beta\, m_{k-1} + g^{(B_k)}, \qquad \theta_{k+1} = \theta_k - \alpha\, m_k, \tag{5.77}$$

where $\beta \in [0, 1)$, commonly 0.9, sets the memory. The across-valley components change sign from step to step and cancel in the average, while the along-valley components keep their sign and add up, so the step lengthens along the valley floor. With $\beta = 0$ it reduces to gradient descent, and the effective step along a persistent direction is $\alpha/(1-\beta)$, which is why α is usually reduced when β is raised. Only the gradient is modified; the preconditioner is still the identity and no curvature information is used. On a deterministic full-data problem momentum has little to offer: conjugate gradient is the optimal form of the same idea for a quadratic and needs no β, and BFGS or Newton–Raphson do better still. It is a modifier designed for SGD.

5.7.6.6 Adam

Momentum fixes the direction but not the scale: one step size α serves every coordinate, although the coordinates of a network can have gradients of very different sizes. **Adam** (adaptive moment estimation; Kingma and Ba, 2014, "A Method for Stochastic Optimization") adds a step size for each coordinate, estimated from the gradients themselves. Besides the averaged gradient it keeps an exponential average of the elementwise squared gradient,

$$m_k = \beta_1 m_{k-1} + (1-\beta_1)\, g^{(B_k)}, \qquad s_k = \beta_2 s_{k-1} + (1-\beta_2)\, g^{(B_k)} \odot g^{(B_k)}, \tag{5.78}$$

started at $m_0 = s_0 = 0$. Both averages are corrected for that start, $\hat{m}_k = m_k/(1-\beta_1^k)$ and $\hat{s}_k = s_k/(1-\beta_2^k)$, and the update is

$$\theta_{k+1} = \theta_k - \alpha\, \frac{\hat{m}_k}{\sqrt{\hat{s}_k} + \varepsilon}, \tag{5.79}$$

with the usual defaults $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-8}$.

The update needs only the batch gradient $g^{(B_k)}$: s_k is a running average of the squared gradient **over iterations**. Unrolling the recursion shows why the zero start does no harm: $\hat{s}_k = \sum_{j\le k} \beta_2^{k-j} g_j^2 \big/ \sum_{j\le k} \beta_2^{k-j}$ is a proper weighted average of the squared gradients seen so far, with no trace of s_0; in particular the first step is $\alpha\, g_1/(|g_1| + \varepsilon) \approx \alpha\ \mathrm{sign}(g_1)$ in each coordinate. Dividing by $\sqrt{\hat{s}_k}$ makes each coordinate move roughly α per step whatever the scale of f: coordinates with large gradients take smaller steps and flat ones larger steps, and the iteration becomes far less sensitive to how the parameters are scaled. Adam is momentum with a step size for each coordinate: with s_k held constant it is exactly momentum, so it contains the averaged gradient and adds the rescaling. It is the default optimizer for neural networks. As with momentum, its value lies in SGD: on a deterministic full-batch problem it rescales coordinates but uses no curvature, and gradient descent with a line search, conjugate gradient, BFGS or Newton–Raphson are better choices.

5.7.6.7 Comments on the Use of Squared Gradients for Rescaling in Adam

AdaGrad, RMSProp and Adam divide the gradient, coordinate by coordinate, by the square root of a running average of its own squared values:

Table 5.4: Adaptive step-size methods built from the history of the batch gradients.

Method	Running average s_k of $\left(g^{(B_j)}\right)^2$ (elementwise)	Step
AdaGrad (Duchi, Hazan and Singer, 2011)	cumulative sum $\sum_{j\le k}\left(g^{(B_j)}\right)^2$	$\alpha\, g^{(B_k)}/\sqrt{s_k}$
RMSProp (Hinton, 2012)	exponential average $\beta_2 s_{k-1}+(1-\beta_2)\left(g^{(B_k)}\right)^2$	$\alpha\, g^{(B_k)}/\sqrt{s_k}$
Adam (Kingma and Ba, 2014)	as RMSProp, bias-corrected	$\alpha\,\hat{m}_k/\sqrt{\hat{s}_k}$, with momentum

AdaGrad's cumulative sum grows without bound, so its steps keep shrinking; RMSProp and Adam forget old gradients and do not. Why this rescaling helps has been explained from several perspectives. They are described below, with their limits, and none of them accounts for all of the behaviour seen in practice.

Perspective 1: the empirical Fisher information. Adam's authors remark that $\hat{s}_k$ approximates the diagonal of the Fisher information (Kingma and Ba, 2015, citing Pascanu and Bengio, 2013), which suggests reading the method as a diagonal version of natural gradient. To see what this does and does not mean, recall the empirical Fisher information of this chapter.

For a loss that is a sum over cases, $\mathcal{L}=\sum_i \mathcal{L}_i$, the per-case gradients $g^{(i)}=\nabla\mathcal{L}_i$ define the **empirical Fisher information**

$$\hat{F}(\theta)=\sum_{i=1}^{n} g^{(i)}(\theta)\, g^{(i)}(\theta)'. \tag{5.80}$$

Connection to the Fisher information. Each case's score has mean zero at the true parameter, so the Fisher information of the n observations is

$$I_n(\theta)=nI(\theta)=\sum_{i=1}^{n} E\big[g^{(i)}g^{(i)\prime}\big]=E\big[J(\theta)\big] \tag{5.81}$$

by the information equality. $\hat{F}$ replaces the expectation by the observed scores: it estimates $I_n(\theta)$ from **first derivatives only**, and it is positive semidefinite by construction, whatever θ.

BHHH. Using $\hat{F}$ as the curvature in a Newton step gives the **BHHH** algorithm (Berndt, Hall, Hall and Hausman, 1974), also called the outer-product-of-gradients method,

$$\theta_{k+1}=\theta_k-\alpha_k\,\hat{F}(\theta_k)^{-1}g_k, \qquad \alpha_k \text{ from a line search}, \tag{5.82}$$

beside Newton–Raphson, which uses the observed curvature J, and Fisher scoring, which uses I_n. Every step is a descent direction, and $\hat{F}(\hat{\theta})^{-1}$ at the optimum is the **OPG estimate of the variance**, which with the Hessian also forms the sandwich estimator $J^{-1}\hat{F}J^{-1}$. BHHH is standard in econometrics (`maxLik(..., method = "BHHH")` in R). It needs per-case gradients, and $\hat{F}$ can be a poor stand-in for I_n far from the MLE or when the model is misspecified.

The diagonal. $\hat{F}$ is a $p\times p$ matrix: $O(p^2)$ storage and an $O(p^3)$ solve, impossible for a network with p in the millions. Its **diagonal** is only p numbers, the sum over cases of the squared gradients,

$$\big[\operatorname{diag}\hat{F}\big]_r=\sum_{i=1}^{n}\Big(\frac{\partial\mathcal{L}_i}{\partial\theta_r}\Big)^2, \qquad \text{that is,} \quad \operatorname{diag}\hat{F}=\sum_{i=1}^{n} g^{(i)}\odot g^{(i)}, \tag{5.83}$$

a curvature scale for each coordinate θ_r, computed from gradients that are already available, in $O(p)$ storage.

Adam's $\hat{s}_k$ looks like diag $\hat{F}$, but the resemblance is loose:

- s_k squares the *batch average* $g^{(B_k)}$; it does not sum the squares of the case gradients. For a batch of m cases drawn at a fixed θ, with $\bar{g} = \frac{1}{n}\sum_i g^{(i)}$,

$$E\big[g^{(B)} \odot g^{(B)}\big] = \bar{g}^2 + \frac{1}{m}\Big(\frac{1}{n}\operatorname{diag}\hat{F} - \bar{g}^2\Big), \tag{5.84}$$

  so it contains the squared mean gradient, which is the signal and not the noise. Only near the optimum, where $\bar{g} \approx 0$, and if θ changes little over the averaging window of about $1/(1-\beta_2) = 1000$ iterations, is $\hat{s}_k \approx \frac{1}{nm}\operatorname{diag}\hat{F}$. With full-batch gradients s_k is just the squared mean gradient, which tends to zero at the optimum.
- Even where $\hat{s}_k$ does estimate diag $\hat{F}$, $\hat{F}$ stands for curvature only when the model is correctly specified ($\hat{F} \approx I_n \approx J$). Otherwise, and far from the optimum, it is a poor substitute for curvature (Kunstner, Hennig and Balles, 2019).
- A Newton or scoring step multiplies by the *inverse* $\hat{F}^{-1}$, a power -1, whereas these methods use the power $-\frac{1}{2}$. With $\hat{s} \propto g^2$ a power of -1 would give the step α/g, which explodes where the gradient is small. Kingma and Ba note this themselves, calling the square root "more conservative" than natural gradient.

Methods that really do approximate the Fisher information of a network include **K-FAC** (Martens and Grosse, 2015), one Kronecker-factored block per layer, and **natural gradient** (Amari, 1998).

Perspective 2: a signal-to-noise ratio. Kingma and Ba's own explanation is that $\hat{m}_k/\sqrt{\hat{s}_k}$ is a signal-to-noise ratio. When the gradient direction is uncertain, $\hat{m}_k$ is small relative to $\sqrt{\hat{s}_k}$ and the step shrinks, which they call "a form of automatic annealing" near an optimum. The step is also bounded by about α and is invariant to the scale of the gradient: multiplying g by a constant c cancels in $\hat{m}_k/\sqrt{\hat{s}_k}$, so rescaling the parameters or the loss does not change the iteration. This explains the robustness to scale that makes Adam easy to use. It does not by itself show an improvement in the rate of convergence.

Perspective 3: online learning. AdaGrad was derived from a regret bound in online convex optimization (Duchi, Hazan and Singer, 2011): the per-coordinate step $1/\sqrt{\sum_j g_j^2}$ is the best fixed scaling in hindsight, and it gives rates that adapt to sparse gradients. The bound is for AdaGrad, not for RMSProp or Adam, which forget old gradients. The convergence proof in the original Adam paper was later shown to be wrong, with simple convex problems on which Adam does not converge, and the variant AMSGrad was proposed (Reddi, Kale and Kumar, 2018).

Perspective 4: sign descent. Dividing a gradient by its magnitude leaves its sign. With $\beta_1 = \beta_2 = 0$, Adam is sign descent, $\theta_{k+1} = \theta_k - \alpha\,\operatorname{sign}(g)$, which is steepest descent in the max-norm (Bernstein et al., 2018). Balles and Hennig (2018) split Adam's update into a sign and a magnitude, with the magnitude being a variance-based damping of the sign. On transformers, sign descent without the averaging recovers much of Adam's advantage over SGD (Kunstner et al., 2023).

Perspective 5: heterogeneous noise and curvature. Two explanations for why adaptive methods beat SGD on attention models are heavy-tailed gradient noise, where the per-coordinate normalization acts like clipping (Zhang et al., 2020), and a Hessian whose blocks differ strongly from one parameter block to another, which a single learning rate cannot suit (Zhang et al., 2024). Kunstner et al. (2023) find that noise is not the main factor behind the gap between Adam and SGD on transformers, so these accounts compete.

Practical benefits. Whatever the explanation, the benefits are not in doubt. Adam needs little tuning, since its default α, β_1, β_2 work across a wide range of problems, it is insensitive to the scale of the parameters and the loss, it copes with coordinates whose gradients differ by orders of magnitude, and it costs only $O(p)$ extra storage. It, or its variant AdamW, is the default for training transformers and many other deep networks.

An open question. There is still no theory that explains why this rescaling helps so much for deep networks, and the Fisher interpretation in particular is not supported. What can be said with some confidence is what the rescaling does (it equalizes the movement of coordinates and is invariant to scale) and that it is effective empirically. Which of the properties above is responsible for its advantage over SGD, and in which problems, remains to be explored.

5.7.7 Shinylive App for Comparing the Optimizers

There is a shinylive app to compare the paths that gradient descent, momentum, Adam, conjugate gradient, BFGS and safeguarded Newton take on the same objective, on six test functions from a quadratic to the Rosenbrock banana and Rastrigin's many local minima. The app uses exact gradients, a deterministic setting in which momentum and Adam are not at their best, which is the point of the comparison: curvature information (BFGS, Newton) and line searches (CG, BFGS) pay off there, while momentum needs a learning rate suited to each function and diverges when it is too large, and Adam's single default $\alpha = 0.1$ works on all six but is slow along curved valleys.

5.8 Examples of Fitting Neural Networks (Shinylive App Illustration)

Both examples below use one input x and four `tanh` hidden units, so that θ has 13 weights and the fit can be drawn as a curve. They differ only in the likelihood, hence in the output link and in the starting value δ_{η_i} of backpropagation (Section 5.6.5); every other step of the gradient computation is the same. Each app fits the same data with seven optimizers: fixed-rate gradient descent, mini-batch SGD, momentum, Adam, gradient descent with a Wolfe line search, conjugate gradient and BFGS. One SGD iteration is an epoch, so that its progress can be compared with the full-batch methods.

5.8.1 Gaussian Neural Network

For $y_i = \mu(x_i) + \varepsilon_i$ with $\varepsilon_i \sim N(0, \sigma^2)$ the mean is the network output itself (identity link), $\mu_i = \eta_i = \sum_{j=0}^{4} v_j h_{ij}$, and the loss is half the mean squared error,

$$\mathcal{L}(\theta) = \frac{1}{2n} \sum_{i=1}^{n} (y_i - \mu_i)^2, \qquad \delta_{\eta_i} = \mu_i - y_i. \tag{5.85}$$

The factor $\frac{1}{2}$ changes neither the minimizer nor the shape of the loss. It is there because it makes the loss the Gaussian negative log-likelihood per observation, $-\log N(y_i \mid \mu_i, \sigma^2) = \frac{(y_i - \mu_i)^2}{2\sigma^2} + \text{const}$, with $\sigma^2 = 1$ and the constant dropped, and because it cancels the 2 produced by differentiating the square, so that $\delta_{\eta_i} = \mu_i - y_i$ without a factor. Its minimum is not zero but about $\sigma^2/2$, the **noise floor**, which the app draws as a dashed line. The model is non-convex: the four units can be permuted, and several of them can end up doing nearly the same work, so different optimizers and starting weights may reach different, equally good fits, and only some find a good one quickly. The Gaussian app lets you set the noise level σ, draws the weights and their gradient as networks, and shows on the data scale how the four scaled hidden units add up to the fitted mean.

5.8.2 Poisson Neural Network

With a log link, $\lambda_i = e^{\eta_i}$, and counts $y_i \mid x_i \sim \text{Poisson}(\lambda_i)$, the loss is $\mathcal{L} = \frac{1}{n}\sum_i(\lambda_i - y_i\eta_i)$ and $\delta_{\eta_i} = \lambda_i - y_i$; this is the model whose gradient is worked out in full in the backpropagation section above. The log link makes the hidden units add on the log scale, so $\lambda(x)$ is a *product* of their contributions $e^{v_j h_j(x)}$, and the loss is much more sensitive to a weight that is too large than in the Gaussian case, so a fixed learning rate is harder to choose. The Poisson app offers the same controls as the Gaussian one, with the sample size going up to 2000 so that mini-batch SGD can be compared with the full-batch methods when n is large.

5.9 Examples of Fitting NN with Tensorflow

The Gaussian network with a linear output (identity link) written in Keras, now for a mean function of **two predictors with four modes**: $\mu(x_1, x_2)$ is a sum of four Gaussian bumps of different heights centred at $(\pm 1.5, \pm 1.5)$,

$$\mu(x_1, x_2) = \sum_{k=1}^{4} h_k \exp\Big(- \frac{(x_1 - c_{k1})^2 + (x_2 - c_{k2})^2}{2 \cdot 0.8^2} \Big), \tag{5.86}$$

with centres $c_1 = (-1.5, -1.5)$, $c_2 = (-1.5, 1.5)$, $c_3 = (1.5, -1.5)$, $c_4 = (1.5, 1.5)$ and heights $h = (2.0, 1.5, 1.0, 1.8)$, and $y = \mu(x_1, x_2) + \varepsilon$ with $\varepsilon \sim N(0, 0.2^2)$, so that the smallest attainable loss is the noise floor $\sigma^2 = 0.2^2 = 0.04$. The network has one hidden layer of 16 `tanh` units and a linear output, that is two layers of weights and $2 \cdot 16 + 16 + 16 + 1 = 65$ parameters. The data are split into a training and a test set; the network is fitted to the training set only, and the test set shows how well it predicts new observations.

Fitting Neural Networks with Tensorflow

```
import os
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"          # quiet TensorFlow's start-up messages
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt


tf.keras.utils.set_random_seed(1)                 # numpy, python and TensorFlow seeds

# the true mean: four bumps of different heights, centred at (+-1.5, +-1.5)
centres = np.array([[-1.5, -1.5], [-1.5, 1.5], [1.5, -1.5], [1.5, 1.5]])
heights = np.array([2.0, 1.5, 1.0, 1.8])
def mu_true(x):
    d2 = ((x[:, None, :] - centres[None]) ** 2).sum(axis=-1)       # n x 4 squared distances
    return (heights * np.exp(-d2 / (2 * 0.8 ** 2))).sum(axis=1, keepdims=True)

# data: y = mu(x1, x2) + noise
rng = np.random.default_rng(1)
n, sigma = 1500, 0.2
x = rng.uniform(-3, 3, size=(n, 2)).astype("float32")
y = (mu_true(x) + sigma * rng.normal(size=(n, 1))).astype("float32")

# train / test split: the first 1200 observations fit the network, the last 300 test it
```

```
ntr = 1200
x_tr, y_tr, x_te, y_te = x[:ntr], y[:ntr], x[ntr:], y[ntr:]

# the model: eta = v_0 + sum_j v_j tanh(w_j1 x_1 + w_j2 x_2 + w_j0), identity link
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(2,)),
    tf.keras.layers.Dense(16, activation="tanh"),   # hidden layer: W (2 x 16), w_0 (16)
    tf.keras.layers.Dense(1),                       # output layer: v (16 x 1), v_0
])

# the loss is chosen separately from the model. Keras calls it as loss(y_true, y_pred):
#   y_true = a mini-batch of the targets y_tr passed to fit()
#   y_pred = the output of the last layer of the network for the same rows, here eta = mu;
#            Keras does not check that this suits the loss, so the output layer and the
#            loss have to be chosen together. The names y, mu are free.
# the mean squared error, the Gaussian negative log-likelihood up to a constant
square_loss = lambda y, mu: tf.reduce_mean(tf.square(y - mu))

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.01), loss=square_loss)

# fit(x, y): x is fed to the network, y is held back as the targets of the loss; the rows
# of x_tr and y_tr are shuffled together and cut into mini-batches of 50 observations
history = model.fit(x_tr, y_tr, batch_size=50, epochs=600, verbose=0)   # one epoch = 1200/50
 ↪  mini-batch steps

print("training loss:", history.history["loss"][-1], "  noise floor:", sigma**2)
```

```
training loss: 0.04587170109152794   noise floor: 0.04000000000000001
```

```
print("number of weights:", model.count_params())
```

```
number of weights: 65
```

Evaluating the Fitting by Tensorflow

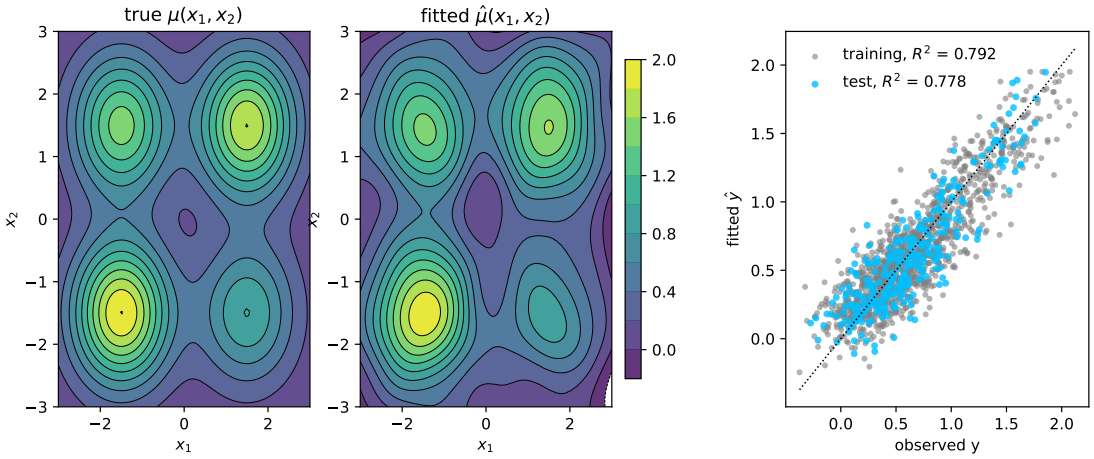
```
# fitted values, and R^2 = 1 - SSE / SST on each set
r2 = lambda y, yhat: 1 - np.sum((y - yhat)**2) / np.sum((y - y.mean())**2)
yhat_tr = model.predict(x_tr, verbose=0)
yhat_te = model.predict(x_te, verbose=0)

# the true and the fitted mean on a grid, drawn with the same contour levels
g = np.linspace(-3, 3, 100, dtype="float32")
G1, G2 = np.meshgrid(g, g)
grid = np.column_stack([G1.ravel(), G2.ravel()])
Z_true = mu_true(grid).reshape(G1.shape)
Z_fit = model.predict(grid, verbose=0).reshape(G1.shape)
levels = np.linspace(-0.2, 2.0, 12)

fig, ax = plt.subplots(1, 3, figsize=(12, 4.4))
for a, Z, title in [(ax[0], Z_true, r"true $\mu(x_1, x_2)$"), (ax[1], Z_fit, r"fitted
 ↪  $\hat\mu(x_1, x_2)$")]:
```

```
    cs = a.contourf(G1, G2, Z, levels=levels, cmap="viridis", alpha=0.85)
    a.contour(G1, G2, Z, levels=levels, colors="k", linewidths=0.5)
    a.set_title(title); a.set_xlabel(r"$x_1$"); a.set_ylabel(r"$x_2$")
cbar = fig.colorbar(cs, ax=ax[:2], shrink=0.85, pad=0.02)   # assigned so that the object is
 ↪  not printed

# right: fitted value against observed response
ax[2].scatter(y_tr, yhat_tr, s=8, color="grey", alpha=0.6,
              label=f"training, $R^2$ = {r2(y_tr, yhat_tr):.3f}")
ax[2].scatter(y_te, yhat_te, s=12, color="deepskyblue", alpha=0.8,
              label=f"test, $R^2$ = {r2(y_te, yhat_te):.3f}")
lim = [y.min(), y.max()]
ax[2].plot(lim, lim, "k:", lw=1)
ax[2].set_xlabel("observed y"); ax[2].set_ylabel(r"fitted $\hat{y}$")
ax[2].legend(frameon=False, loc="upper left")
plt.show()
```



In Keras the model only defines the network, $x \mapsto \eta(x;\theta)$; the response enters through the `loss` given to `compile`. Choosing the squared-error loss $(y-\mu)^2$ is what makes this a Gaussian model with an identity link. For counts one would end the network with the same linear layer and use the loss $e^{\eta} - y\eta$ of the Poisson model, so the output layer and the loss have to be chosen consistently by the user. Everything the chapter did by hand happens inside `fit`: the forward pass computes μ_i, backpropagation produces the gradient of the mini-batch loss, and Adam turns that gradient into the update $\theta_{k+1} = \theta_k - \alpha_k P_k m_k$ of Table 5.5. The optimizer is the only line to change in order to repeat the comparison of the apps: `SGD(learning_rate=0.1)`, `SGD(learning_rate=0.1, momentum=0.9)` or `Adam(0.01)`. Setting `batch_size=ntr` makes each step a full-batch one. The loss is not convex, so different random initial weights can end in different, equally good fits. The R^2 cannot reach 1 even for the true mean, because the noise variance 0.04 is part of the variance of y; the contour plot is the better check that all four modes have been found. The test R^2 is close to the training one because the network has 65 weights for 1200 observations; a network with many more weights would show the gap that the test set exists to reveal.

The same network gives a Poisson regression by changing only the loss. The last layer is still a linear `Dense(1)`, whose output is now the log-mean $\eta_i = \log \lambda_i$, so the loss must exponentiate it:

```
# Poisson negative log-likelihood per observation, dropping log(y!):  lambda - y * eta,
# with lambda = exp(eta); here the second argument is eta = log(lambda), not lambda
poisson_nll = lambda y, eta: tf.reduce_mean(tf.exp(eta) - y * eta)

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.01), loss=poisson_nll)
model.fit(x_tr, y_tr, batch_size=50, epochs=600, verbose=0)    # y_tr now holds counts

lambda_hat = np.exp(model.predict(x_te, verbose=0))            # predictions are on the eta
 ↪  scale
```

Keras also ships such losses, for example `tf.keras.losses.Poisson`, but that one expects the *mean* λ as its second argument and uses $\lambda - y \log \lambda$, so it would need an `exp` activation on the last layer. Writing the loss on the η scale as above is numerically safer and matches the loss $\lambda_i - y_i \eta_i$ of the chapter. The code above is shown, not run.

5.10 Remarks on Implementation of Optimization Algorithms

The chapter has covered a lot of ground, from bisection on a scalar gradient g to stochastic gradient descent on millions of parameters. This section collects the gradient-based methods, Newton–Raphson included, into one reference: first as a single update rule in which they differ only in the gradient they use and in how it is modified, then by what each needs and costs, then by which R function to reach for and how to choose among them. Derivative-free methods such as Nelder–Mead are left out.

5.10.1 One Form for All the Methods

Every method of this chapter that uses the gradient can be written as the same update,

$$\theta_{k+1} = \theta_k - \alpha_k \, P_k \, m_k, \tag{5.87}$$

where $g_k = \nabla f(\theta_k)$ is the gradient of the objective $f = \mathcal{L}$ at the current iterate, and the three remaining ingredients are what distinguishes the methods:

- m_k is the **gradient used**: the gradient itself, a sampled version of it (mini-batches), or a smoothed version of it (a running average of past gradients);
- P_k is the **modifier** or *preconditioner*, a matrix that rescales or rotates the step, built from curvature, from Fisher information, or from the history of gradients, with $P_k = I$ when there is none;
- α_k is the **step size**: set by hand, found by a line search, or fixed at 1.

Table 5.5: Writes every gradient-based method of this chapter as $\theta_{k+1} = \theta_k - \alpha_k P_k m_k$, showing the gradient each one uses, how it modifies it, and how the step size is chosen. The common form of the gradient-based methods.

Method	Gradient used, m_k	Modifier, P_k	Step size, α_k
Steepest descent, fixed rate	g_k	I	α, set by hand

Method	Gradient used, m_k	Modifier, P_k	Step size, α_k
Steepest descent, line search	g_k	I	α_k from a Wolfe line search
Momentum	$v_k = \beta v_{k-1} + g_k$, a weighted sum of past gradients	I	α, set by hand
Adam	$\widehat{m}_k$, a bias-corrected exponential average of past gradients	$\mathrm{diag}\big(1/(\sqrt{\hat{s}_k} + \varepsilon)\big)$, with $\hat{s}_k$ the running average of $g^{(B_j)} \odot g^{(B_j)}$	α, set by hand
Conjugate gradient	$d_k = -g_k + \beta_k d_{k-1}$, the gradient combined with the previous direction	implicit: successive directions conjugate with respect to the Hessian	α_k from a line search
BFGS, L-BFGS	g_k	$B_k \approx J(\theta_k)^{-1}$, built from $s_k = \theta_{k+1} - \theta_k$ and $y_k = g_{k+1} - g_k$; L-BFGS keeps only the last m pairs	α_k from a Wolfe line search
Newton–Raphson	g_k	$J(\theta_k)^{-1}$, the inverse observed information	1, or step-halving as a safeguard
Fisher scoring, IRLS	g_k	$I(\theta_k)^{-1}$, the inverse expected information	1
Stochastic gradient descent	$g_k^B = \frac{1}{m}\sum_{i \in B_k} \nabla \mathcal{L}_i(\theta_k)$ on a random mini-batch, with $E\, g_k^B = g_k$	I	α_k decreasing, set by hand

Read down the table and the progression of the chapter appears. Steepest descent uses the gradient as it is. Conjugate gradient changes *which direction* is used, by combining the gradient with the past direction, which damps the zig-zag across a valley. Momentum does so for SGD by averaging the gradient over the past, and Adam also changes the *scale* of each coordinate with a diagonal estimate built from the gradients alone. BFGS learns a full inverse-Hessian approximation from gradient differences, and Newton–Raphson and Fisher scoring compute that matrix from second derivatives, which is why they alone take unit steps and converge quadratically near the solution. Stochastic gradient descent leaves P_k alone and changes the gradient instead, to a cheap and noisy one, and momentum and Adam can be applied to it unchanged. The same table is the one used in the descriptions of the two neural-network apps.

5.10.2 Comparison

Table 5.6: Summarizes the trade-offs among the gradient-based methods of this chapter: what each needs, what it costs, and how fast and robust it is. Comparison of gradient-based optimization methods. p is the number of parameters and m the L-BFGS memory length (typically 5–20).

Method	Derivatives	Storage	Cost per iteration	Local rate	Robust start	Hessian for SEs
Steepest descent	gradient	$O(p)$	1 gradient + line search	linear (slow)	good	no
Momentum	gradient	$O(p)$	1 gradient + $O(p)$	linear (faster along valleys)	good	no
Adam	gradient	$O(p)$	1 gradient + $O(p)$	sublinear, noisy near the optimum	good	no
Conjugate gradient	gradient	$O(p)$	1 gradient + line search	linear (fast)	moderate	no
BFGS	gradient	$O(p^2)$	1 gradient + $O(p^2)$	superlinear	moderate	approximate
L-BFGS	gradient	$O(mp)$	1 gradient + $O(mp)$	superlinear	moderate	no
Newton–Raphson	gradient, Hessian	$O(p^2)$	1 Hessian + $O(p^3)$	quadratic	poor	exact
Fisher scoring / IRLS	gradient, expected information	$O(p^2)$	1 weighted LS fit	near-quadratic	fair	exact
Stochastic gradient	minibatch gradient	$O(p)$	$O(\text{batch})$	sublinear	good	no

5.10.3 Software Packages for Optimization

The methods of this chapter are rarely coded by hand. The packages below are the commonly used ones, grouped by the job they do: general-purpose numerical optimizers, statistical model fitting, and automatic differentiation with mini-batch optimizers for neural networks.

Table 5.7: Lists the commonly used software for the optimization methods of this chapter, in R and in Python, with the methods each offers. Software packages for optimization

Language	Package, function	Methods available	Notes
R	`optim` (stats)	BFGS, CG, L-BFGS-B	General workhorse; supply `gr` or the gradient is differenced; `hessian = TRUE` for inference; L-BFGS-B allows box constraints

Language	Package, function	Methods available	Notes
R	`nlm` (stats)	Newton-type with ridging	Uses the analytic gradient and Hessian if attached as attributes, otherwise numerical; returns `$hessian`
R	`nlminb` (stats)	Trust-region (PORT)	Reliable on badly scaled problems; supports box constraints
R	`optimize`, `uniroot` (stats)	Golden section, Brent	One dimension only
R	`glm` (stats)	IRLS (Fisher scoring)	The specialised case for generalized linear models; use `start` when separation is suspected
R	`nloptr`	Large NLopt collection	Gradient-based and global methods, with constraints
R	`torch`	SGD, momentum, Adam, L-BFGS	Automatic differentiation and mini-batch training of neural networks
Python	`scipy.optimize`	`minimize` with BFGS, L-BFGS-B, CG, Newton-CG, trust-region methods; `root`; `minimize_scalar`	The standard general-purpose library; the counterpart of `optim` and `nlm`
Python	`statsmodels`	IRLS for GLMs, Newton–Raphson and BFGS for discrete-choice and other likelihood models	Returns standard errors from the Hessian
Python	`scikit-learn`	`LogisticRegression` solvers `lbfgs`, `newton-cg`, `sag`, `saga`; `SGDClassifier`, `SGDRegressor`	Fitting of standard models; the optimizer is chosen by an argument
Python	PyTorch (`torch.optim`)	SGD (with momentum), Adam, AdamW, RMSprop, L-BFGS	Gradients by `autograd`; the training loop is written explicitly
Python	TensorFlow / Keras (`tf.keras.optimizers`)	SGD (with momentum), Adam, AdamW, RMSprop, Adagrad	Gradients by `tf.GradientTape`; `model.fit` runs the mini-batch loop; BFGS and L-BFGS in TensorFlow Probability (`tfp.optimizer`)

Language	Package, function	Methods available	Notes
Python	JAX with Optax	SGD, momentum, Adam, AdamW; `jax.scipy.optimize.minimize` for BFGS	`jax.grad` differentiates ordinary numerical functions; the optimizers are composable

For small and moderate likelihood problems the first rows are all that is needed: supply the negative log-likelihood and, if possible, its gradient, and let BFGS or a Newton-type method do the rest. Fitting a neural network is different. The gradient is obtained by automatic differentiation, which carries out the backpropagation of Section 5.6.5 for any model written with the library's operations, and the update is applied to mini-batches by one of the optimizers of this chapter. **TensorFlow** and **PyTorch** are the two dominant libraries for this, and the Keras interface of TensorFlow makes the choice of optimizer a single argument. The Poisson network of this chapter, with the loss $\lambda_i - y_i \eta_i$ and four hidden units, is fitted by Adam as follows.

5.10.4 Optimization in Practice for Deep Learning

Fitting a deep network is the setting for which the methods of this section are used at the largest scale, with p from millions to hundreds of billions and n so large that a full pass through the data is rarely made. The practice differs from the small examples of this chapter in four ways.

- **Only mini-batch gradients.** The gradient comes from backpropagation (Section 5.6.5) on a mini-batch, and it is the only derivative information used. Full-batch methods, and second-order methods such as Newton–Raphson or BFGS, need a full pass or $O(p^2)$ storage, which is out of reach. Their place is taken by the diagonal rescaling of the Adam family (Table 5.4), and in research by cheaper approximations of curvature such as Shampoo and K-FAC.
- **Adam-type methods as the default.** AdamW, which is Adam with the weight decay applied directly to the weights rather than through the gradient, is the usual choice for transformers and language models, mainly because it needs little tuning and is robust to badly scaled weights. SGD with momentum, usually with a carefully tuned schedule and weight decay, remains common for image classification, where it is often reported to generalize slightly better; this is a rule of thumb that depends on the model and the data, not a theorem.
- **The learning-rate schedule matters as much as the optimizer.** A typical recipe is a short warmup followed by a cosine or linear decay of α_k, in the spirit of the Robbins–Monro condition above. Gradient clipping, large batches with a rescaled learning rate, and mixed-precision arithmetic are further standard ingredients.
- **The loss is not the only goal.** The objective is highly non-convex, with many equivalent minima, and a loss driven to its exact minimum can overfit. Practitioners stop early, regularize, and judge the fit on held-out data, so convergence to a stationary point is rarely checked in the sense used earlier in this chapter.

The two apps in the next section are deliberately tiny, with one input and four hidden units, so that the effect of each optimizer on the weights, the gradient and the fitted curve can be seen. The same update rules, with the same formulas, are what runs inside the deep learning libraries.

6 EM Algorithms

6.1 Overview of EM

6.1.1 Setup and motivation

Let Y denote the observed data and θ the parameter, with likelihood $L(\theta) = P(Y \mid \theta)$. In many models this quantity is itself an integral (or sum) over latent or missing data Z:

$$P(Y \mid \theta) = \int P(Y, Z \mid \theta)\, dZ. \tag{6.1}$$

The integral is what makes direct maximization awkward: it destroys the product structure that would otherwise make the log-likelihood a sum of simple terms. The EM algorithm avoids it by working with the complete-data likelihood, which retains that structure, and correcting for the fact that Z is unobserved.

Table 6.1: Fixes the notation for the observed-data and complete-data likelihoods used throughout the chapter.
Observed and complete likelihoods

	Likelihood	Log-likelihood
Observed	$L_{\text{obs}}(\theta) = P(Y \mid \theta)$	$\ell_{\text{obs}}(\theta) = \log P(Y \mid \theta)$
Complete	$L_{\text{comp}}(\theta) = P(Y, Z \mid \theta)$	$\ell_{\text{comp}}(\theta; Y, Z) = \log P(Y, Z \mid \theta)$

6.1.2 The algorithm

Start from $\widehat{\theta}^{(0)}$. For $t = 0, 1, 2, \ldots$:

E-step. Replace $\ell_{\text{obs}}(\theta)$ by the expected complete log-likelihood,

$$Q\big(\theta \mid \widehat{\theta}^{(t)}\big) = E_{Z|Y,\widehat{\theta}^{(t)}}\left[\ell_{\text{comp}}(\theta; Y, Z)\right]. \tag{6.2}$$

M-step. Maximize it,

$$\widehat{\theta}^{(t+1)} = \arg\max_{\theta}\ Q\big(\theta \mid \widehat{\theta}^{(t)}\big). \tag{6.3}$$

Iterate until $\widehat{\theta}^{(t)}$ converges.

> ⚠ A common misreading
>
> Q is an *average of* $\ell_{\text{comp}}(\theta; Y, Z)$ over $P(Z \mid Y, \hat{\theta}^{(t)})$. It is **not** $\ell_{\text{comp}}\big(\theta; Y, E(Z \mid Y, \hat{\theta}^{(t)})\big)$ — that is, EM is not "fill in the missing values with their expectations and maximize". The two coincide only when ℓ_{comp} happens to be linear in Z, which is true in both examples of this chapter and false in general.

6.1.3 Q as an average of complete log-likelihoods

Fix $\hat{\theta}^{(t)}$. Each draw $Z^{(s)} \sim P(Z \mid Y, \hat{\theta}^{(t)})$ gives one curve $\ell_{\text{comp}}(\theta; Y, Z^{(s)})$, and $Q(\theta \mid \hat{\theta}^{(t)})$ is their expectation. The figure uses the censored Poisson model of Section 6.3 with $\lambda^{(t)} = 1.5$.

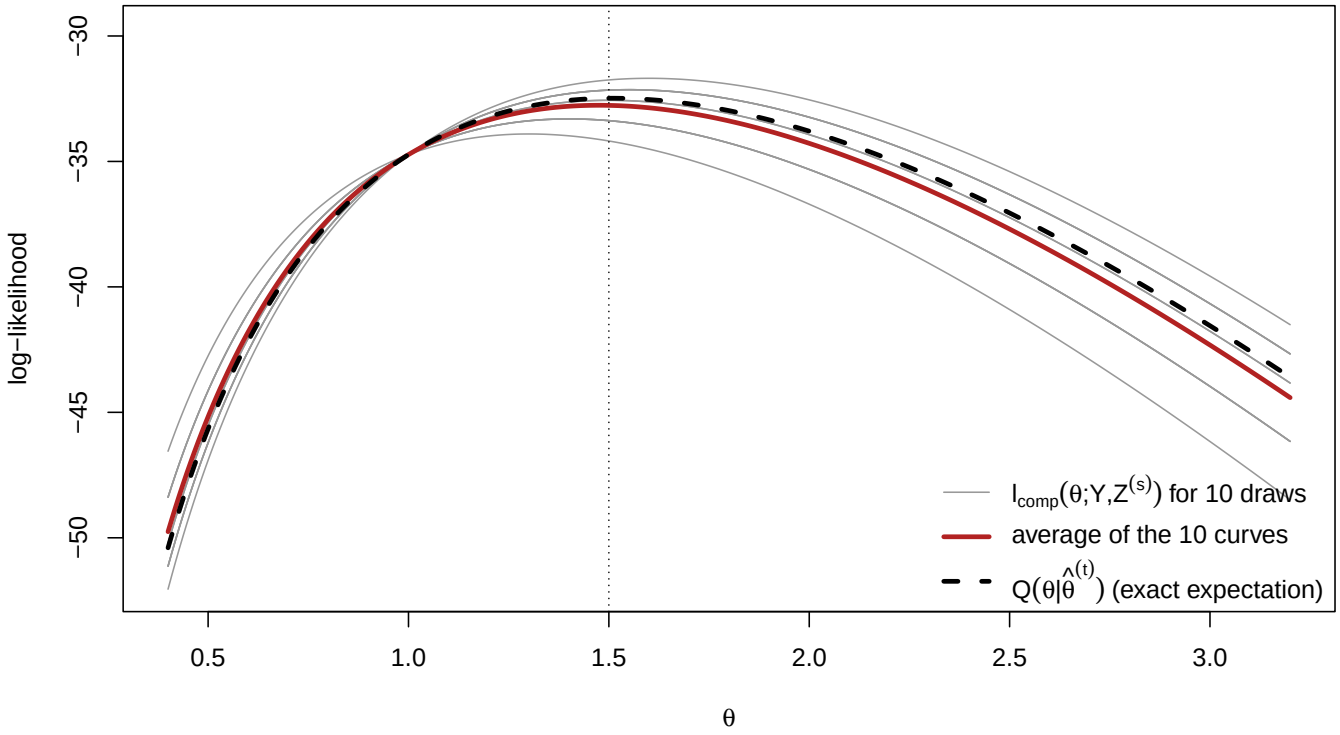


Figure 6.1: Illustrates the Q function of the EM algorithm as the expectation of the complete-data log-likelihood over the unobserved data, approximated here by averaging simulated completions. Ten complete-data log-likelihood curves, their average, and the exact Q function.

Why each iteration cannot decrease ℓ_{obs}, and the view of EM as alternating maximization of a single function, are developed in Section 6.7 at the end of the chapter.

6.2 Example 1: Two-Component Normal Mixture

6.2.1 Model and likelihoods

$$Z_i \sim \text{Bern}(p), \qquad Y_i \mid Z_i = 1 \sim N(\mu_1, 1), \qquad Y_i \mid Z_i = 0 \sim N(\mu_0, 1). \tag{6.4}$$

Observed: $y_1, \dots, y_n$. Hidden: $z_1, \dots, z_n$. Parameter $\theta = (\mu_0, \mu_1, p)$.

Complete likelihood

$$P(y, z \mid \theta) = \prod_{i=1}^{n} P(y_i \mid z_i, \theta)\, P(z_i \mid \theta) = \prod_{i=1}^{n} \varphi(y_i \mid \mu_{z_i}, 1)\, p^{z_i} (1-p)^{1-z_i}. \tag{6.5}$$

Observed likelihood, summing out each z_i,

$$P(y \mid \theta) = \prod_{i=1}^{n} \left[\varphi(y_i \mid \mu_1, 1)\, p + \varphi(y_i \mid \mu_0, 1)(1-p) \right]. \tag{6.6}$$

The sum inside the product is what makes direct maximization awkward.

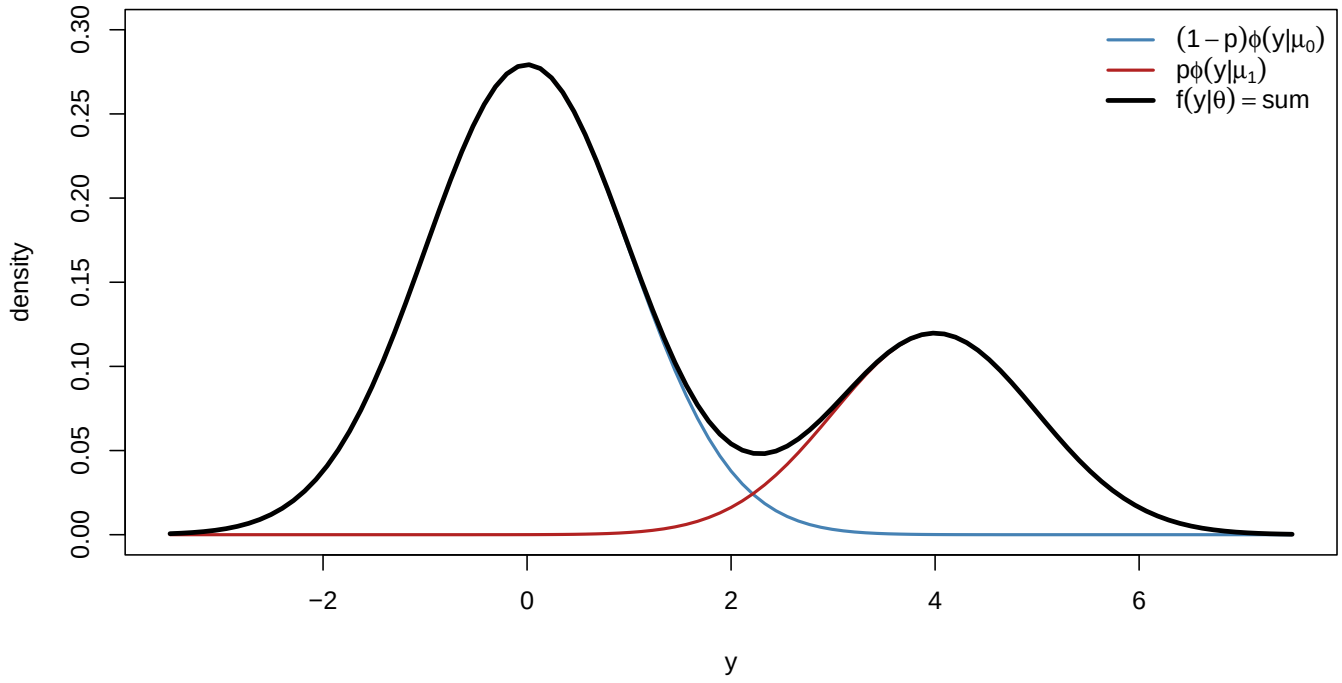


Figure 6.2: Introduces the two-component normal mixture model by showing that its density is the weighted sum of the two component densities. The observed density is a sum of two weighted normal components.

The marginal density may be bimodal or unimodal depending on p and $|\mu_1 - \mu_0|$ — a point worth keeping in mind, since the unimodal case is where EM struggles most.

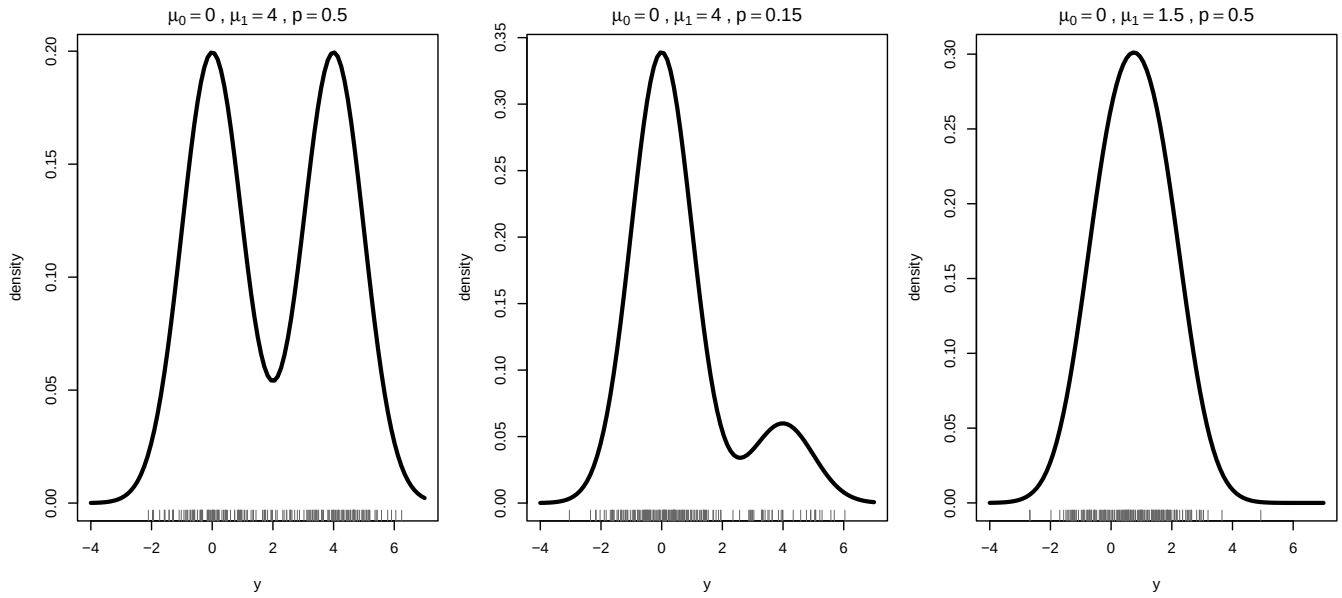


Figure 6.3: Shows that a mixture density can look unimodal or bimodal depending on its parameters, which affects how easily those parameters can be estimated. Separation and mixing proportion determine whether the mixture is visibly bimodal.

6.2.2 Complete log-likelihood

With $\log \varphi(y \mid \mu, 1) = -\frac{1}{2}(y-\mu)^2 - \log\sqrt{2\pi}$,

$$\begin{aligned}\ell_{\text{comp}}(\theta; y, z) &= \sum_{i=1}^{n} \Big[z_i \log p + (1 - z_i) \log(1-p) - \tfrac{1}{2}(y_i - \mu_{z_i})^2 \Big] + C \\ &= \left(\sum_{i=1}^{n} z_i \right) \log \frac{p}{1-p} + n \log(1-p) \\ &\quad - \frac{1}{2} \sum_{i=1}^{n} \left[z_i (y_i - \mu_1)^2 + (1 - z_i)(y_i - \mu_0)^2 \right] + C.\end{aligned} \tag{6.7}$$

Two features drive everything that follows. The expression is **linear in each** z_i, so the E-step needs only $E(Z_i \mid y_i, \widehat{\theta}^{(t)})$ rather than the full conditional distribution. And given z, the parameters **separate**: p is a Bernoulli proportion while μ_1 and μ_0 are group means.

6.2.3 E-step: soft classification

Given $\widehat{\theta}^{(t)} = (\widehat{\mu}_0^{(t)}, \widehat{\mu}_1^{(t)}, \widehat{p}^{(t)})$, Bayes' rule gives

$$\widehat{p}_i^{(t)} \equiv P\big(Z_i = 1 \mid y_i, \widehat{\theta}^{(t)}\big) = \frac{\varphi\big(y_i \mid \widehat{\mu}_1^{(t)}, 1\big)\, \widehat{p}^{(t)}}{\varphi\big(y_i \mid \widehat{\mu}_0^{(t)}, 1\big)(1 - \widehat{p}^{(t)}) + \varphi\big(y_i \mid \widehat{\mu}_1^{(t)}, 1\big)\, \widehat{p}^{(t)}}, \tag{6.8}$$

and therefore

$$Q\big(\theta \mid \hat{\theta}^{(t)}\big) = \left(\sum_i \hat{p}_i^{(t)}\right) \log\frac{p}{1-p} + n\log(1-p) \\ -\frac{1}{2}\sum_i \Big[\hat{p}_i^{(t)}(y_i-\mu_1)^2 + \big(1-\hat{p}_i^{(t)}\big)(y_i-\mu_0)^2\Big]\,. \tag{6.9}$$

Computationally, $\hat{p}_i^{(t)}$ must be formed on the log scale. Writing $a_i = \log\varphi(y_i \mid \hat{\mu}_1^{(t)}, 1) + \log\hat{p}^{(t)}$ and $b_i = \log\varphi(y_i \mid \hat{\mu}_0^{(t)}, 1) + \log(1-\hat{p}^{(t)})$, we have $\hat{p}_i^{(t)} = \exp\{a_i - \mathrm{lse}(a_i, b_i)\}$ where lse is the log-sum-exp function of the chapter on computer arithmetic. Forming the ratio directly underflows as soon as an observation sits far from both means.

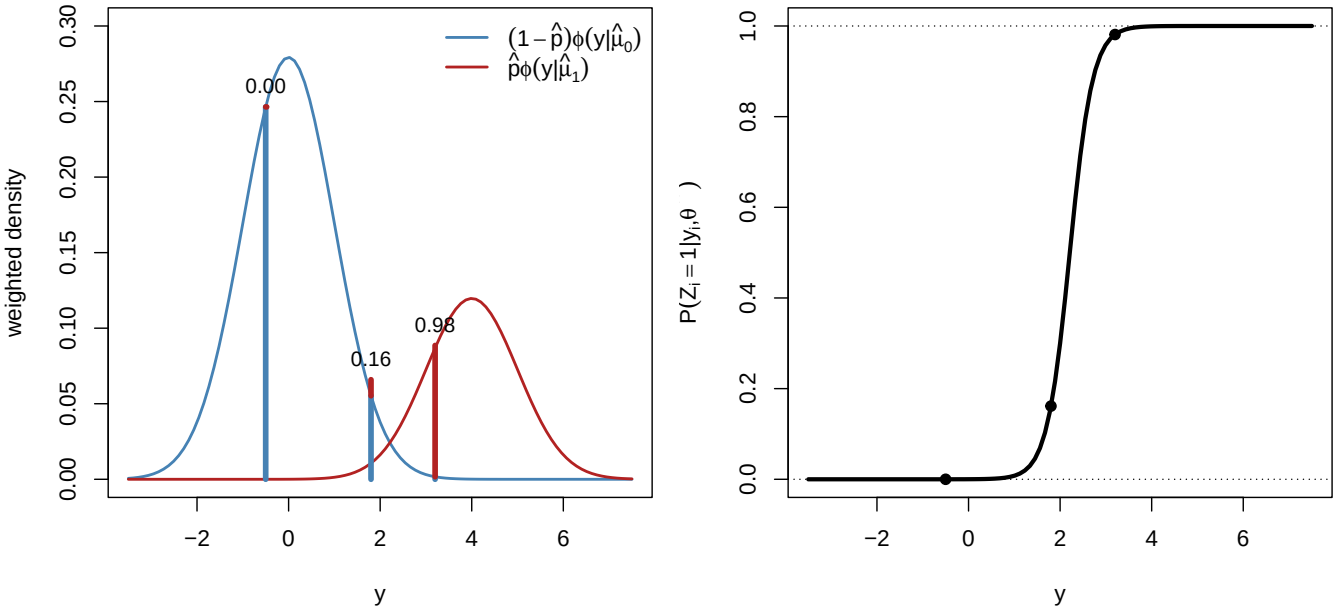


Figure 6.4: Illustrates the E-step of EM for a mixture: the responsibility of a component is the posterior probability that an observation came from it. Left: the responsibility at each y_i is the red share of the total height. Right: responsibility as a function of y.

6.2.4 M-step: weighted proportions and means

Q separates into one term in p and two in μ_1, μ_0, each maximized in closed form.

Table 6.2: Derives the M-step updates for the mixture model, showing that each is a weighted version of the complete-data MLE. M-step updates for the two-component mixture

Update	Stationary equation $\Longrightarrow$ solution
p	$\dfrac{\partial Q}{\partial p} = \dfrac{\sum_i \hat{p}_i^{(t)}}{p(1-p)} - \dfrac{n}{1-p} = 0 \implies \hat{p}^{(t+1)} = \dfrac{1}{n}\sum_{i=1}^{n} \hat{p}_i^{(t)}$
μ_1	$\dfrac{\partial Q}{\partial \mu_1} = \sum_i \hat{p}_i^{(t)}(y_i - \mu_1) = 0 \implies \hat{\mu}_1^{(t+1)} = \dfrac{\sum_i \hat{p}_i^{(t)} y_i}{\sum_i \hat{p}_i^{(t)}}$
μ_0	weights $1 - \hat{p}_i^{(t)}$: $\hat{\mu}_0^{(t+1)} = \dfrac{\sum_i \left(1 - \hat{p}_i^{(t)}\right) y_i}{\sum_i \left(1 - \hat{p}_i^{(t)}\right)}$

In words: the M-step is the complete-data MLE with each observation split between the two groups in proportion to its soft classification. Return to the E-step with $\hat{\theta}^{(t+1)}$ and recompute the responsibilities.

6.2.5 Implementation

```
log_sum_exp <- function(log_x) {
    max_log_x <- max(log_x)
    max_log_x + log(sum(exp(log_x - max_log_x)))
}

### observed log-likelihood, theta = (p, mu1, mu0)
log_like_obs <- function(theta, Y) {
    log_joint <- cbind(log(theta[1])     + dnorm(Y, theta[2], 1, log = TRUE),
                       log(1 - theta[1]) + dnorm(Y, theta[3], 1, log = TRUE))
    sum(apply(log_joint, 1, log_sum_exp))
}

### E-step: responsibilities P(Z_i = 1 | y_i, theta)
e_step <- function(theta, Y) {
    l1 <- dnorm(Y, theta[2], 1, log = TRUE) + log(theta[1])
    l0 <- dnorm(Y, theta[3], 1, log = TRUE) + log(1 - theta[1])
    exp(l1 - apply(cbind(l1, l0), 1, log_sum_exp))
}

### M-step: closed-form weighted updates
m_step <- function(w, Y) {
    c(mean(w),
      sum(Y * w) / sum(w),
      sum(Y * (1 - w)) / sum(1 - w))
```

```
}

em_mixnorm <- function(theta0, Y, no_iters) {
    out <- matrix(NA_real_, no_iters + 1, 4,
                  dimnames = list(NULL, c("p", "mu1", "mu0", "log_lik")))
    theta <- theta0
    out[1, ] <- c(theta, log_like_obs(theta, Y))
    for (i in seq_len(no_iters)) {
        theta <- m_step(e_step(theta, Y), Y)
        out[i + 1, ] <- c(theta, log_like_obs(theta, Y))
    }
    out
}
```

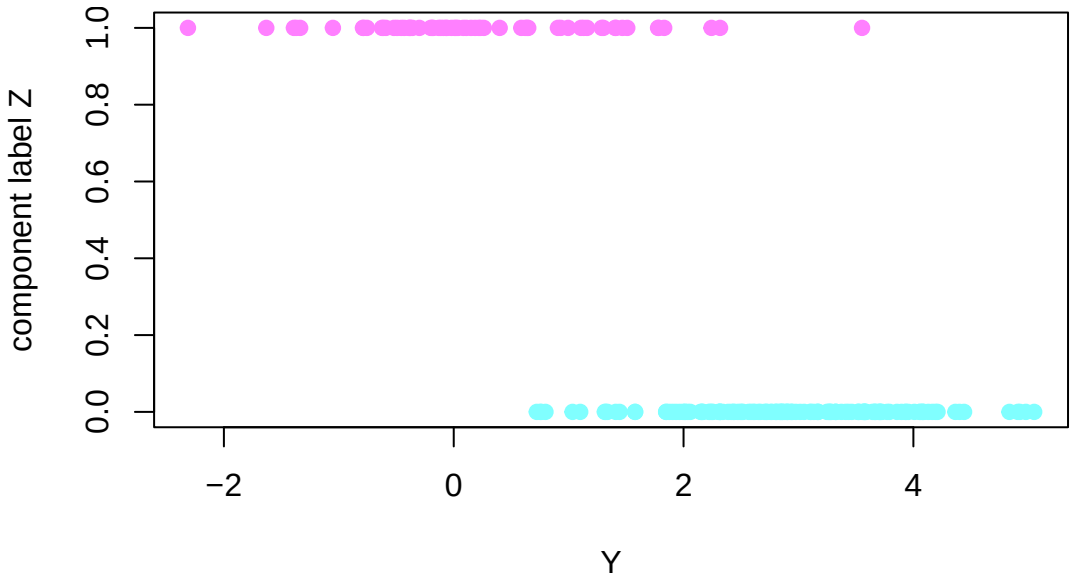


Figure 6.5: Shows the simulated data used to demonstrate EM for a mixture; the colours reveal the component labels, which are unobserved in practice and must be inferred. Simulated mixture data, coloured by the (normally unobserved) component label.

A well-chosen start converges quickly:

```
round(em_mixnorm(c(0.5, 0, 3), data, 10), 4)
```

```
          p    mu1    mu0   log_lik
 [1,] 0.5000 0.0000 3.0000 -375.3490
 [2,] 0.3377 0.2695 3.0137 -361.6709
 [3,] 0.3242 0.2319 2.9768 -361.3077
 [4,] 0.3154 0.1956 2.9584 -361.1388
 [5,] 0.3094 0.1694 2.9463 -361.0581
 [6,] 0.3054 0.1513 2.9380 -361.0202
 [7,] 0.3026 0.1389 2.9323 -361.0026
 [8,] 0.3007 0.1306 2.9285 -360.9945
 [9,] 0.2995 0.1249 2.9258 -360.9908
```

```
[10,] 0.2986 0.1211 2.9240 -360.9891
[11,] 0.2980 0.1185 2.9228 -360.9884
```

The monotone increase of the last column is the ascending property in action; it is the single most useful diagnostic when debugging an EM implementation, since any decrease indicates an error in the E- or M-step.

6.2.6 Shinylive App for the EM Algorithm for a Normal Mixture

There is a shinylive app to let you watch the EM algorithm fit a two-component normal mixture, showing how the estimates and the likelihood evolve from iteration to iteration.

6.2.7 Comparison with general-purpose optimizers

EM is not the only way to maximize the observed likelihood. Applying a general-purpose optimizer directly requires attention to the parameter space: p must lie in $(0, 1)$, so we reparameterize with the logit.

```r
neg_loglike_obs <- function(theta, Y) -log_like_obs(theta, Y)

### transformed parameterization: ttheta[1] = logit(p)
neg_loglike_obs_transf <- function(ttheta, Y) {
    theta <- ttheta
    theta[1] <- 1 / (1 + exp(-ttheta[1]))
    neg_loglike_obs(theta, Y)
}

lp0 <- log(0.3 / 0.7)

### Newton-type, untransformed: p can leave (0, 1)
try(nlm(neg_loglike_obs, p = c(0.3, 0, 3), Y = data)$estimate)
```

```
[1] 0.2968473 0.1131871 2.9203484
```

```r
try(nlm(neg_loglike_obs, p = c(0.5, -10, 5), Y = data)$estimate)
```

```
[1]  5.869764e-06 -1.000004e+01  4.038014e+00
```

```r
### Newton-type, transformed
try(nlm(neg_loglike_obs_transf, p = c(lp0, 0, 3), Y = data)$estimate)
```

```
[1] -0.8623555  0.1131868  2.9203483
```

```r
try(nlm(neg_loglike_obs_transf, p = c(0, -10, 5), Y = data)$estimate)
```

```
[1] -16.604291  -9.996458   2.087049
```

```
### Nelder-Mead
try(optim(c(lp0, 0, 3), neg_loglike_obs_transf, Y = data, method = "Nelder-Mead")$par)
```

```
[1] -0.8623197  0.1132163  2.9203325
```

```
try(optim(c(0, -10, 5), neg_loglike_obs_transf, Y = data, method = "Nelder-Mead")$par)
```

```
[1] -0.8622804  0.1131101  2.9206253
```

```
### Conjugate gradient
try(optim(c(0, 0, 3), neg_loglike_obs_transf, Y = data, method = "CG")$par)
```

```
[1] -0.8623543  0.1131880  2.9203502
```

```
try(optim(c(0, -10, 5), neg_loglike_obs_transf, Y = data, method = "CG")$par)
```

```
[1] -6.090903 -9.998146  2.080878
```

Applied to the observed log-likelihood, these algorithms are more fragile than EM when the initial values are poorly chosen: the untransformed runs can step outside $p \in (0, 1)$ and produce `NaN`, and the gradient-based methods can stall in the flat region that a distant start creates. Nelder–Mead is the most stable of the three here, at the cost of many more function evaluations.

Table 6.3: Helps decide between EM and a general-purpose optimizer by listing the strengths and weaknesses of each. EM compared with direct numerical maximization

	EM	General optimizer on ℓ_{obs}
Constraints	Respected automatically by the M-step	Require reparameterization
Monotonicity	Guaranteed; a decrease signals a coding error	None; a bad step can be accepted
Derivatives	None needed	Gradient helpful, Hessian better
Convergence rate	Linear, rate set by the missing information	Superlinear or quadratic near the optimum
Standard errors	Not produced; need Louis' method or a bootstrap	Hessian available directly
Behaviour from poor starts	Slow but stable	Can fail outright

The last row is the practical trade. EM rarely fails and rarely needs tuning, which is why it dominates in mixture and latent-variable modelling; it is also slow near the optimum and gives no standard errors, which is why a hybrid — EM to get close, then a Newton step to polish and to obtain the Hessian — is common in production code.

6.3 Example 2: Censored Poisson Data

6.3.1 Truncated distributions in brief

Let $X \sim f(x)$ and $Y = I(X > c)$. Then $Y \sim \text{Bern}\{P(X > c)\}$, and the conditional distribution of X given Y is f restricted to one side of c and renormalized:

$$f(x \mid Y = 1) = \frac{f(x)\, I(x > c)}{\int_c^{\infty} f(t)\, dt}, \qquad f(x \mid Y = 0) = \frac{f(x)\, I(x \le c)}{\int_{-\infty}^{c} f(t)\, dt}. \tag{6.10}$$

For discrete X the integrals are sums. This is exactly the distribution the E-step needs when an observation is known only to lie on one side of c.

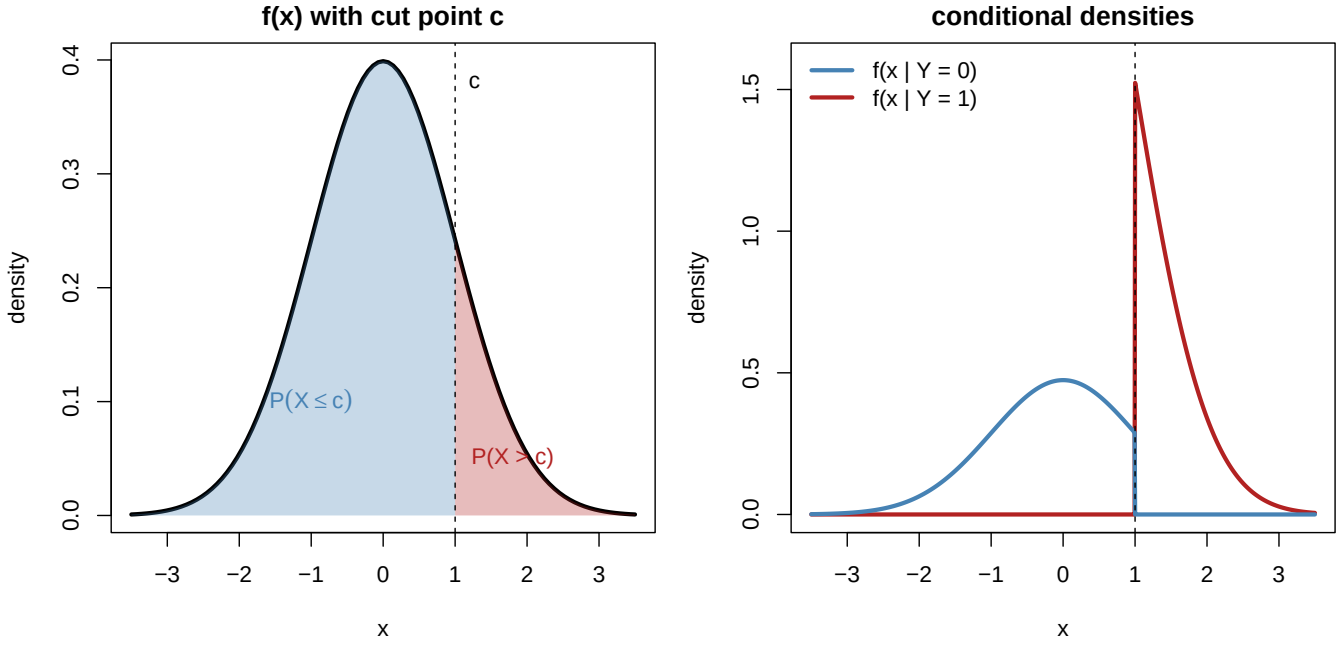


Figure 6.6: Illustrates the conditional distributions needed in the E-step when an observation is only known to lie on one side of a cut point. A density split at a cut point, and the two renormalized conditional densities.

6.3.2 Data and likelihoods

Let $y_i \overset{iid}{\sim} \text{Pois}(\lambda)$ for $i = 1, \dots, n$. The first m values are observed exactly; for $i > m$ we know only that $y_i < 2$, recorded as $x_i = I(y_i < 2) = 1$.

Observed data: $y_1, \dots, y_m$ together with $x_{m+1} = \cdots = x_n = 1$. **Missing**: $y_{m+1}, \dots, y_n$.

Complete likelihood

$$\prod_{i=1}^{n} f^{\text{Pois}}(y_i \mid \lambda) \cdot \prod_{i=m+1}^{n} I(y_i < 2). \tag{6.11}$$

Observed likelihood

$$\prod_{i=1}^{m} f^{\text{Pois}}(y_i \mid \lambda) \cdot \prod_{i=m+1}^{n} P(y_i < 2 \mid \lambda), \qquad P(y_i < 2 \mid \lambda) = e^{-\lambda}(1+\lambda). \tag{6.12}$$

Taking logarithms and dropping constants,

$$\ell_{\text{obs}}(\lambda) = m\bar{y}_{1:m} \log \lambda - n\lambda + (n-m)\log(1+\lambda). \tag{6.13}$$

The second factor makes the MLE nonlinear in λ; EM sidesteps it.

6.3.3 E-step

$$\ell_{\text{comp}}(\lambda) = -n\lambda + (\log \lambda) \sum_{i=1}^{m} y_i + (\log \lambda) \underbrace{\sum_{i=m+1}^{n} y_i}_{\text{missing}} - \sum_{i=1}^{n} \log(y_i!). \tag{6.14}$$

For $i > m$ we have $y_i \in \{0, 1\}$, and the Poisson pmf truncated to that set given $\lambda^{(t)}$ is

$$P(y_i = 0 \mid x_i = 1) = \frac{1}{1+\lambda^{(t)}}, \qquad P(y_i = 1 \mid x_i = 1) = \frac{\lambda^{(t)}}{1+\lambda^{(t)}}, \tag{6.15}$$

so that

$$\hat{y}^{(t)} \equiv E\big(y_i \mid x_i = 1, \lambda^{(t)}\big) = \frac{\lambda^{(t)}}{1+\lambda^{(t)}}. \tag{6.16}$$

Since $y_i! = 1$ for $y_i \in \{0, 1\}$, the $\log(y_i!)$ term is a known constant and no further expectation is needed.

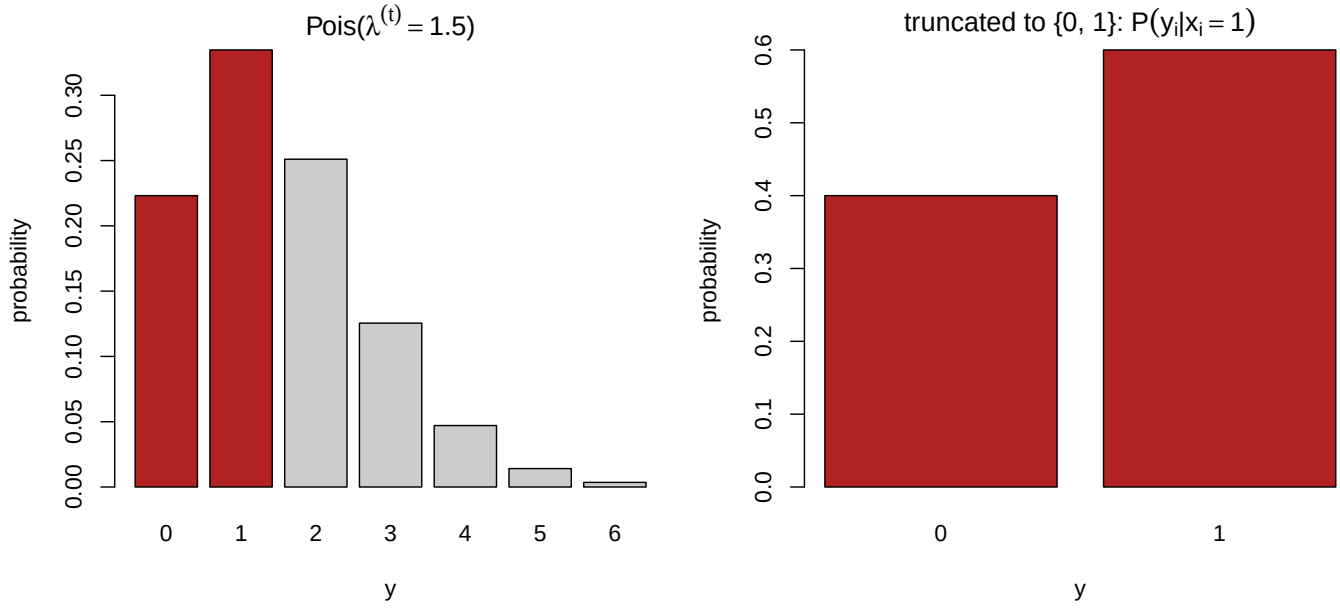


Figure 6.7: Illustrates the E-step for censored Poisson counts: the missing count follows the Poisson distribution restricted to the values consistent with the censoring information. The conditional distribution of a censored count is the Poisson pmf restricted to $\{0, 1\}$ and renormalized.

6.3.4 M-step and the EM update

$$Q\big(\lambda \mid \lambda^{(t)}\big) = -n\lambda + (\log\lambda)\left[\sum_{i=1}^{m} y_i + (n-m)\,\hat{y}^{(t)}\right] + C^{(t)}, \tag{6.17}$$

$$\frac{\partial Q}{\partial \lambda} = -n + \frac{1}{\lambda}\left[\sum_{i=1}^{m} y_i + (n-m)\,\hat{y}^{(t)}\right] = 0, \tag{6.18}$$

$$\Longrightarrow \quad \lambda^{(t+1)} = \frac{m\,\bar{y}_{1:m} + (n-m)\dfrac{\lambda^{(t)}}{1+\lambda^{(t)}}}{n}. \tag{6.19}$$

The update is the ordinary Poisson MLE — the sample mean — with each censored value replaced by its conditional expectation. As warned in the overview, this works *only* because ℓ_{comp} is linear in the missing y_i.

6.3.5 Implementation

```
### observed log-likelihood, up to an additive constant
##   m    : number of exactly observed counts
##   ybar : their mean
##   ncen : number of counts known only to be < 2
```

```
loglik_obs_pois <- function(m, ybar, ncen, lambda) {
    m * ybar * log(lambda) - (m + ncen) * lambda + ncen * log(1 + lambda)
}

em_censored_poisson <- function(m, ybar, ncen,
                                lambda0 = (m * ybar + ncen) / (m + ncen),
                                iterations = 20) {
    out <- matrix(NA_real_, iterations + 1, 2,
                  dimnames = list(NULL, c("lambda", "log_lik")))
    lambda <- lambda0
    out[1, ] <- c(lambda, loglik_obs_pois(m, ybar, ncen, lambda))
    for (i in seq_len(iterations)) {
        y_mis  <- lambda / (1 + lambda)                       # E-step
        lambda <- (m * ybar + ncen * y_mis) / (m + ncen)       # M-step
        out[i + 1, ] <- c(lambda, loglik_obs_pois(m, ybar, ncen, lambda))
    }
    out
}
```

```
set.seed(812)
y    <- rpois(200, 3)
ncen <- sum(y < 2)
m    <- length(y) - ncen
ybar <- mean(y[y >= 2])
c(m = m, ncen = ncen, ybar_observed = ybar)
```

```
           m         ncen ybar_observed
   157.000000   43.000000     3.528662
```

Discarding the censored observations and averaging the rest gives an estimate clearly above the true mean of 3: conditioning on $y \geq 2$ removes the small counts and biases the mean upward. EM recovers them.

```
round(em_censored_poisson(m, ybar, ncen, lambda0 = 100, iterations = 10), 5)
```

```
        lambda      log_lik
 [1,] 100.00000 -17250.28553
 [2,]   2.98287     68.31091
 [3,]   2.93102     68.40282
 [4,]   2.93031     68.40284
 [5,]   2.93030     68.40284
 [6,]   2.93030     68.40284
 [7,]   2.93030     68.40284
 [8,]   2.93030     68.40284
 [9,]   2.93030     68.40284
[10,]   2.93030     68.40284
[11,]   2.93030     68.40284
```

```
round(em_censored_poisson(m, ybar, ncen, lambda0 = ybar, iterations = 10), 5)
```

```
      lambda  log_lik
 [1,] 3.52866 57.76492
 [2,] 2.93752 68.40109
 [3,] 2.93040 68.40284
```

```
 [4,] 2.93030 68.40284
 [5,] 2.93030 68.40284
 [6,] 2.93030 68.40284
 [7,] 2.93030 68.40284
 [8,] 2.93030 68.40284
 [9,] 2.93030 68.40284
[10,] 2.93030 68.40284
[11,] 2.93030 68.40284
```

Both starting values reach the same limit, and the log-likelihood increases monotonically from each. Note how slowly the run from $\lambda^{(0)} = 100$ descends at first — a single-parameter illustration of the same slow-drift behaviour seen in the mixture app.

6.3.6 Comparison with general-purpose optimizers

```
neg_loglik_pois <- function(lambda, m, ybar, ncen) {
    -loglik_obs_pois(m, ybar, ncen, lambda)
}

try(nlm(neg_loglik_pois, p = ybar, m = m, ybar = ybar, ncen = ncen)$estimate)
```

```
[1] 2.930295
```

```
try(nlm(neg_loglik_pois, p = 100,  m = m, ybar = ybar, ncen = ncen)$estimate)
```

```
[1] 2.930295
```

```
try(optim(ybar, neg_loglik_pois, method = "CG",
          m = m, ybar = ybar, ncen = ncen)$par)
```

```
[1] 2.930297
```

```
try(optim(100,  neg_loglik_pois, method = "CG",
          m = m, ybar = ybar, ncen = ncen)$par)
```

```
[1] 2.930297
```

With one parameter and a smooth concave objective this is an easy problem for any method; the contrast with the mixture example is instructive, since there the multimodality and the constraint on p were what made the general optimizers fragile.

6.4 Example 3: Right-Censored Weibull Data

Survival and reliability data are the standard home of censoring. A unit is followed until it fails or until the study ends, whichever comes first, and a unit still running at the end tells us only that its lifetime exceeds the follow-up time. The Poisson example of Section 6.3 censored a count; here the missing quantity is a continuous lifetime, and, unlike the Poisson case, the complete-data log-likelihood is *not* linear in it.

6.4.1 Data and likelihoods

Let $T_1, \dots, T_n$ be independent lifetimes with the Weibull density and survival function

$$f(t \mid k, \lambda) = \frac{k}{\lambda}\Big(\frac{t}{\lambda}\Big)^{k-1} e^{-(t/\lambda)^k}, \qquad S(t \mid k, \lambda) = P(T > t) = e^{-(t/\lambda)^k}, \qquad t > 0, \tag{6.20}$$

with shape $k > 0$ and scale $\lambda > 0$ ($k = 1$ is the exponential distribution); Figure 6.8 shows how they change the density. Unit i is followed up to a censoring time c_i, independent of T_i, and we record

$$y_i = \min(T_i, c_i), \qquad \delta_i = I(T_i \le c_i). \tag{6.21}$$

Let $d = \sum_i \delta_i$ be the number of observed failures. For a censored unit ($\delta_i = 0$) we know only that $T_i > c_i = y_i$.

Observed data: (y_i, δ_i), $i = 1, \dots, n$. **Missing**: T_i for the units with $\delta_i = 0$.

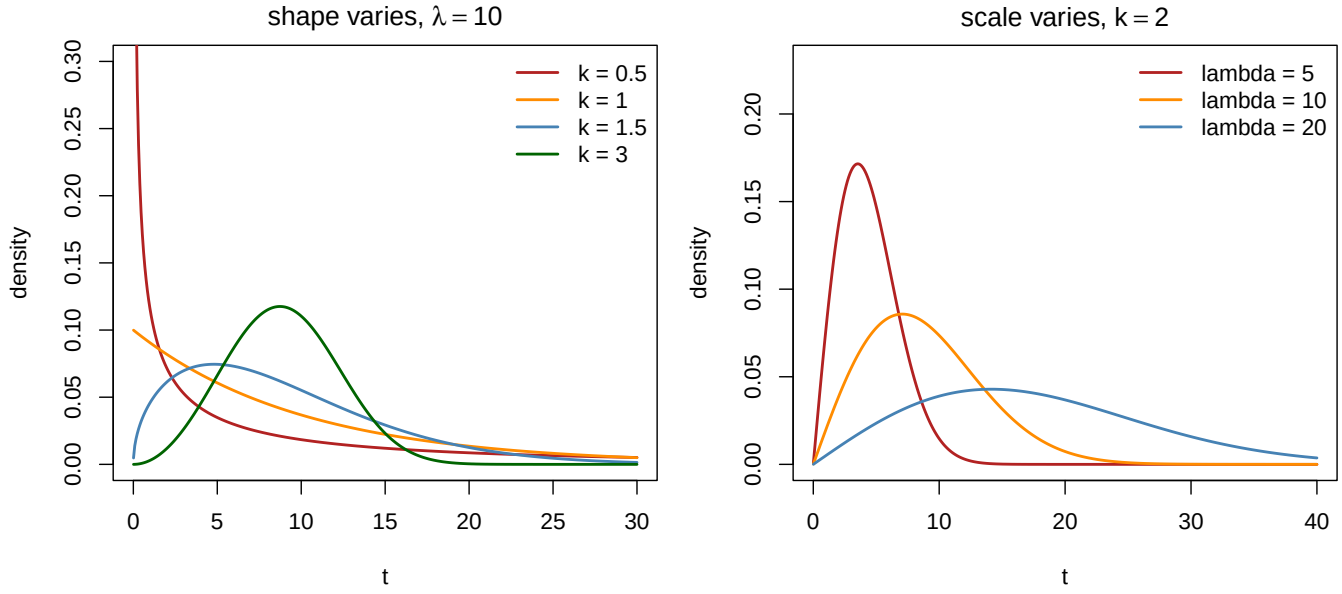


Figure 6.8: Illustrates how the shape k and the scale λ change the Weibull density. Left: different shapes with $\lambda = 10$ ($k < 1$ is decreasing with a spike at 0, $k = 1$ is exponential, $k > 1$ has a mode). Right: different scales with $k = 2$, which stretch the density along the time axis.

Observed likelihood. A failure contributes its density, a censored unit its survival probability:

$$L_{\text{obs}}(k, \lambda) = \prod_{i:\,\delta_i = 1} f(y_i \mid k, \lambda) \prod_{i:\,\delta_i = 0} S(y_i \mid k, \lambda), \tag{6.22}$$

$$\ell_{\text{obs}}(k, \lambda) = d \log k - d\,k \log \lambda + (k - 1) \sum_{i:\,\delta_i = 1} \log y_i - \sum_{i=1}^{n} \Big(\frac{y_i}{\lambda}\Big)^k. \tag{6.23}$$

If k is known, setting $\partial \ell_{\text{obs}}/\partial \lambda = 0$ gives $\hat{\lambda}^k = \sum_{i=1}^n y_i^k / d$, but for unknown k there is no closed form, and the maximization has to be done numerically. EM replaces it by a sequence of simpler problems.

Complete-data log-likelihood. If all T_i were seen,

$$\ell_{\text{comp}}(k, \lambda) = n \log k - nk \log \lambda + (k-1) \sum_{i=1}^n \log T_i - \lambda^{-k} \sum_{i=1}^n T_i^k. \tag{6.24}$$

It depends on the missing T_i through $\log T_i$ and T_i^k, two *nonlinear* functions. As warned in the overview, the E-step therefore needs $E(\log T_i \mid \cdot)$ and $E(T_i^k \mid \cdot)$, not $\big(\log E(T_i \mid \cdot)\big)$ or $E(T_i \mid \cdot)^k$, and substituting $E(T_i \mid \cdot)$ for T_i would give the wrong algorithm.

6.4.2 E-step

Write $\theta^{(t)} = (k_t, \lambda_t)$ for the current value. The key fact is that $U = (T/\lambda_t)^{k_t}$ has the standard exponential distribution, so for a censored unit

$$T_i > c_i \iff U_i > a_i, \qquad a_i = (c_i/\lambda_t)^{k_t}, \tag{6.25}$$

and by the memoryless property of the exponential distribution $U_i \mid U_i > a_i \overset{d}{=} a_i + E$ with $E \sim \text{Exp}(1)$. Hence, given the data and $\theta^{(t)}$,

$$T_i = \lambda_t \, (a_i + E)^{1/k_t}, \qquad E \sim \text{Exp}(1), \tag{6.26}$$

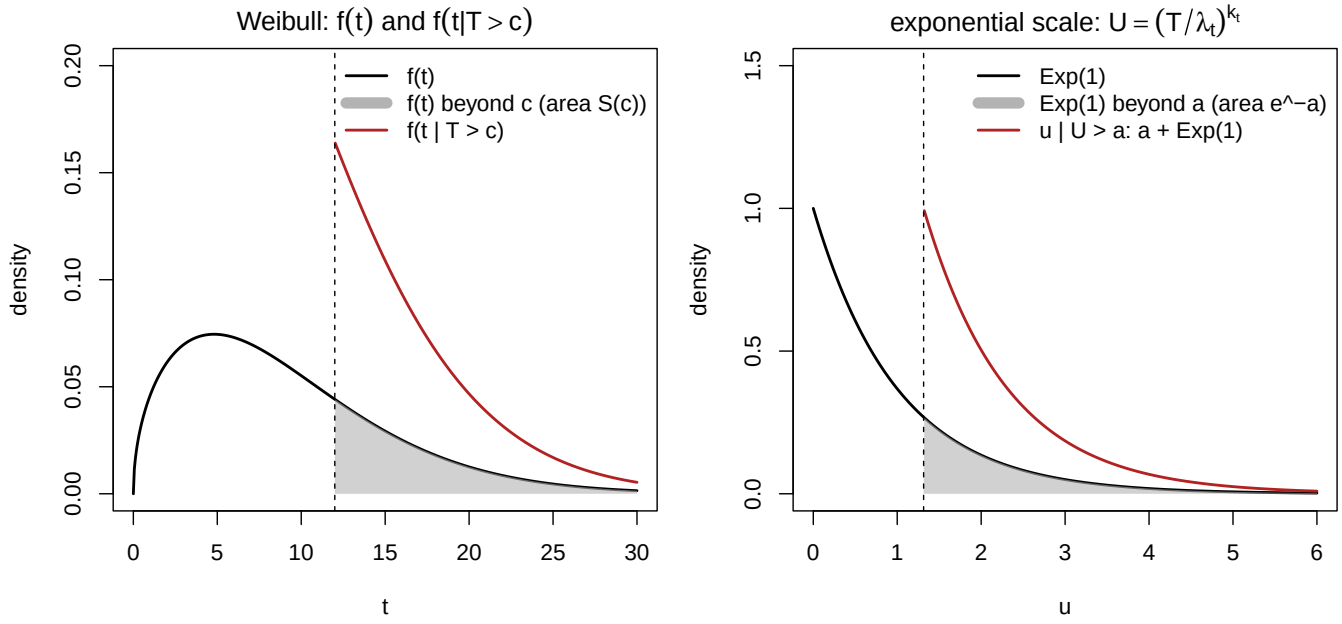


Figure 6.9: Illustrates the conditional distribution of a censored lifetime in the E-step. Left: the Weibull density $f(t \mid k_t, \lambda_t)$ and, in red, the density of T given $T > c$, which is the part of f beyond c divided by $S(c)$. Right: the same on the exponential scale $U = (T/\lambda_t)^{k_t}$, where the conditional density $e^{-(u-a)}$ is the standard exponential shifted to start at $a = (c/\lambda_t)^{k_t}$ (the memoryless property).

Figure 6.9 illustrates the two conditional distributions. which gives both expectations that are needed. For any power $s > 0$,

$$E(T_i^s \mid T_i > c_i, \theta^{(t)}) = \lambda_t^s \, E\big[(a_i + E)^{s/k_t}\big] = \lambda_t^s \, e^{a_i} \, \Gamma(1 + s/k_t,\ a_i), \tag{6.27}$$

where $\Gamma(b, a) = \int_a^\infty u^{b-1} e^{-u} \, du$ is the upper incomplete gamma function. For the power $s = k_t$ this reduces to the elementary expression $\lambda_t^{k_t}(a_i + 1) = c_i^{k_t} + \lambda_t^{k_t}$: a censored unit is expected to run on for one more unit of the exponential clock. For the logarithm, integration by parts gives

$$E(\log T_i \mid T_i > c_i, \theta^{(t)}) = \log \lambda_t + \frac{1}{k_t}\Big[\log a_i + e^{a_i} E_1(a_i)\Big], \qquad E_1(a) = \int_a^\infty \frac{e^{-u}}{u} \, du, \tag{6.28}$$

the exponential integral. For an observed failure the expectations are simply y_i^s and $\log y_i$. Collect

$$S_{\log}^{(t)} = \sum_{i=1}^n E(\log T_i \mid \cdot), \qquad S_k^{(t)}(k) = \sum_{i=1}^n E(T_i^k \mid \cdot) = \sum_{i:\,\delta_i=1} y_i^k + \sum_{i:\,\delta_i=0} \lambda_t^k \, e^{a_i} \, \Gamma(1 + k/k_t,\ a_i). \tag{6.29}$$

Then

$$Q(k, \lambda \mid \theta^{(t)}) = n \log k - nk \log \lambda + (k-1) \, S_{\log}^{(t)} - \lambda^{-k} S_k^{(t)}(k). \tag{6.30}$$

6.4.3 M-step

For fixed k, $\partial Q/\partial\lambda = -nk/\lambda + k\lambda^{-k-1} S_k^{(t)}(k) = 0$ gives

$$\lambda^k = \frac{1}{n} S_k^{(t)}(k), \tag{6.31}$$

the Weibull MLE of λ^k for complete data, with each T_i^k replaced by its conditional expectation. Substituting this back gives the profile of Q in k alone, up to a constant,

$$Q_p(k) = n \log k - n \log\Big(\tfrac{1}{n} S_k^{(t)}(k)\Big) + (k-1)\, S_{\log}^{(t)}, \tag{6.32}$$

so the M-step maximizes the one-dimensional function Q_p (here by `optimize`) and then sets $\lambda_{t+1} = \big(S_k^{(t)}(k_{t+1})/n\big)^{1/k_{t+1}}$. Because $S_k^{(t)}(k)$ depends on k, no closed form exists for k_{t+1}, but every M-step is a cheap one-dimensional problem, whereas the original problem couples k and λ through the censored terms.

6.4.4 Special case: known shape

If k is known, only $E(T_i^k \mid \cdot) = c_i^k + \lambda_t^k$ is needed, and the update is in closed form,

$$\lambda_{t+1}^k = \frac{1}{n}\Big[\sum_{i:\,\delta_i=1} y_i^k + \sum_{i:\,\delta_i=0} (y_i^k + \lambda_t^k)\Big] = \frac{\sum_{i=1}^n y_i^k + (n-d)\,\lambda_t^k}{n}, \tag{6.33}$$

the same structure as the Poisson update: the complete-data estimate with each censored value replaced by its conditional expectation. Its fixed point satisfies $\lambda^k = (\sum_i y_i^k + (n-d)\lambda^k)/n$, that is $\lambda^k = \sum_i y_i^k/d$, which is the MLE. Subtracting the fixed point from the update gives, exactly,

$$\lambda_{t+1}^k - \hat\lambda^{\,k} = \frac{n-d}{n}\,(\lambda_t^k - \hat\lambda^{\,k}), \tag{6.34}$$

so the error contracts by the *fraction of censored units* at every iteration. This is the general rule that EM converges at a rate equal to the fraction of missing information: with 10% censoring the algorithm is fast, and with 70% censoring it is slow.

6.4.5 Implementation

```r
### observed log-likelihood; y = min(T, c), event = (T <= c)
loglik_weib <- function(k, lam, y, event) {
    sum(event * (log(k) - k * log(lam) + (k - 1) * log(y))) - sum((y / lam)^k)
}

### E(log(a + E)), E ~ Exp(1): the integral behind E(log T | T > c)
elog_term <- function(a) integrate(function(x) log(a + x) * exp(-x), 0, Inf)$value

### E-step: the two sums S_log and S_k(.) of the text
e_step_weib <- function(y, event, k_t, lam_t) {
    cen <- !event
    a   <- (y[cen] / lam_t)^k_t
    elogT <- log(y)
    elogT[cen] <- log(lam_t) + vapply(a, elog_term, numeric(1)) / k_t
    ## S_k(k) = sum_i E(T_i^k | data, theta_t); computed on the log scale so that
    ## a large a = (c / lam_t)^k_t cannot overflow exp(a)
    S_k <- function(k) {
        s <- k / k_t
        sum(y[event]^k) +
            sum(lam_t^k * exp(a + pgamma(a, s + 1, lower.tail = FALSE, log.p = TRUE) +
                              lgamma(s + 1)))
    }
    list(S_log = sum(elogT), S_k = S_k)
}

### EM: the M-step maximizes the profile Q_p(k) in one dimension, then sets lambda
em_weib <- function(y, event, k0 = 1, lam0 = mean(y), iterations = 200, tol = 1e-10) {
    n <- length(y); k <- k0; lam <- lam0
    out <- matrix(NA_real_, iterations + 1, 3,
                  dimnames = list(NULL, c("k", "lambda", "log_lik")))
    out[1, ] <- c(k, lam, loglik_weib(k, lam, y, event))
    for (i in seq_len(iterations)) {
        E  <- e_step_weib(y, event, k, lam)                                  # E-step
        Qp <- function(kk) n * log(kk) - n * log(E$S_k(kk) / n) + (kk - 1) * E$S_log
        k  <- optimize(Qp, c(k / 4, 4 * k), maximum = TRUE, tol = 1e-12)$maximum   # M-step
        lam <- (E$S_k(k) / n)^(1 / k)
        out[i + 1, ] <- c(k, lam, loglik_weib(k, lam, y, event))
        if (abs(out[i + 1, 3] - out[i, 3]) < tol) {
            out <- out[seq_len(i + 1), , drop = FALSE]; break
        }
    }
    out
}

### known shape: the closed-form update of the text
em_weib_known_k <- function(y, event, k, lam0 = mean(y), iterations = 200, tol = 1e-10) {
    n <- length(y); d <- sum(event); path <- lam <- lam0
    for (i in seq_len(iterations)) {
        lam_new <- ((sum(y^k) + (n - d) * lam^k) / n)^(1 / k)
        path <- c(path, lam_new)
        if (abs(lam_new - lam) < tol * lam) break
```

```
        lam <- lam_new
    }
    path
}
```

Before using the E-step it is worth checking the two formulas against brute force. Draw $T = \lambda_t (a + E)^{1/k_t}$ many times and compare:

```
set.seed(812)
a <- 1.3; k_t <- 1.7; lam_t <- 8
Tsim <- lam_t * (a + rexp(1e6))^(1 / k_t)
c(monte_carlo = mean(log(Tsim)),  formula = log(lam_t) + elog_term(a) / k_t)
```

```
monte_carlo     formula
   2.526064    2.526132
```

```
c(monte_carlo = mean(Tsim^2),
  formula = lam_t^2 * exp(a + pgamma(a, 2 / k_t + 1, lower.tail = FALSE, log.p = TRUE) +
                          lgamma(2 / k_t + 1)))
```

```
monte_carlo     formula
   173.4084    173.4163
```

6.4.6 Simulation 1: one data set

Lifetimes are Weibull with $k = 1.5$ and $\lambda = 10$, and censoring times are exponential with mean 20, independent of the lifetimes, which censors about 30% of $n = 200$ units.

```
set.seed(812)
n <- 200; k_true <- 1.5; lam_true <- 10
Tlife <- rweibull(n, k_true, lam_true)
cens  <- rexp(n, 1 / 20)
y     <- pmin(Tlife, cens)
event <- Tlife <= cens
c(censored_fraction = mean(!event), failures = sum(event))
```

```
censored_fraction          failures
            0.285           143.000
```

Figure 6.10 shows the data against the generating density: a censored unit is plotted at its censoring time, so the red points sit where the follow-up ended and say only that the lifetime lies to their right.

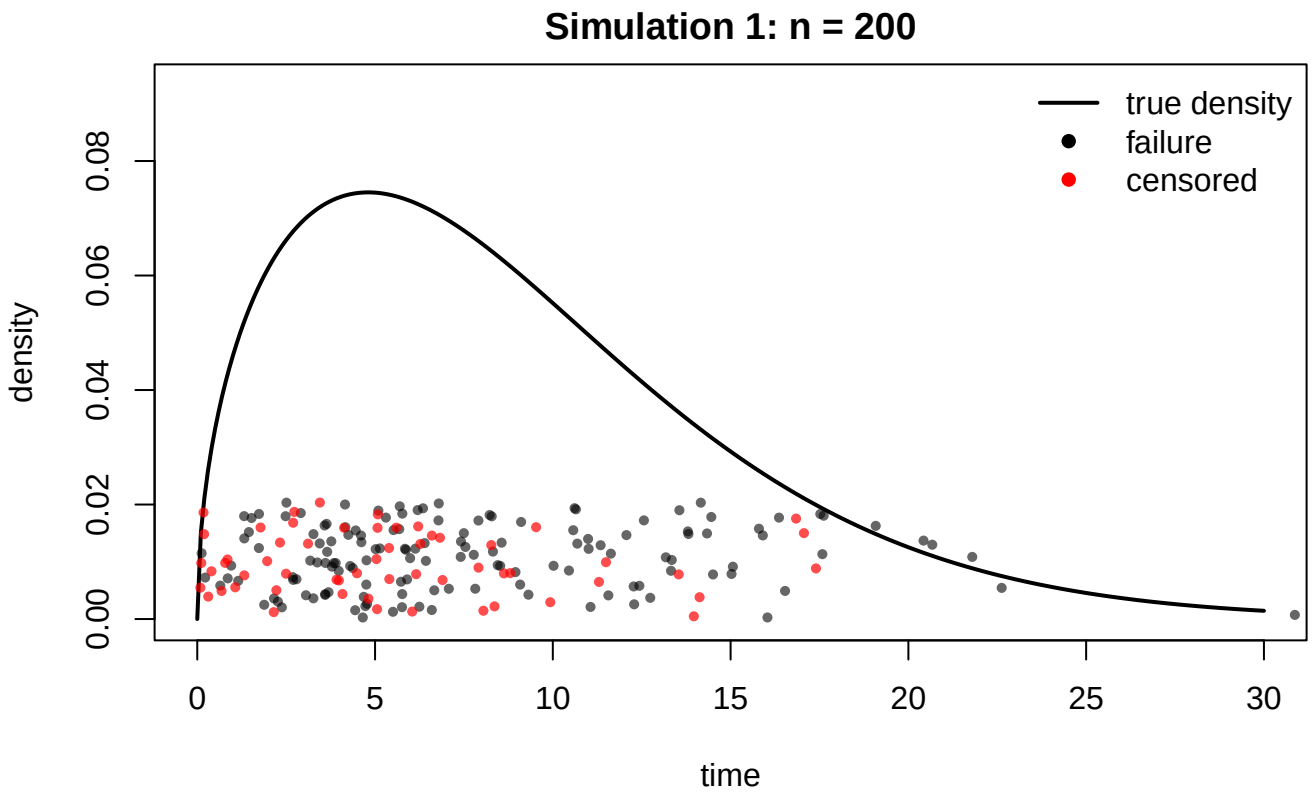


Figure 6.10: Illustrates the data of Simulation 1 with the Weibull density that generated the lifetimes. Black points are observed failures and red points are right-censored times; heights are random and only separate the points.

```
fit <- em_weib(y, event, k0 = 1, lam0 = mean(y))
round(fit[c(1:6, nrow(fit)), ], 5)
```

```
          k   lambda   log_lik
[1,] 1.00000  7.18357 -481.9669
[2,] 1.41501 10.16854 -455.0381
[3,] 1.50462 10.25382 -454.2307
[4,] 1.52980 10.18649 -454.1279
[5,] 1.53877 10.14845 -454.1106
[6,] 1.54225 10.13184 -454.1077
[7,] 1.54457 10.12033 -454.1071
```

```
nrow(fit) - 1                                  # iterations used
```

```
[1] 15
```

```
all(diff(fit[, "log_lik"]) >= -1e-10)          # ascending property
```

```
[1] TRUE
```

The log-likelihood increases at every iteration, as the ascending property requires. To check the limit, compare with a direct numerical maximization of ℓ_{obs} and with the Weibull regression of the `survival` package, which fits the same model by a Newton-type method:

```
direct <- optim(c(0, 0), function(p) -loglik_weib(exp(p[1]), exp(p[2]), y, event),
                control = list(reltol = 1e-14))
sr <- survival::survreg(survival::Surv(y, event) ~ 1, dist = "weibull")
rbind(EM      = fit[nrow(fit), c("k", "lambda")],
      optim   = exp(direct$par),
      survreg = c(1 / sr$scale, exp(coef(sr))))
```

```
              k   lambda
EM      1.544572 10.12033
optim   1.544572 10.12033
survreg 1.544572 10.12033
```

The three agree to several digits. Figure 6.11 shows the EM paths from three different starting values on the contours of ℓ_{obs}.

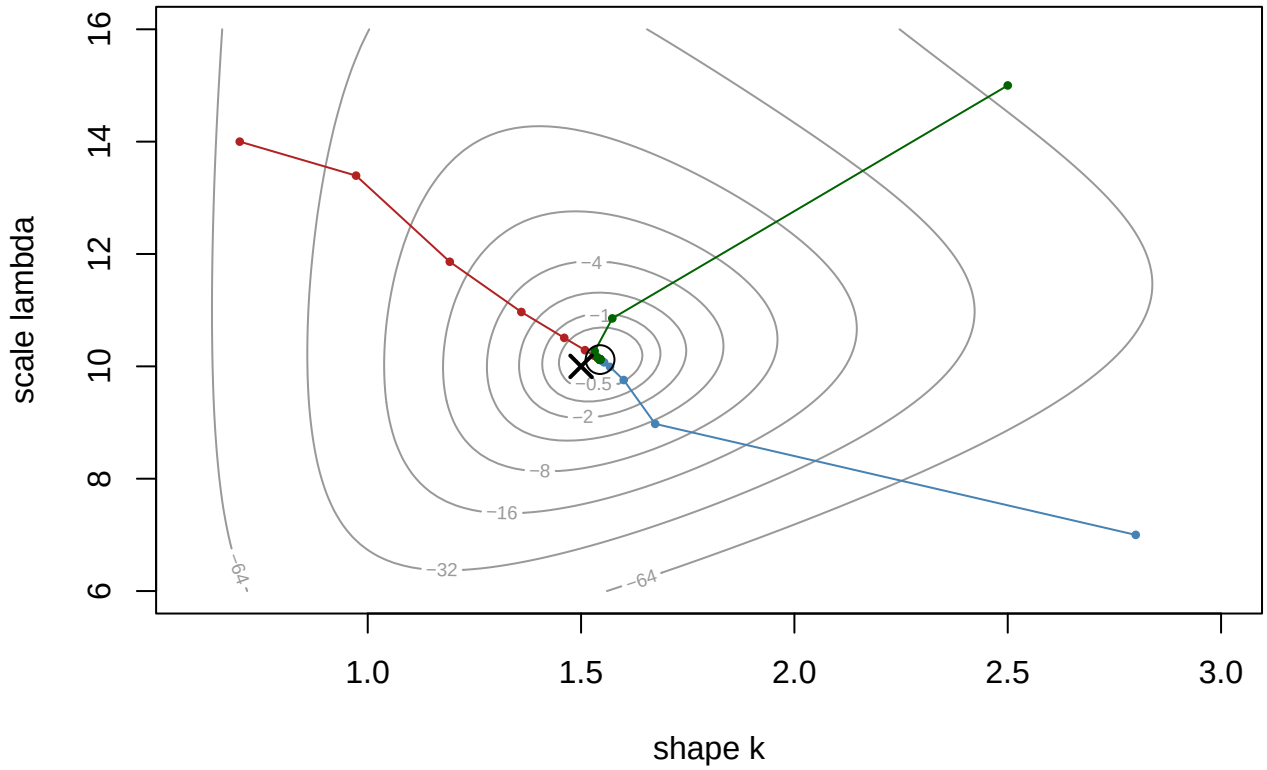


Figure 6.11: Illustrates EM for right-censored Weibull data from three starting values: every run climbs the observed log-likelihood to the same maximum. Contours of the observed log-likelihood in (k, λ) with the EM paths; the cross marks the true value and the circle the maximum likelihood estimate.

6.4.7 Simulation 2: the censoring rate sets the speed

For known shape ($k = 2$) the contraction factor is the censored fraction, as derived above. The simulation draws four data sets of $n = 400$ lifetimes with increasing censoring (Figure 6.12), then runs the closed-form EM on each and compares the observed ratio of successive errors with the censored fraction (Table 6.4).

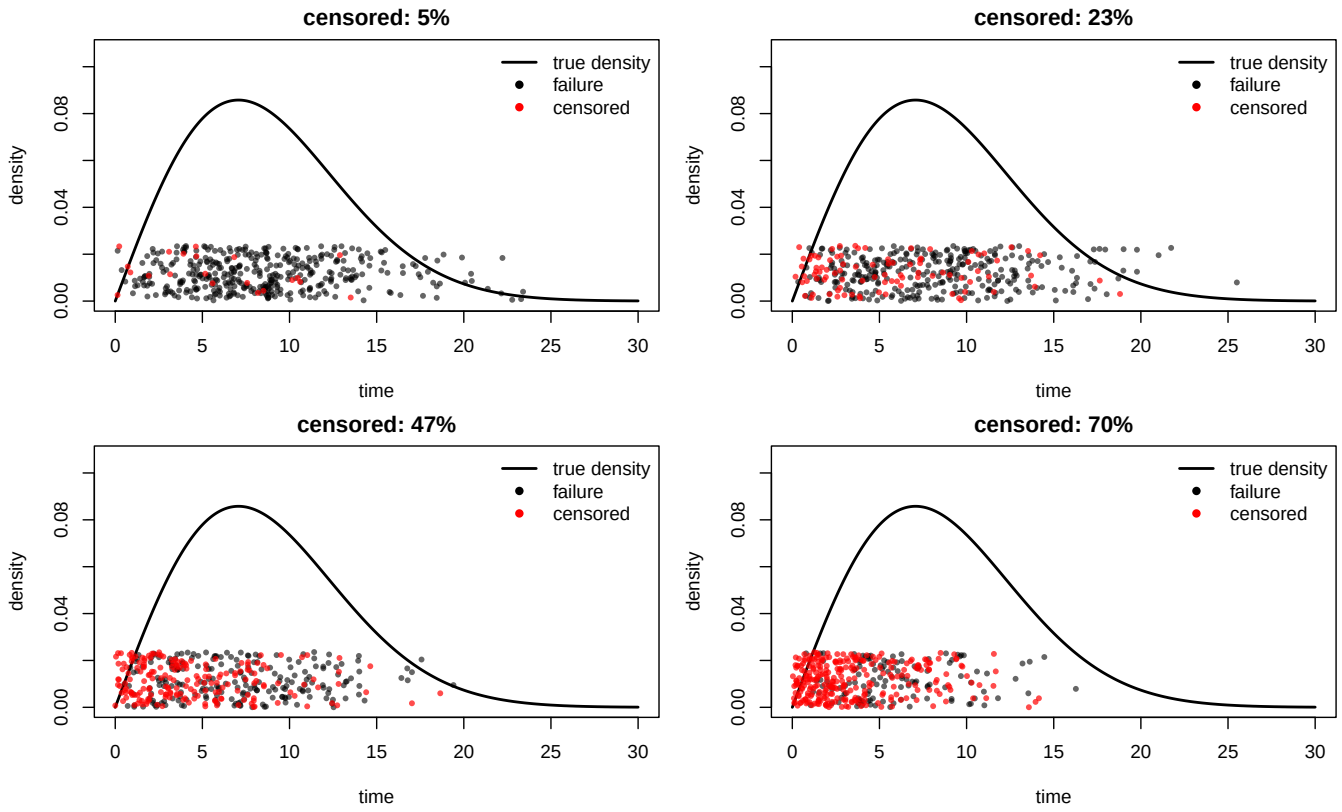


Figure 6.12: Illustrates the four data sets of Simulation 2 ($n = 400$, $k = 2$, $\lambda = 10$), from light to heavy censoring. Black points are failures and red points are right-censored times; the curve is the generating density.

The *error* of iteration t is the distance to the limit, $e_t = |\lambda_t - \hat{\lambda}|$, where $\hat{\lambda}$ is the value the iteration converges to (the last iterate of the run, which equals the MLE $(\sum_i y_i^k/d)^{1/k}$ up to the tolerance). The *error ratio* is e_{t+1}/e_t, the factor by which one more EM iteration shrinks the error; a ratio of 0.5 means the error halves each iteration, and a ratio close to 1 means the iteration crawls. The derivation above shows that $\lambda_{t+1}^k - \hat{\lambda}^k = \frac{n-d}{n}(\lambda_t^k - \hat{\lambda}^k)$ holds exactly, so the ratio for λ^k is the censored fraction at every iteration; for λ itself it approaches the same value once the iterates are close to $\hat{\lambda}$. The table reports the ratio of the errors at iterations 7 and 6 (late enough for the approach to the limit, early enough that rounding error in $\hat{\lambda}$ does not matter), next to the realized censored fraction. The number of iterations needed to gain a fixed accuracy is then about $\log(\text{tolerance})/\log(\text{ratio})$, which is why the iteration count grows sharply as the ratio approaches 1.

```
rate_row <- function(dat, k = 2) {
    p   <- em_weib_known_k(dat$y, dat$event, k, tol = 1e-13)
    err <- abs(p - tail(p, 1))
    c(censored = mean(!dat$event), iterations = length(p) - 1,
      error_ratio = err[8] / err[7])
}
rate_tab <- t(sapply(rate_data, rate_row))
knitr::kable(round(rate_tab, 3))
```

Table 6.4: Shows that the error ratio of the closed-form EM for known shape is close to the censored fraction, so heavier censoring needs more iterations. Known-shape EM on the four data sets of Simulation 2: realized censored fraction, iterations to convergence, and the error ratio e_7/e_6 of two successive iterations.

censored	iterations	error_ratio
0.052	11	0.052
0.228	21	0.227
0.472	39	0.472
0.703	80	0.697

The same holds for the full two-parameter EM, whose speed also falls as censoring grows (Table 6.5, column *iterations*).

6.4.8 Simulation 3: what ignoring the censoring costs

Finally, a Monte Carlo experiment with $n = 100$ and true values $k = 2$, $\lambda = 10$. Each data set is analysed in three ways: EM with the censoring accounted for, discarding the censored units, and treating the censoring times as if they were failure times. The averages over 200 replications are in Table 6.5. Figure 6.13 shows one simulated data set at each censoring level.

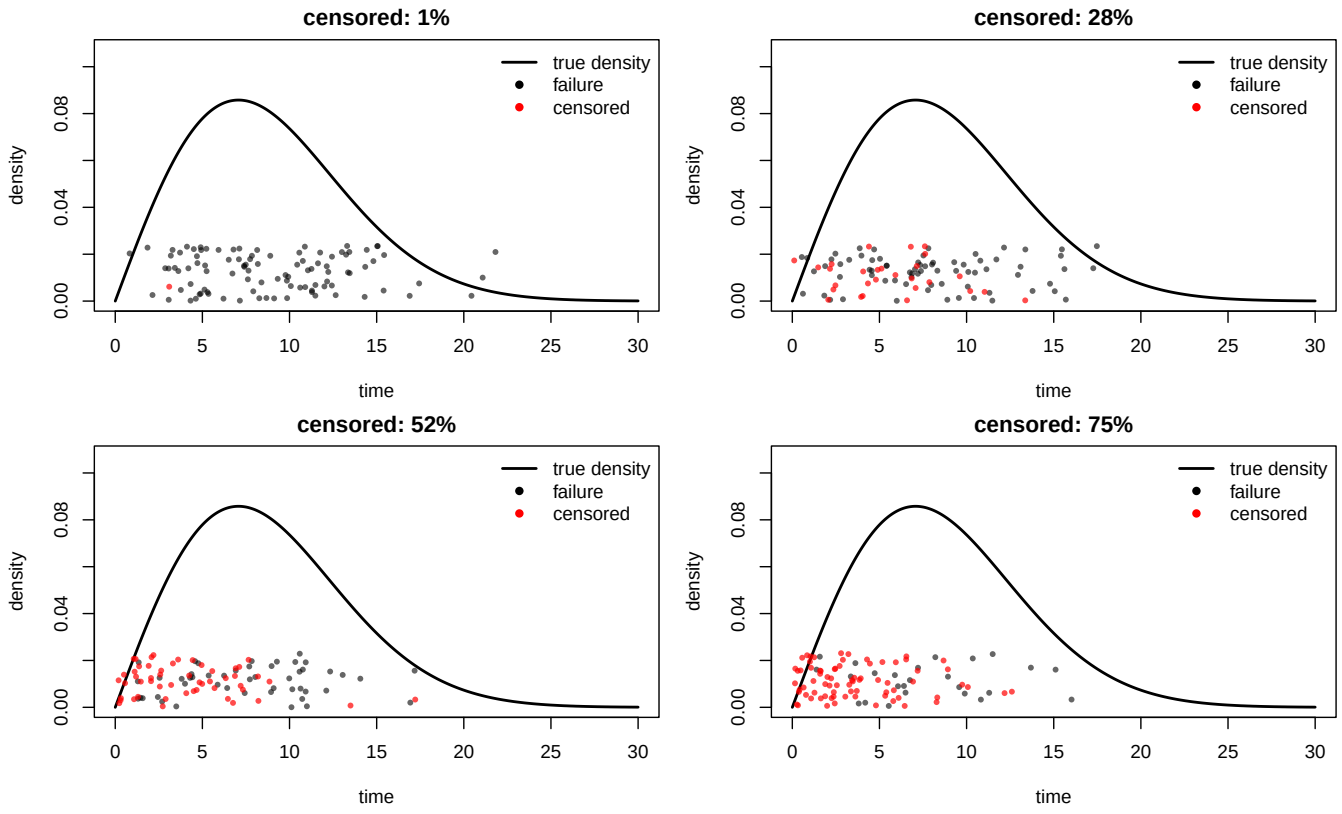


Figure 6.13: Illustrates one simulated data set ($n = 100$, $k = 2$, $\lambda = 10$) at each of the four censoring levels of Simulation 3. Black points are failures and red points are right-censored times; the curve is the generating density.

```
set.seed(812)
sim_weib <- function(mean_cens, nrep = 200, n = 100, k = 2, lam = 10) {
    res <- replicate(nrep, {
        Tl <- rweibull(n, k, lam); cc <- rexp(n, 1 / mean_cens)
        yy <- pmin(Tl, cc); ee <- Tl <= cc
        r   <- em_weib(yy, ee, 1, mean(yy), tol = 1e-8)
        drop <- survival::survreg(survival::Surv(yy[ee], rep(TRUE, sum(ee))) ~ 1,
                                  dist = "weibull")
        asfail <- survival::survreg(survival::Surv(yy, rep(TRUE, n)) ~ 1, dist = "weibull")
        c(censored = mean(!ee), iterations = nrow(r) - 1,
          k_EM = unname(r[nrow(r), "k"]), lambda_EM = unname(r[nrow(r), "lambda"]),
          k_drop = 1 / drop$scale, lambda_drop = unname(exp(coef(drop))),
          k_asfail = 1 / asfail$scale, lambda_asfail = unname(exp(coef(asfail))))
    })
    rowMeans(res)
}
mc <- t(sapply(c(200, 30, 12, 6), sim_weib))
mc <- as.data.frame(round(mc, 2))
names(mc) <- c("censored", "iterations", "k_EM", "lam_EM", "k_drop", "lam_drop",
               "k_fail", "lam_fail")
gt::gt(mc) |>
```

Table 6.5: Shows what ignoring the censoring costs: EM stays near the truth while the two naive analyses are biased. Monte Carlo averages over 200 data sets of $n = 100$ (true $k = 2$, $\lambda = 10$), for four censoring levels: the EM estimates are close to the truth at every level, while discarding the censored units or treating them as failures biases the scale (and the shape) downward, increasingly so with the amount of censoring.

		EM		Drop Censored Obs		As Failures	
censored	iterations	k	λ	k	λ	k	λ
0.04	5.60	2.03	10.05	2.01	9.93	1.95	9.75
0.25	11.71	2.04	10.06	1.97	9.24	1.66	8.25
0.49	21.52	2.05	10.03	1.90	8.21	1.39	6.36
0.71	43.49	2.07	10.06	1.84	6.80	1.21	4.49

```
    gt::tab_spanner("EM", columns = c(k_EM, lam_EM)) |>
    gt::tab_spanner("Drop Censored Obs", columns = c(k_drop, lam_drop)) |>
    gt::tab_spanner("As Failures", columns = c(k_fail, lam_fail)) |>
    gt::cols_label(censored = "censored", iterations = "iterations",
                   k_EM = gt::md("*k*"), k_drop = gt::md("*k*"), k_fail = gt::md("*k*"),
                   lam_EM = " ", lam_drop = " ",
                   lam_fail = " ")
```

With light censoring all three analyses are close. As censoring grows, discarding the censored units removes exactly the long lifetimes and treating the censoring times as failures declares them to have failed early, so both underestimate the scale and the shape; the EM estimates stay near the truth because each censored unit contributes what the model says about its unobserved lifetime, not a made-up value.

6.5 Example 4: Logistic Regression as a Latent-Variable Problem

Logistic regression is usually fitted directly, by Newton's method on the observed log-likelihood (`glm` uses iteratively reweighted least squares, which for the logit link is exactly Newton's method). It can also be viewed as a missing-data problem: each binary response is a thresholded version of an unobserved continuous variable. This example derives the EM algorithm implied by that view and compares it with `glm`. The point is not speed (EM will lose) but to see the general machinery at work on a familiar model, and to see the fraction of missing information decide the convergence rate once more.

6.5.1 The latent-variable model

Let x_i be the covariate vector of unit i (with a leading 1) and $\mu_i = x_i^\top \beta$. Suppose

$$z_i = \mu_i + \varepsilon_i, \qquad \varepsilon_i \overset{iid}{\sim} \text{Logistic}(0,1), \qquad y_i = I(z_i > 0), \tag{6.35}$$

where the standard logistic distribution has CDF and density

$$F(r) = \frac{1}{1+e^{-r}}, \qquad f(r) = F(r)\{1-F(r)\} = \frac{e^{-r}}{(1+e^{-r})^2}. \tag{6.36}$$

That is, $z_i \sim \text{Logistic}(x_i^\top \beta)$ and only its sign is recorded. Because the logistic density is symmetric,

$$P(y_i = 1 \mid \beta) = P(\varepsilon_i > -\mu_i) = 1 - F(-\mu_i) = F(\mu_i) = \frac{1}{1+e^{-x_i^\top \beta}}, \tag{6.37}$$

which is the logistic regression model. The left panel of Figure 6.14 shows $p_i = P(y_i = 1)$ as the area of the latent density to the right of 0. (With a normal ε_i the same construction gives probit regression.)

Observed data: (x_i, y_i), $i = 1, \dots, n$. **Missing**: $z_1, \dots, z_n$.

Complete log-likelihood. Since y_i is a function of z_i, the complete data are just the z_i, and

$$\ell_{\text{comp}}(\beta) = \sum_{i=1}^{n} \log f(z_i - x_i^\top \beta), \qquad \log f(r) = -r - 2\log(1+e^{-r}). \tag{6.38}$$

This is a linear regression with logistic instead of normal errors. It is not linear in z_i, so the E-step needs the whole conditional distribution of z_i, not just its mean (the warning of the overview), and the M-step has no closed form.

Observed log-likelihood.

$$\ell_{\text{obs}}(\beta) = \sum_{i=1}^{n} \{y_i \log p_i + (1-y_i)\log(1-p_i)\}, \qquad p_i = F(x_i^\top \beta). \tag{6.39}$$

6.5.2 E-step

Given y_i and the current β_t, z_i has the Logistic$(\mu_i^{(t)})$ density truncated to $(0,\infty)$ if $y_i = 1$ and to $(-\infty, 0]$ if $y_i = 0$ (right panel of Figure 6.14). The truncation is easiest to describe on the probability scale. Let $U_i = F(z_i - \mu_i^{(t)})$. Without conditioning, U_i is uniform on $(0,1)$; the event $z_i > 0$ is the event $U_i > F(-\mu_i^{(t)}) = 1 - p_i^{(t)}$. Hence

$$U_i \mid y_i, \beta_t \sim \begin{cases} \text{Unif}(1-p_i^{(t)},\ 1), & y_i = 1, \\ \text{Unif}(0,\ 1-p_i^{(t)}), & y_i = 0, \end{cases} \qquad z_i = \mu_i^{(t)} + \text{logit}(U_i). \tag{6.40}$$

Write (a_i, b_i) for the interval of U_i. Then

$$Q(\beta \mid \beta_t) = \sum_{i=1}^{n} \frac{1}{b_i - a_i} \int_{a_i}^{b_i} \log f(\mu_i^{(t)} + \text{logit}(u) - x_i^\top \beta)\, du. \tag{6.41}$$

The integrals have no closed form, but each is one-dimensional over a known interval, so Gauss–Legendre quadrature with K nodes u_{ik} and weights w_k ($\sum_k w_k = 1$) on (a_i, b_i) handles it:

$$Q(\beta \mid \beta_t) \approx \sum_{i=1}^{n} \sum_{k=1}^{K} w_k \log f(z_{ik} - x_i^\top \beta), \qquad z_{ik} = \mu_i^{(t)} + \operatorname{logit}(u_{ik}). \tag{6.42}$$

The E-step therefore replaces each binary y_i by K weighted pseudo-values z_{ik} of the latent variable, all on the side of 0 that y_i dictates.

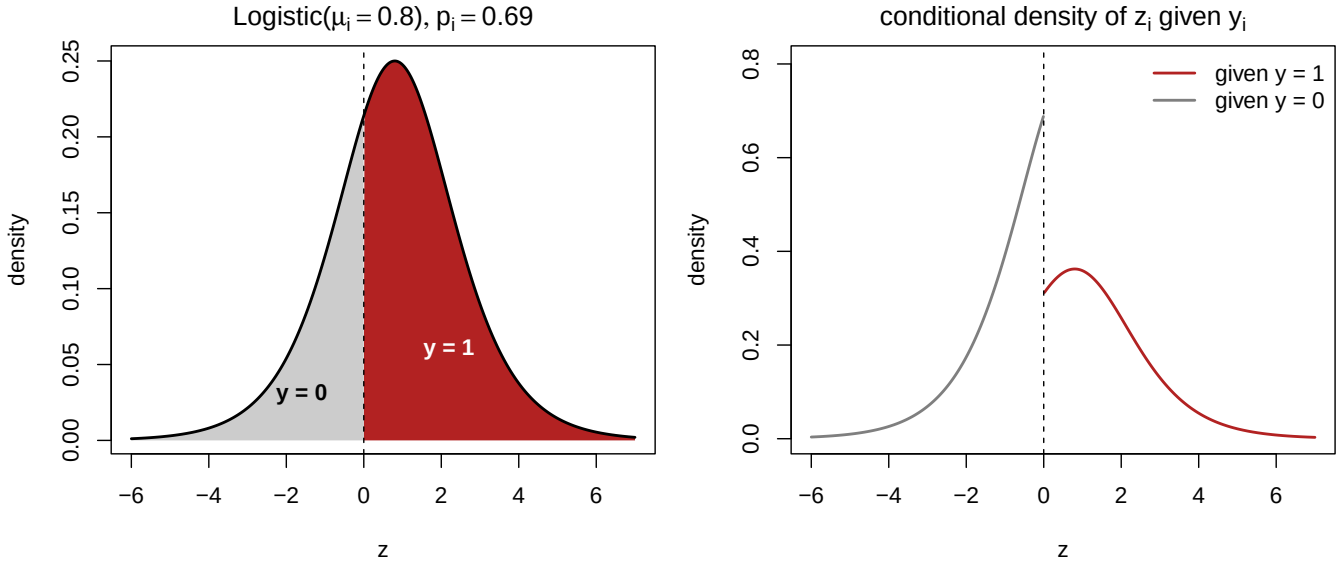


Figure 6.14: Illustrates logistic regression as a thresholded latent variable and the conditional distributions of the latent variable used in the E-step. Left: the latent $z_i \sim \text{Logistic}(\mu_i)$ with $\mu_i = 0.8$; the red area to the right of 0 is $p_i = P(y_i = 1)$ and the grey area is $1 - p_i$. Right: the density of z_i given $y_i = 1$ (red) and given $y_i = 0$ (grey), each the matching part of the left density renormalized.

6.5.3 M-step

The M-step maximizes the weighted pseudo-data log-likelihood of the logistic-error regression. With $r_{ik} = z_{ik} - x_i^\top \beta$ and $\psi(r) = -\frac{d}{dr} \log f(r) = 2F(r) - 1$, so that $\psi'(r) = 2f(r)$, its gradient and Hessian are

$$\nabla_\beta Q = \sum_{i=1}^{n} x_i \sum_{k=1}^{K} w_k \, \psi(r_{ik}), \qquad \nabla^2_\beta Q = -\sum_{i=1}^{n} x_i x_i^\top \sum_{k=1}^{K} w_k \, 2f(r_{ik}). \tag{6.43}$$

The Hessian is negative definite, so $Q(\cdot \mid \beta_t)$ is concave and a few Newton steps from β_t find its maximum. The M-step is thus itself an iterative algorithm, as it was for the Weibull shape in Section 6.4. The price is specific to the logistic link: with normal errors (probit) ℓ_{comp} is a least-squares criterion, and the M-step is the ordinary least-squares fit of $E(z_i \mid y_i, \beta_t)$, a truncated-normal mean, on x_i.

6.5.4 The fixed point is the logistic MLE

At $\beta = \beta_t$, $r_i = z_i - \mu_i^{(t)} = \text{logit}(U_i)$, so $\psi(r_i) = 2U_i - 1$. Averaging over the uniform distributions of the E-step,

$$E(2U_i - 1 \mid y_i, \beta_t) = \begin{cases} 2(1 - p_i^{(t)}/2) - 1 = 1 - p_i^{(t)}, & y_i = 1, \\ 2(1 - p_i^{(t)})/2 - 1 = -p_i^{(t)}, & y_i = 0, \end{cases} \; = \; y_i - p_i^{(t)}, \tag{6.44}$$

and therefore

$$\nabla_\beta Q(\beta \mid \beta_t)\Big|_{\beta=\beta_t} = \sum_{i=1}^{n} x_i (y_i - p_i^{(t)}) = \nabla \ell_{\text{obs}}(\beta_t), \tag{6.45}$$

the logistic-regression score (Fisher's identity). A fixed point of EM solves $\sum_i x_i(y_i - p_i) = 0$, which is the equation `glm` solves. Because the integrand $2u - 1$ is linear in u, Gauss–Legendre quadrature computes this gradient exactly for every K, so the number of nodes affects the path of the iterates but not where they end.

6.5.5 Implementation

```
### Gauss-Legendre nodes and weights on (0, 1), weights summing to 1
### (Golub-Welsch: eigen-decomposition of the Jacobi matrix)
gauss_legendre01 <- function(K) {
    j <- seq_len(K - 1); b <- j / sqrt(4 * j^2 - 1)
    J <- matrix(0, K, K); J[cbind(j, j + 1)] <- b; J[cbind(j + 1, j)] <- b
    e <- eigen(J, symmetric = TRUE)
    list(u = (rev(e$values) + 1) / 2, w = rev(e$vectors[1, ])^2)
}

### observed log-likelihood of logistic regression
loglik_logit <- function(beta, X, y) {
    eta <- drop(X %*% beta)
    sum(plogis(eta, log.p = TRUE)[y == 1]) + sum(plogis(-eta, log.p = TRUE)[y == 0])
}

### E-step: n x K matrix of pseudo-values z_ik = mu_i + logit(u_ik), with the
### nodes u_ik placed in (1 - p_i, 1) when y_i = 1 and in (0, 1 - p_i) when y_i = 0
e_step_logit <- function(beta_t, X, y, gl) {
    mu <- drop(X %*% beta_t)
    p  <- plogis(mu)
    lo <- ifelse(y == 1, 1 - p, 0)
    hi <- ifelse(y == 1, 1, 1 - p)
    mu + qlogis(lo + outer(hi - lo, gl$u))
}

### M-step: Newton's method for the logistic-error regression of the weighted
### pseudo-values on X, started at the current beta (step halving as a safeguard)
```

```
m_step_logit <- function(Z, X, w, beta, tol = 1e-10, maxit = 50) {
    Q <- function(b) sum(dlogis(Z - drop(X %*% b), log = TRUE) %*% w)
    for (it in seq_len(maxit)) {
        R    <- Z - drop(X %*% beta)
        grad <- crossprod(X, drop((2 * plogis(R) - 1) %*% w))
        hess <- crossprod(X, X * drop((2 * dlogis(R)) %*% w))   # minus the Hessian
        step <- drop(solve(hess, grad))
        Q0   <- Q(beta)
        while (Q(beta + step) < Q0 - 1e-12) step <- step / 2
        beta <- beta + step
        if (max(abs(step)) < tol) break
    }
    beta
}

em_logit <- function(X, y, beta0 = rep(0, ncol(X)), K = 20,
                     iterations = 2000, tol = 1e-9) {
    gl <- gauss_legendre01(K); beta <- beta0
    out <- matrix(NA_real_, iterations + 1, ncol(X) + 1,
                  dimnames = list(NULL, c(colnames(X), "log_lik")))
    out[1, ] <- c(beta, loglik_logit(beta, X, y))
    for (i in seq_len(iterations)) {
        Z        <- e_step_logit(beta, X, y, gl)          # E-step
        beta_new <- m_step_logit(Z, X, gl$w, beta)        # M-step
        out[i + 1, ] <- c(beta_new, loglik_logit(beta_new, X, y))
        done <- max(abs(beta_new - beta)) < tol
        beta <- beta_new
        if (done) break
    }
    out[seq_len(i + 1), , drop = FALSE]
}
```

6.5.6 Comparison with `glm`

Two data sets of $n = 500$ share the covariates, an intercept, a standard normal x_1 and a binary x_2, and differ in the strength of the effects: $\beta = (0.3, 0.5, -0.5)$ (weak) and four times that (strong). EM starts at $\beta = 0$ and uses $K = 20$ nodes; `glm` uses its default start and a tight convergence tolerance so that both are accurate to many digits.

```
set.seed(812)
n <- 500
X <- cbind("(Intercept)" = 1, x1 = rnorm(n), x2 = rbinom(n, 1, 0.5))
beta_true <- list(weak = c(0.3, 0.5, -0.5), strong = 4 * c(0.3, 0.5, -0.5))
y_logit   <- lapply(beta_true, function(b) rbinom(n, 1, plogis(drop(X %*% b))))

fits <- lapply(y_logit, function(y) {
    glm_time <- system.time(
        fit <- glm(y ~ X - 1, family = binomial, control = glm.control(epsilon = 1e-14))
    )[["elapsed"]]
    em_time <- system.time(path <- em_logit(X, y))[["elapsed"]]
```

Table 6.6: Shows that the EM algorithm built on the latent logistic variable reaches the same estimates and log-likelihood as `glm`, but needs many more iterations, especially when the effects are strong. Estimates, maximized log-likelihood, iterations and elapsed time (seconds) of `glm` and of EM ($K = 20$ quadrature nodes, started at $\beta = 0$) on the two data sets.

data	method	$\hat\beta_0$	$\hat\beta_1$	$\hat\beta_2$	ℓ_{obs}	iterations	seconds
weak	glm	0.408191	0.457244	-0.599900	-329.974957	5	0.002
weak	EM	0.408191	0.457244	-0.599900	-329.974957	22	0.055
strong	glm	0.969009	2.017459	-1.939325	-220.775391	6	0.001
strong	EM	0.969009	2.017459	-1.939325	-220.775391	109	0.163

```
    list(glm = fit, path = path, glm_time = glm_time, em_time = em_time)
})
```

The two methods agree to about six digits in every coefficient and to the displayed precision in the log-likelihood, as the fixed-point argument predicts. `glm` needs a handful of Newton iterations; EM needs a few times more on the weak data set and many more on the strong one, and each EM iteration is also more expensive (an $n \times K$ E-step and an inner Newton loop). The next subsection explains the difference between the two data sets.

6.5.7 Convergence rate and the missing information

Near the MLE $\hat\beta$, EM is the linear map $\beta_{t+1} - \hat\beta \approx J\,(\beta_t - \hat\beta)$ with

$$J = I - I_{\text{comp}}^{-1}\, I_{\text{obs}}, \qquad I_{\text{obs}} = \sum_{i=1}^{n} x_i x_i^{\top} \hat p_i (1 - \hat p_i), \qquad I_{\text{comp}} = -\nabla^2_{\beta} Q(\beta \mid \hat\beta)\Big|_{\beta=\hat\beta} = \sum_{i=1}^{n} x_i x_i^{\top} c_i, \tag{6.46}$$

where I_{comp} is the complete-data information given the observed data, and J is the matrix fraction of missing information. The error eventually shrinks by the factor ρ, the largest eigenvalue of J, per iteration. At $\beta = \hat\beta$, $f(r_i) = U_i(1 - U_i)$, so $c_i = E\{2U_i(1 - U_i) \mid y_i\}$ with U_i uniform on (a_i, b_i):

$$c_i = \frac{2\{G(b_i) - G(a_i)\}}{b_i - a_i}, \qquad G(u) = \frac{u^2}{2} - \frac{u^3}{3}. \tag{6.47}$$

When $\hat p_i = 1/2$, $c_i = 1/3$ while $\hat p_i(1 - \hat p_i) = 1/4$, so a unit carries $3/4$ of its complete-data information and the missing fraction is $1/4$. When $\hat p_i$ is near 0 or 1, $\hat p_i(1 - \hat p_i) \to 0$ while c_i for the likely outcome stays near $1/3$: knowing that z_i landed on the side of 0 where it almost surely lands says almost nothing about where z_i is, so nearly all the information about z_i is missing. Strong effects push many $\hat p_i$ toward 0 and 1, ρ toward 1, and EM slows down.

```r
em_rate_logit <- function(beta, X, y) {
    p  <- plogis(drop(X %*% beta))
    lo <- ifelse(y == 1, 1 - p, 0); hi <- ifelse(y == 1, 1, 1 - p)
    G  <- function(u) u^2 / 2 - u^3 / 3
    I_comp <- crossprod(X, X * 2 * (G(hi) - G(lo)) / (hi - lo))
    I_obs  <- crossprod(X, X * p * (1 - p))
    max(Mod(eigen(diag(ncol(X)) - solve(I_comp, I_obs), only.values = TRUE)$values))
}
```

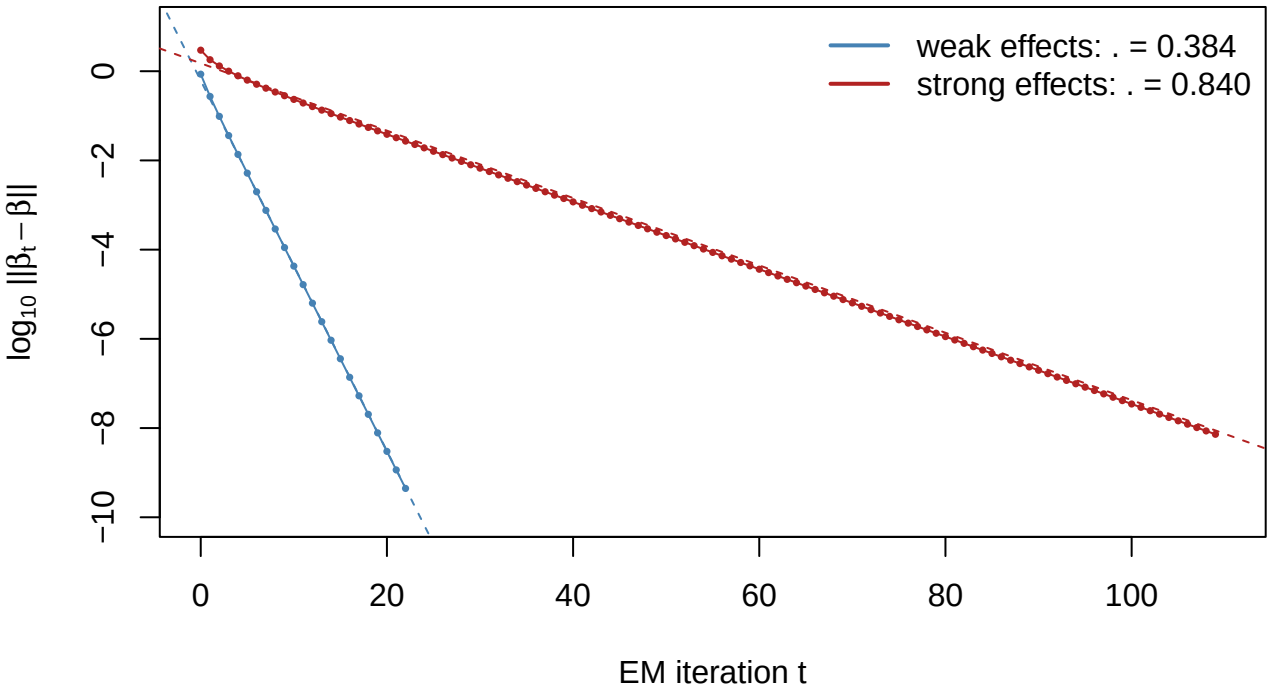


Figure 6.15: Illustrates that the convergence rate of EM for logistic regression is set by the fraction of missing information, which is larger when the effects are strong. Distance $\|\beta_t - \hat{\beta}\|$ of the EM iterates from the MLE (log scale) on the two data sets; the dashed lines have the slopes $\log_{10}\rho$ predicted by the largest eigenvalue ρ of $I - I_{\text{comp}}^{-1} I_{\text{obs}}$.

The observed slopes follow the predicted ones: on the weak data set $\rho \approx 0.38$ and the error falls by an order of magnitude every two or three iterations; on the strong one $\rho \approx 0.84$ and each order of magnitude costs about 13 iterations. Newton's method (`glm`) is immune to this, because it uses I_{obs} directly and converges quadratically.

6.5.8 Remarks

- **The choice of augmentation matters.** The same model admits other latent variables. Writing the logistic likelihood as a mixture over Pólya–Gamma variables ω_i (Polson, Scott and Windle, 2013) gives an EM whose E-step is $E(\omega_i \mid y_i, \beta_t) = \tanh(\mu_i^{(t)}/2)/(2\mu_i^{(t)})$ and whose M-step is the closed-form weighted least-squares fit $\beta_{t+1} = (X^\top \Omega_t X)^{-1} X^\top (y - \tfrac{1}{2})$, $\Omega_t = \operatorname{diag}\{E(\omega_i \mid y_i, \beta_t)\}$. The check below runs it on the same two data sets; it reaches the same MLE in fewer iterations than the latent-logistic EM, each of them cheap.

- **Why bother?** For a point estimate, not much: Newton/IRLS is faster. The latent-variable view earns its keep elsewhere. Thresholded latent variables underlie ordinal and multivariate probit models, and the same augmentations give the Gibbs samplers of Bayesian binary regression (Albert and Chib, 1993, for probit; Pólya–Gamma for logit), where the conditional distributions derived above are exactly what is sampled.

```
em_pg_logit <- function(X, y, beta = rep(0, ncol(X)), iterations = 2000, tol = 1e-9) {
    for (i in seq_len(iterations)) {
        mu    <- drop(X %*% beta)
        omega <- ifelse(abs(mu) < 1e-8, 1 / 4, tanh(mu / 2) / (2 * mu))   # E-step
        beta_new <- drop(solve(crossprod(X, X * omega), crossprod(X, y - 1 / 2)))  # M-step
        done <- max(abs(beta_new - beta)) < tol
        beta <- beta_new
        if (done) break
    }
    c(beta, iterations = i)
}
t(sapply(names(fits), function(s)
    c(round(em_pg_logit(X, y_logit[[s]]), 6),
      latent_logistic_EM_iterations = nrow(fits[[s]]$path) - 1)))
```

```
       (Intercept)       x1        x2 iterations latent_logistic_EM_iterations
weak      0.408191 0.457244 -0.599900         11                            22
strong    0.969009 2.017459 -1.939325         45                           109
```

6.6 More Complex Applications

6.6.1 Model-based clustering

$$Z_i \sim \text{Categorical}(\pi_1, \dots, \pi_K), \qquad Y_i \mid Z_i = k \sim N_d(\mu_k, \Sigma_k). \tag{6.48}$$

This is the multivariate, K-component generalization of Section 6.2. The E-step computes soft cluster memberships $P(Z_i = k \mid y_i, \widehat{\theta}^{(t)})$; the M-step returns weighted means, covariances and proportions. The `mclust` package implements this with a family of constrained covariance structures.

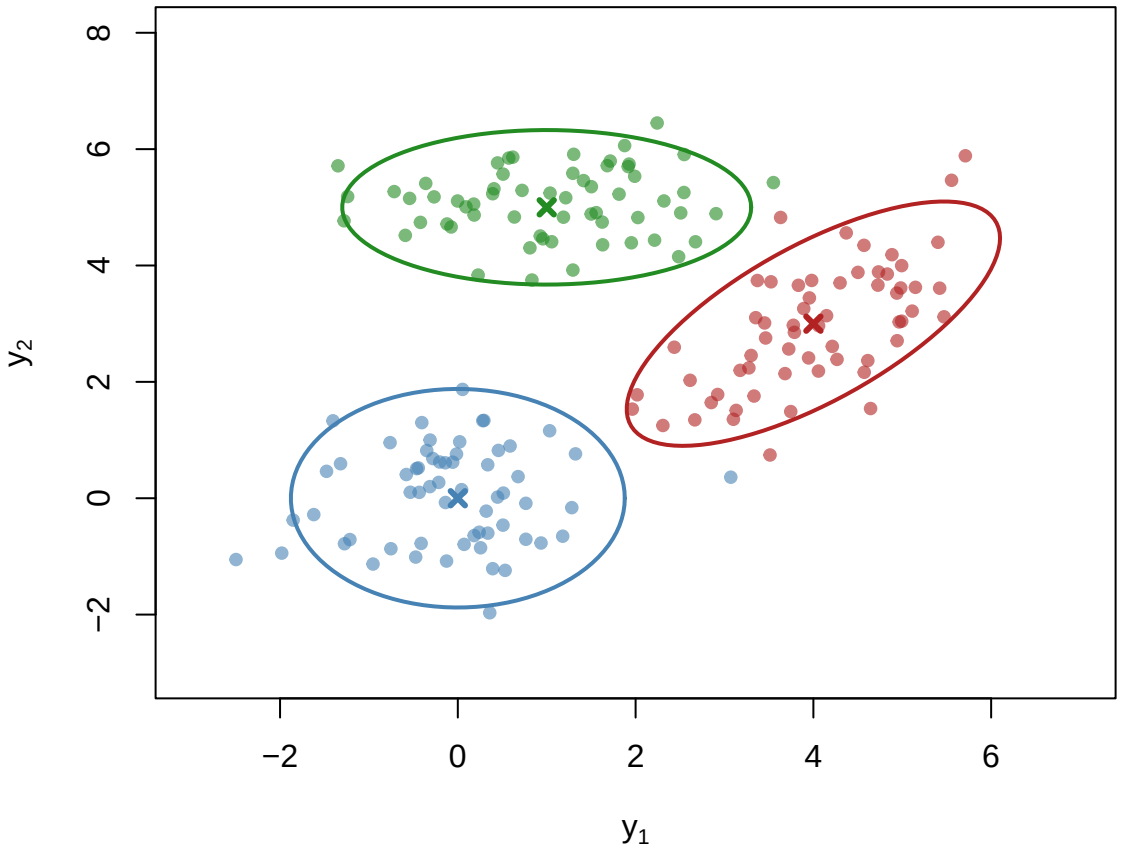


Figure 6.16: Illustrates model-based clustering, the multivariate extension of the normal mixture model, in which EM assigns observations to clusters of different shapes. Three-component multivariate normal mixture with unequal covariances.

6.6.2 Hidden Markov models

Hidden states $h_1 \to h_2 \to \cdots \to h_n$ form a Markov chain, and each observation depends only on its own state:

$$P(y_{1:n} \mid h_{1:n}, \theta) = \prod_{i=1}^{n} P(y_i \mid h_i, \theta). \tag{6.49}$$

The E-step is the forward–backward algorithm, which computes $P(h_i \mid y_{1:n}, \hat{\theta}^{(t)})$ and $P(h_i, h_{i+1} \mid y_{1:n}, \hat{\theta}^{(t)})$ without enumerating all K^n state sequences; the M-step produces weighted transition and emission estimates. The combination is known as the Baum–Welch algorithm, and it is the clearest example of an E-step that is itself a nontrivial computation.

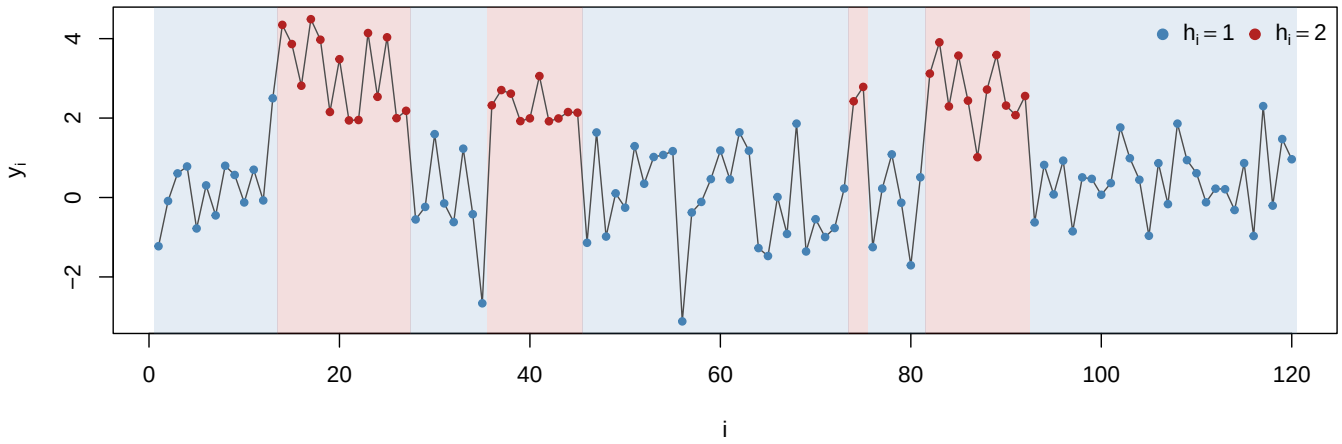


Figure 6.17: Illustrates a hidden Markov model, whose unobserved state sequence generates the data, as a further example to which EM applies. A two-state hidden Markov chain with normal emissions; background shading shows the hidden state.

6.6.3 Factor analysis and the common pattern

$$y_i = \Gamma x_i + \varepsilon_i, \qquad x_i \sim N_q(0, I) \text{ hidden}, \qquad \varepsilon_i \sim N_d(0, \Psi). \tag{6.50}$$

The E-step uses the fact that $x_i \mid y_i, \hat{\theta}^{(t)}$ is normal with closed-form mean and covariance; the M-step is a regression of y on the expected factors.

Across all these applications the recipe is the same:

1. Introduce a hidden variable that makes the complete-data model tractable.
2. **E-step**: compute the conditional distribution of the hidden data, or just the expected sufficient statistics if ℓ_{comp} is linear in them.
3. **M-step**: compute the complete-data MLE with those expected sufficient statistics in place of the unobserved ones.

The art is entirely in step 1. The same observed-data model often admits several choices of Z, and they lead to EM algorithms with quite different convergence rates — faster when the augmentation adds little information, slower when the missing information is a large fraction of the total.

6.7 Justification of the EM Algorithm

This section shows why EM works: the observed log-likelihood never decreases, and EM is coordinate ascent on a single function of θ and a distribution of the hidden data.

6.7.1 Cross-entropy and Kullback-Leibler distance

For distributions P and q of Z, the **cross-entropy** of q relative to P is

$$H(P, q) = -\int P(z) \log q(z)\, dz = -E_P[\log q(Z)]\,, \tag{6.51}$$

so $H(P, P)$ is the entropy of P, and the Kullback-Leibler distance is

$$\mathrm{KL}(P \| q) = \int P(z) \log \frac{P(z)}{q(z)}\, dz = H(P, q) - H(P, P). \tag{6.52}$$

Gibbs' inequality. $\mathrm{KL}(P\|q) \geq 0$, with equality if and only if $q = P$. Equivalently, $H(P, q) \geq H(P, P)$: the cross-entropy is smallest at $q = P$. It follows from $\log x \leq x - 1$:

$$\mathrm{KL}(P\|q) = -\int P(z) \log \frac{q(z)}{P(z)}\, dz \;\geq\; -\int P(z) \left(\frac{q(z)}{P(z)} - 1 \right) dz = -\int [q(z) - P(z)]\, dz = 0. \tag{6.53}$$

6.7.2 The key identity

For every z, $\log P(Y \mid \theta) = \log P(Y, z \mid \theta) - \log P(z \mid Y, \theta)$, because $P(Y, z \mid \theta) = P(Y \mid \theta)\, P(z \mid Y, \theta)$. For a fixed θ, the two curves $\log P(Y, z \mid \theta)$ and $\log P(z \mid Y, \theta)$, as functions of z, differ by the same vertical gap $\ell(Y \mid \theta) = \log P(Y \mid \theta)$ everywhere.

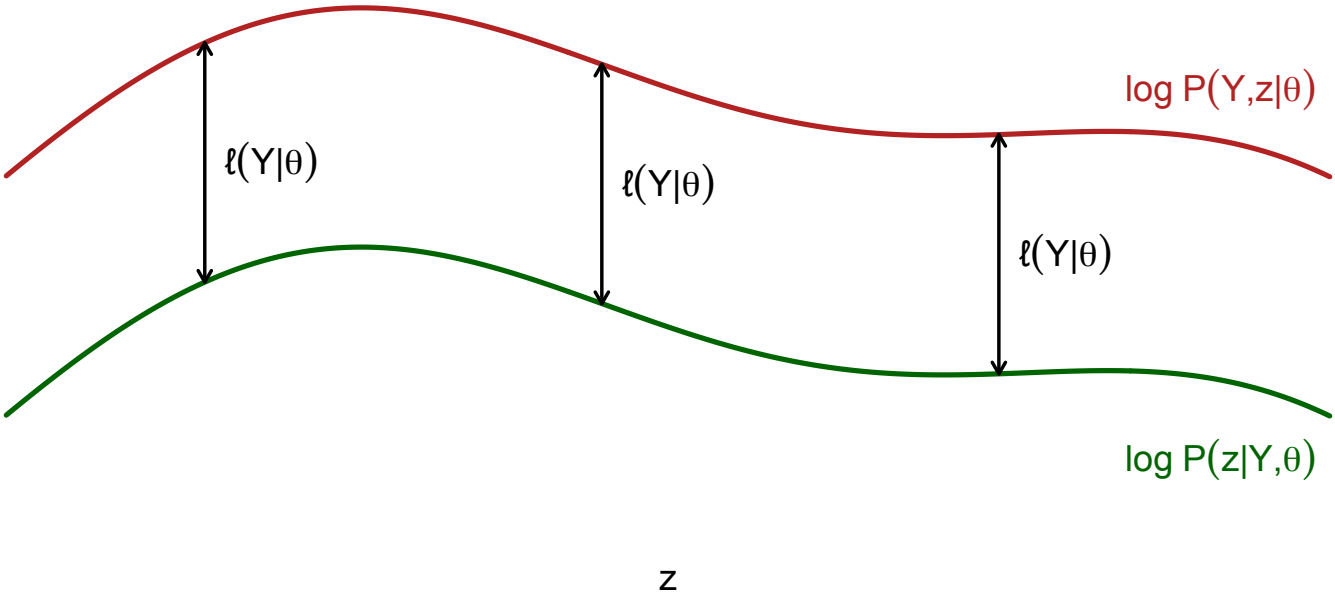


Figure 6.18: Illustrates the key identity for a single value of the hidden data: the complete log-likelihood and the conditional log-density of the hidden data differ by the observed log-likelihood. For a fixed θ, $\log P(Y, z \mid \theta)$ and $\log P(z \mid Y, \theta)$ as functions of z are separated by the constant gap $\ell(Y \mid \theta)$.

Averaging both sides over an *arbitrary* distribution $\tilde{P}$ of Z (the left side does not depend on z, so the average leaves it unchanged) gives, for any $\tilde{P}$,

$$\ell_{\text{obs}}(\theta) = \int \tilde{P}(z) \log P(Y, z \mid \theta)\, dz + H\big(\tilde{P},\ P(\cdot \mid Y, \theta)\big)\,. \tag{6.54}$$

The first term is the average complete log-likelihood under $\tilde{P}$ and the second is a cross-entropy. For a fixed $\tilde{P}$ the two terms, as functions of θ, add up to ℓ_{obs}; Figure 6.19 shows one such split. When Z is discrete, $\log P(z \mid Y, \theta) \le 0$, so the cross-entropy is positive and the average complete log-likelihood lies below ℓ_{obs}; the gap is not constant in θ. The identity is used twice below: for the ascending property of EM and for the alternating-optimization view of EM.

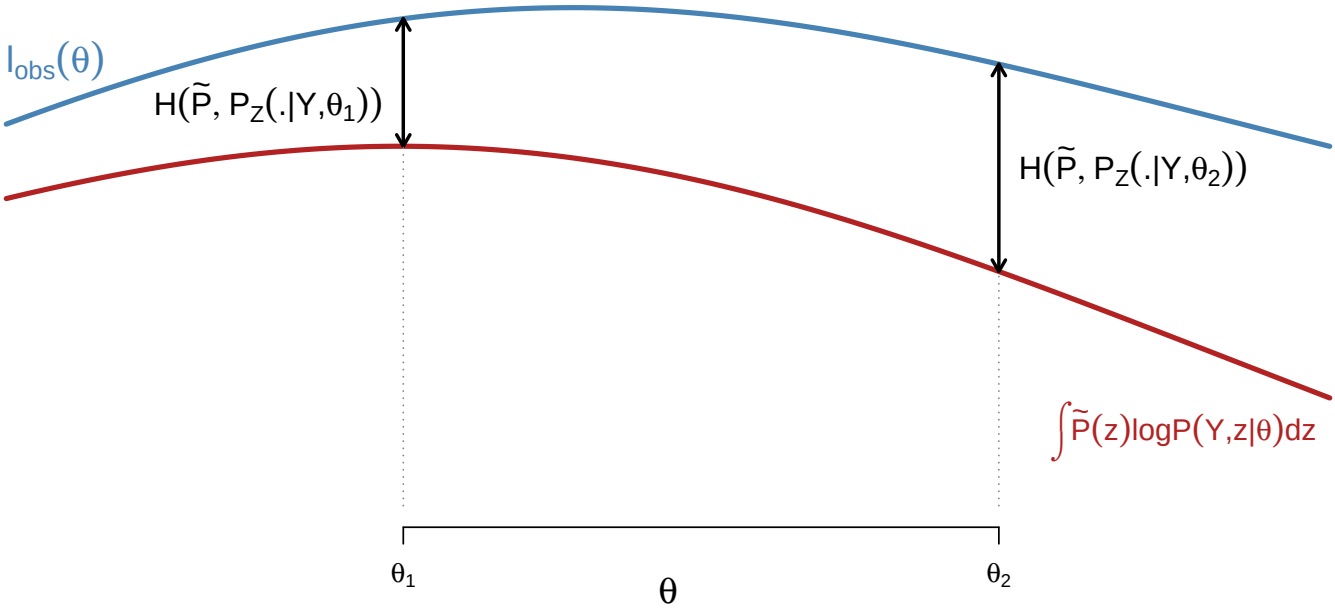


Figure 6.19: Illustrates the key identity: the average complete log-likelihood under a fixed distribution of the hidden data differs from the observed log-likelihood by a cross-entropy. The gap at each θ is the cross-entropy $H(\tilde{P}, P_Z(\cdot \mid Y, \theta))$, which varies with θ and is positive when Z is discrete.

6.7.3 The ascending property

> **Theorem**
>
> $\ell_{\text{obs}}\big(\hat{\theta}^{(t+1)}\big) \ge \ell_{\text{obs}}\big(\hat{\theta}^{(t)}\big)$ at every iteration.

Plug $\tilde{P} = P(\cdot \mid Y, \hat{\theta}^{(t)})$ into Equation 6.54. The first term becomes $Q(\theta \mid \hat{\theta}^{(t)})$:

$$\ell_{\text{obs}}(\theta) = \underbrace{\int P(z \mid Y, \hat{\theta}^{(t)}) \log P(Y, z \mid \theta)\, dz}_{Q(\theta \mid \hat{\theta}^{(t)})} + H\big(P(\cdot \mid Y, \hat{\theta}^{(t)}),\ P(\cdot \mid Y, \theta)\big)\,. \tag{6.55}$$

The M-step makes Q larger, and the cross-entropy is smallest at $\theta = \hat{\theta}^{(t)}$ (Gibbs' inequality), so it can only grow when θ moves. Both terms increase, hence so does ℓ_{obs}.

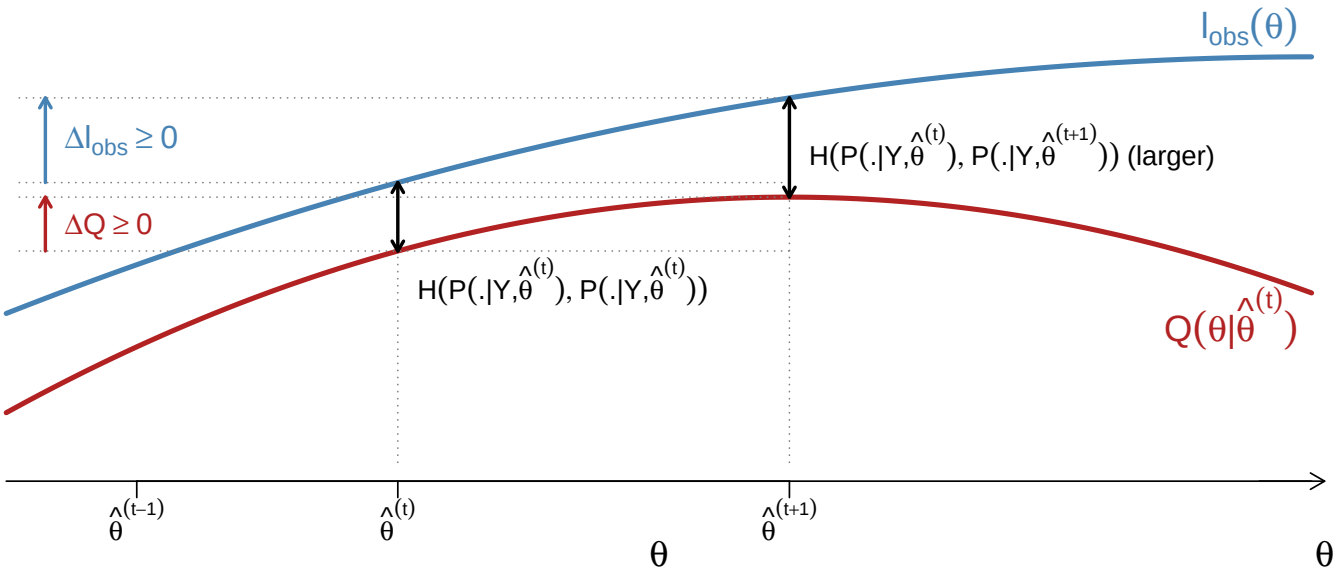


Figure 6.20: Illustrates the ascending property: moving from the current estimate to the next one raises Q and also raises the cross-entropy gap, so the observed log-likelihood rises by at least the gain in Q. The gap between ℓ_{obs} and $Q(\theta \mid \hat{\theta}^{(t)})$ is the cross-entropy H, smallest at $\hat{\theta}^{(t)}$.

Proof: comparing two successive points. Write $\tilde{P} = P(\cdot \mid Y, \hat{\theta}^{(t)})$ and $P_\theta = P(\cdot \mid Y, \theta)$, so $\tilde{P} = P_{\hat{\theta}^{(t)}}$. Subtract the identity at $\theta = \hat{\theta}^{(t)}$ from the identity at $\theta = \hat{\theta}^{(t+1)}$:

$$\begin{aligned} \ell_{\text{obs}}\big(\hat{\theta}^{(t+1)}\big) - \ell_{\text{obs}}\big(\hat{\theta}^{(t)}\big) &= \Big[Q\big(\hat{\theta}^{(t+1)} \mid \hat{\theta}^{(t)}\big) - Q\big(\hat{\theta}^{(t)} \mid \hat{\theta}^{(t)}\big)\Big] + \Big[H\big(\tilde{P}, P_{\hat{\theta}^{(t+1)}}\big) - H\big(\tilde{P}, \tilde{P}\big)\Big] \\ &= \underbrace{\Big[Q\big(\hat{\theta}^{(t+1)} \mid \hat{\theta}^{(t)}\big) - Q\big(\hat{\theta}^{(t)} \mid \hat{\theta}^{(t)}\big)\Big]}_{\geq 0 \text{ by the M-step}} + \underbrace{\mathrm{KL}\big(\tilde{P} \,\|\, P_{\hat{\theta}^{(t+1)}}\big)}_{\geq 0 \text{ by Gibbs' inequality}} \;\geq\; 0. \end{aligned} \tag{6.56}$$

The second line uses $H(\tilde{P}, P_{\hat{\theta}^{(t)}}) = H(\tilde{P}, \tilde{P})$ and $H(P, q) - H(P, P) = \mathrm{KL}(P \| q)$. ■

Note carefully what the theorem does and does not say: the observed likelihood never decreases, but that guarantees convergence only to a stationary point, which may be a local maximum or even a saddle point.

The ascending property was proved by Dempster, Laird and Rubin (1977), who named the algorithm and gave it in this generality; their proof writes the cross-entropy term as $H(\theta \mid \theta_0)$ and bounds it with Jensen's inequality. Special cases of the iteration were in use earlier, for example Hartley (1958) for incomplete count data, Baum et al. (1970) for hidden Markov models (the Baum-Welch algorithm, with its own proof of monotone increase), and Sundberg (1974) for exponential families. The convergence argument of Dempster, Laird and Rubin had a gap, which Wu (1983) corrected: he gave conditions under which the iterates converge to a stationary point of ℓ_{obs}, and the example of a saddle point in the remark above shows why some conditions are needed.

6.7.4 The function $F(\tilde{P}, \theta)$

For any distribution $\tilde{P}$ of the hidden data Z, define

$$F(\tilde{P}, \theta) = \int \tilde{P}(z) \log P(Y, z \mid \theta)\, dz + H(\tilde{P}, \tilde{P}), \tag{6.57}$$

the average complete log-likelihood plus the entropy $H(\tilde{P}, \tilde{P}) = -\int \tilde{P} \log \tilde{P}$ of $\tilde{P}$. It has a second expression:

$$F(\tilde{P}, \theta) = \ell_{\text{obs}}(\theta) - \mathrm{KL}\Big(\tilde{P} \,\|\, P(\cdot \mid Y, \theta)\Big). \tag{6.58}$$

Why they agree. Rearrange Equation 6.54 by moving the cross-entropy to the right-hand side and adding $H(\tilde{P}, \tilde{P})$ to both sides:

$$\begin{aligned} \int \tilde{P}(z) \log P(Y, z \mid \theta)\, dz + H(\tilde{P}, \tilde{P}) &= \ell_{\text{obs}}(\theta) - \Big[H\Big(\tilde{P}, P(\cdot \mid Y, \theta)\Big) - H(\tilde{P}, \tilde{P})\Big] \\ &= \ell_{\text{obs}}(\theta) - \mathrm{KL}\Big(\tilde{P} \,\|\, P(\cdot \mid Y, \theta)\Big). \end{aligned} \tag{6.59}$$

Since $\mathrm{KL} \geq 0$, $F(\tilde{P}, \theta) \leq \ell_{\text{obs}}(\theta)$ for every $\tilde{P}$ and θ, and equality holds exactly when $\tilde{P} = P(\cdot \mid Y, \theta)$.

6.7.5 EM as alternating optimization

Both steps of EM are coordinate ascent on the single function F:

- **E-step** maximizes F over $\tilde{P}$ with $\theta = \hat{\theta}^{(t)}$ fixed. Since $\mathrm{KL} \geq 0$ with equality if and only if $\tilde{P} = P(\cdot \mid Y, \hat{\theta}^{(t)})$, the E-step sets $\tilde{P}_t$ to the conditional distribution of Z.
- **M-step** maximizes F over θ with $\tilde{P}_t$ fixed. The entropy term does not involve θ, so this is $\arg\max_\theta Q(\theta \mid \hat{\theta}^{(t)})$.
- **Tight bound.** After the E-step, $\mathrm{KL} = 0$ at $\theta = \hat{\theta}^{(t)}$, so the lower bound equals the observed log-likelihood there: $F(\tilde{P}_t, \hat{\theta}^{(t)}) = \ell_{\text{obs}}(\hat{\theta}^{(t)})$.

Figure 6.21 shows one full cycle on the censored Poisson model of Section 6.3, with $\theta = \lambda$, the Poisson mean. The hidden data are the $n - m$ censored counts, each 0 or 1, so a distribution $\tilde{P}$ of Z is specified by success probabilities p_i (taking the counts independent), and $F(\tilde{P}, \theta) = \ell_{\text{obs}}(\theta) - \sum_{i>m} \mathrm{KL}(\mathrm{Bern}(p_i) \,\|\, \mathrm{Bern}(\theta/(1+\theta)))$ can be computed exactly. The E-step sets every p_i to $p_t = \hat{\theta}^{(t)}/(1 + \hat{\theta}^{(t)})$, so $\tilde{P}_t$ has entropy $H(\tilde{P}_t, \tilde{P}_t) = -(n-m)\,[p_t \log p_t + (1-p_t)\log(1-p_t)]$ and the bound

$$F(\tilde{P}_t, \theta) = Q\Big(\theta \mid \hat{\theta}^{(t)}\Big) + H(\tilde{P}_t, \tilde{P}_t) \tag{6.60}$$

is the curve $Q(\cdot \mid \hat{\theta}^{(t)})$ shifted vertically by this number.

For a fixed $\tilde{P}_t$, the curve $F(\tilde{P}_t, \cdot)$ is a lower bound of ℓ_{obs}. The E-step at $\hat{\theta}^{(t)}$ replaces the previous distribution by $\tilde{P}_t$ and lifts the bound up to ℓ_{obs} at that point (the green arrows; the dotted gray curve is F for a $\tilde{P}$ whose p_i alternate around p_{t-1}, which is not $P(\cdot \mid Y, \theta)$ for any θ and so lies strictly below ℓ_{obs}). The M-step then moves along the

new bound to its maximum $\hat{\theta}^{(t+1)}$. There ℓ_{obs} lies at least as high as the bound, which lies at least as high as the old value of ℓ_{obs}:

$$\ell_{\text{obs}}\big(\hat{\theta}^{(t+1)}\big) \;\geq\; F\big(\tilde{P}_t, \hat{\theta}^{(t+1)}\big) \;\geq\; F\big(\tilde{P}_t, \hat{\theta}^{(t)}\big) = \ell_{\text{obs}}\big(\hat{\theta}^{(t)}\big)\,. \tag{6.61}$$

In this example the first inequality is nearly an equality: after one step from $\hat{\theta}^{(t-1)} = 0.5$ the iterate is already close to the MLE, and the bound $F(\tilde{P}_t, \cdot)$ is almost indistinguishable from ℓ_{obs} near its maximum.

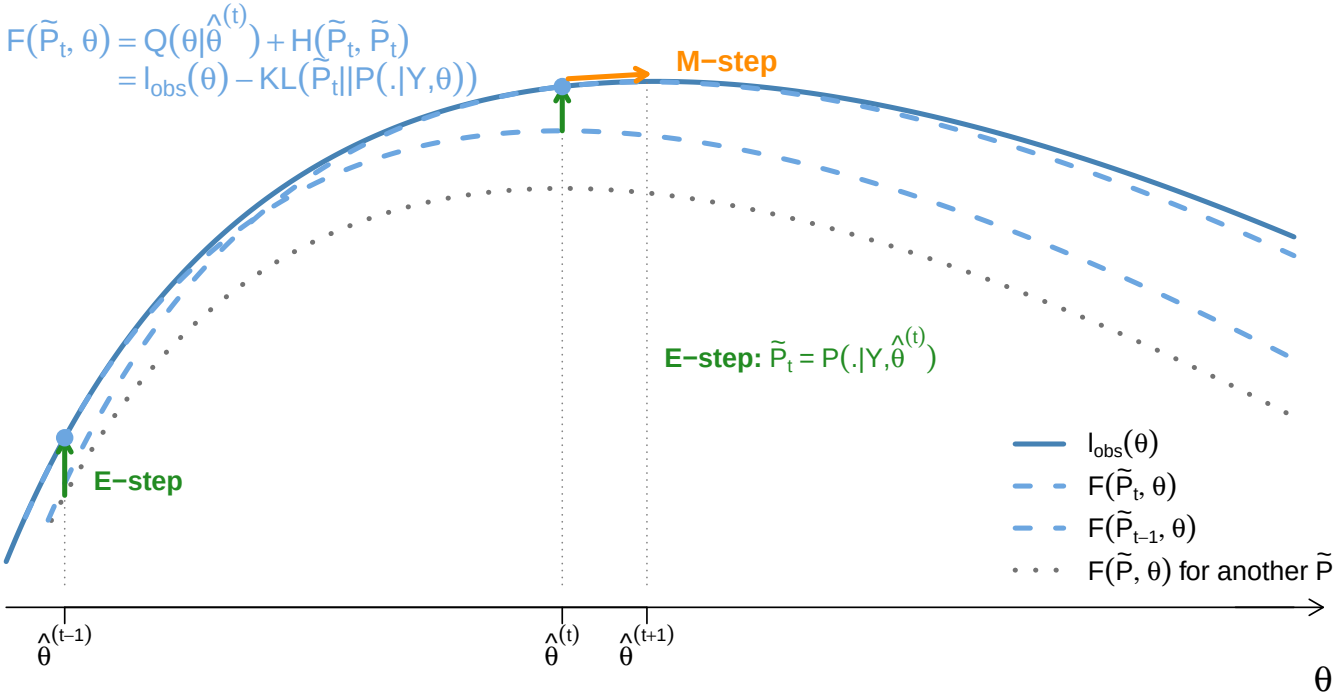


Figure 6.21: Illustrates EM as alternating maximization of F on the censored Poisson example: the E-step lifts the lower bound until it touches the observed log-likelihood at the current estimate, and the M-step moves to the maximum of that bound. The actual $F(\tilde{P}, \theta)$ for the censored Poisson data of Section 6.3 ($\theta = \lambda$), from $\hat{\theta}^{(t-1)} = 0.5$ through two EM steps. Solid blue: $\ell_{\text{obs}}(\theta)$. Light-blue dashed: the bounds $F(\tilde{P}_t, \theta)$ and $F(\tilde{P}_{t-1}, \theta)$. Dotted gray: F for another distribution $\tilde{P}$.

This is the ascending property again, now as a statement about alternating maximization of a single objective, which cannot decrease it. The F formulation is due to Neal and Hinton (1998); related alternating-minimization results for KL divergences were given by Csiszár and Tusnády (1984). The view also explains the variational methods used when the E-step is intractable: restrict $\tilde{P}$ to a manageable family and accept a bound rather than an equality.

6.8 Summary

- EM maximizes ℓ_{obs} by alternating between forming $Q(\theta \mid \hat{\theta}^{(t)}) = E_{Z|Y,\hat{\theta}^{(t)}}[\ell_{\text{comp}}]$ and maximizing it.

- Every iteration weakly increases ℓ_{obs}. Equivalently, EM is coordinate ascent on $F(\tilde{P}, \theta) = \ell_{\text{obs}}(\theta) - \mathrm{KL}(\tilde{P} \,\|\, P(Z \mid Y, \theta))$.
- When ℓ_{comp} is linear in sufficient statistics of Z, the E-step reduces to computing their conditional expectations. Otherwise the full conditional distribution is required, and "plug in the expectation" is wrong.
- Convergence is linear, often slow, and only to a stationary point. Run from several starting values and compare the attained log-likelihoods.
- Standard errors are not a by-product. Use Louis' identity, the SEM algorithm, a bootstrap, or a final Newton step.

6.9 References

- Baum, L. E., Petrie, T., Soules, G. and Weiss, N. (1970). A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *Annals of Mathematical Statistics*, 41(1), 164–171.
- Csiszár, I. and Tusnády, G. (1984). Information geometry and alternating minimization procedures. *Statistics & Decisions*, Supplement Issue 1, 205–237.
- Dempster, A. P., Laird, N. M. and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm (with discussion). *Journal of the Royal Statistical Society, Series B*, 39(1), 1–38.
- Hartley, H. O. (1958). Maximum likelihood estimation from incomplete data. *Biometrics*, 14(2), 174–194.
- Neal, R. M. and Hinton, G. E. (1998). A view of the EM algorithm that justifies incremental, sparse, and other variants. In M. I. Jordan (ed.), *Learning in Graphical Models*, 355–368. Kluwer.
- Sundberg, R. (1974). Maximum likelihood theory for incomplete data from an exponential family. *Scandinavian Journal of Statistics*, 1(2), 49–58.
- Wu, C. F. J. (1983). On the convergence properties of the EM algorithm. *Annals of Statistics*, 11(1), 95–103.

7 Introduction to Bayesian Inference

The last three chapters found a single best-fitting parameter value — the MLE — by optimization: gradient methods and Newton's method for a smooth log-likelihood, EM when part of the data is missing. Bayesian inference asks a different question: not just *where* is the best θ, but how uncertain are we about it, given whatever we already believed before seeing the data? The answer is a full distribution over θ, not a point, and extracting anything useful from that distribution — a mean, a probability, a prediction — is almost always a matter of computing an integral or an optimization. That is why Bayesian inference sets the agenda for the rest of this book: numerical quadrature, Laplace's method, rejection sampling, importance sampling, and Markov chain Monte Carlo are all, at bottom, different strategies for computing with a posterior distribution that has no closed form.

7.1 The Bayesian Framework

Bayesian inference treats the unknown parameter θ itself as a random variable and proceeds in four steps:

1. **Specify a prior** $\pi(\theta)$ — what is known or believed about θ before seeing any data.
2. **Specify a probability model for the data**, $P(y_1, \dots, y_n \mid \theta)$, exactly the sampling model used to build a likelihood in the MLE chapters.
3. **Find the posterior** by Bayes' rule,

$$P(\theta \mid y_{1:n}) = \frac{P(y_{1:n} \mid \theta)\, \pi(\theta)}{\displaystyle\int P(y_{1:n} \mid \theta)\, \pi(\theta)\, d\theta}. \tag{7.1}$$

4. **Interpret the posterior**: summarize θ, make predictions.

Writing $L(\theta) = P(y_{1:n} \mid \theta)$ for the likelihood and $f(\theta) = L(\theta)\pi(\theta)$ for the **unnormalized posterior** — the notation used throughout the rest of this book — step 3 becomes

$$P(\theta \mid y_{1:n}) = \frac{f(\theta)}{P(y_{1:n})}, \qquad P(y_{1:n}) = \int f(\theta)\, d\theta. \tag{7.2}$$

$P(y_{1:n})$, the **marginal likelihood**, is the likelihood averaged over the prior rather than maximized over θ; it does not depend on θ, so it plays no role in *shaping* the posterior, but it is usually the hardest single quantity in the whole framework to compute — it is a p-dimensional integral with no generic closed form, and it is the running example of the next three chapters (numerical quadrature, Laplace approximation, and the Monte Carlo methods after that).

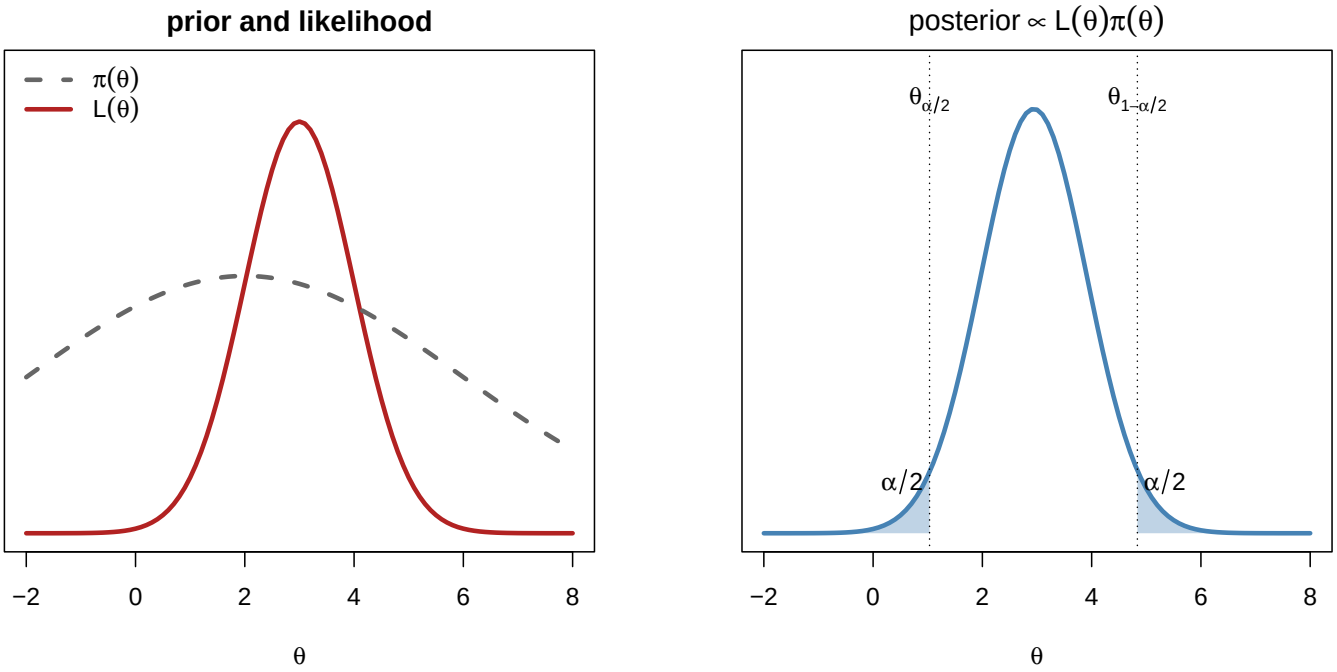


Figure 7.1: Shows how Bayes' rule combines a prior with a likelihood into a posterior, and how a credible interval is read off from the posterior. Left: a flat-ish prior $\pi(\theta)$ (dashed) and a likelihood $L(\theta)$ (solid) that is much more concentrated — the data are far more informative than the prior here. Right: the posterior $\propto L(\theta)\pi(\theta)$, essentially the (renormalized) likelihood since the prior is so flat over the region where L has mass. Shaded tails cut off $\alpha/2$ of the posterior probability each, giving a $(1-\alpha)$ credible interval $(\theta_{\alpha/2}, \theta_{1-\alpha/2})$.

7.2 Summarizing the Posterior

7.2.1 Point estimates from loss functions

A single number for θ is a decision, and a principled way to make a decision is to minimize expected loss. Given a loss function $L(\theta, \hat{\theta})$ measuring the cost of reporting $\hat{\theta}$ when the truth is θ, the **Bayes estimate** minimizes the *posterior expected* loss,

$$\hat{\theta} = \arg\min_{\hat{\theta}} \; E[L(\theta, \hat{\theta}) \mid y_{1:n}] = \arg\min_{\hat{\theta}} \int L(\theta, \hat{\theta})\, P(\theta \mid y_{1:n})\, d\theta. \tag{7.3}$$

Three standard losses give three familiar summaries:

Loss $L(\theta, \hat{\theta})$	Bayes estimate $\hat{\theta}$
$(\theta - \hat{\theta})^2$ (squared error)	posterior mean $E(\theta \mid y_{1:n})$
$\lvert\theta - \hat{\theta}\rvert$ (absolute error)	posterior median
$I(\theta \neq \hat{\theta})$ (0–1 loss)	posterior mode (MAP)

Interval estimates go one step further: a $(1-\alpha)$ **credible interval** $(\theta_{\alpha/2}, \theta_{1-\alpha/2})$ contains θ with posterior probability $1-\alpha$ — a direct probability statement about θ itself, unlike a frequentist confidence interval, whose $1-\alpha$ coverage is a statement about the *procedure* across repeated samples, not about θ given the data actually observed.

Every entry in that table is either an integral (a mean, a median, a probability) or an optimization (the mode) over the posterior. This is the sense in which Bayesian inference is inherently computational: even the most basic summaries require operations that, outside of conjugate models like the three worked below, have no closed form.

7.2.2 Posterior prediction

Often the real target is not θ itself but a future observation. The **posterior predictive distribution** of y_{n+1} is

$$P(y_{n+1} \mid y_{1:n}) = \frac{\int P(y_{n+1}, y_{1:n} \mid \theta)\, \pi(\theta)\, d\theta}{\int P(y_{1:n} \mid \theta)\, \pi(\theta)\, d\theta}. \tag{7.4}$$

If $y_1, \ldots, y_n, y_{n+1}$ are conditionally independent given θ — the usual iid-sampling assumption — the joint density factors and this simplifies to

$$P(y_{n+1} \mid y_{1:n}) = \int P(y_{n+1} \mid \theta)\, P(\theta \mid y_{1:n})\, d\theta, \tag{7.5}$$

an *average* of the sampling density over the posterior, rather than the sampling density evaluated at one plug-in estimate $\hat{\theta}$. Given posterior draws $\theta_1, \ldots, \theta_J \sim P(\theta \mid y_{1:n})$ (the subject of the Monte Carlo chapters later in the book),

$$P(y_{n+1} \mid y_{1:n}) \approx \frac{1}{J} \sum_{j=1}^{J} P(y_{n+1} \mid \theta_j). \tag{7.6}$$

Because it mixes over the whole posterior instead of conditioning on a single $\hat{\theta}$, the predictive distribution is necessarily wider than any individual $P(y_{n+1} \mid \theta_j)$: it carries both the sampling variability *and* the remaining uncertainty about θ. The Poisson–Gamma example below makes this concrete with a closed-form predictive distribution that is visibly overdispersed relative to a plug-in Poisson.

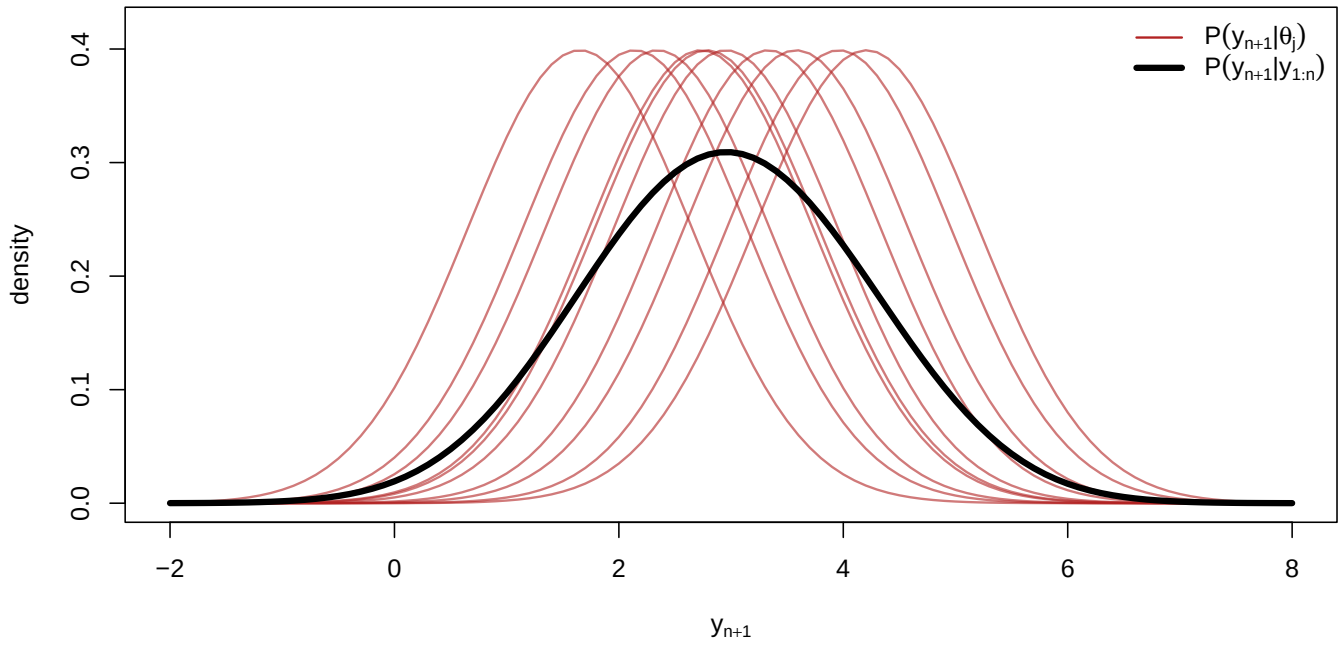


Figure 7.2: Shows how the posterior predictive distribution is built by averaging the sampling density over the posterior, so that it accounts for uncertainty about the parameter. Red: the sampling density $P(y_{n+1} \mid \theta_j)$ for ten posterior draws $\theta_j \sim N(3, 0.8^2)$. Black: their average, the posterior predictive density $P(y_{n+1} \mid y_{1:n})$. The predictive is visibly wider than any single red curve — it inherits both the spread of $P(y_{n+1} \mid \theta)$ and the spread of the θ_j.

7.3 Example: Bernoulli Data with a Beta Prior

7.3.1 Prior

Let $y_1, \dots, y_n \mid \theta \overset{iid}{\sim} \text{Bern}(\theta)$, with $\theta \in (0,1)$ a success probability, and give θ a $\text{Beta}(\alpha_1, \alpha_0)$ prior, $\alpha_1, \alpha_0 > 0$ given:

$$\pi(\theta) = \frac{\Gamma(\alpha_0 + \alpha_1)}{\Gamma(\alpha_0)\Gamma(\alpha_1)}\, \theta^{\alpha_1 - 1}(1-\theta)^{\alpha_0 - 1}, \qquad 0 < \theta < 1, \tag{7.7}$$

$$E(\theta) = \frac{\alpha_1}{\alpha_0 + \alpha_1}, \qquad V(\theta) = \frac{\alpha_0 \alpha_1}{(\alpha_0 + \alpha_1)^2(\alpha_0 + \alpha_1 + 1)}. \tag{7.8}$$

The parameters have a direct reading as *pseudo-data*: α_1 behaves like a prior count of successes, α_0 a prior count of failures, and $\alpha_0 + \alpha_1$ a prior "sample size" — the more of it, the more concentrated $\pi(\theta)$ is around $\alpha_1/(\alpha_0 + \alpha_1)$, exactly as a real sample of that size would concentrate a Bernoulli proportion. $\text{Beta}(1,1)$ is $\text{Unif}(0,1)$, a genuinely flat prior; small α's (e.g. $\text{Beta}(0.5, 0.5)$) give a U-shaped prior that expects θ near 0 or 1; large, unequal α's concentrate the prior away from $\frac{1}{2}$.

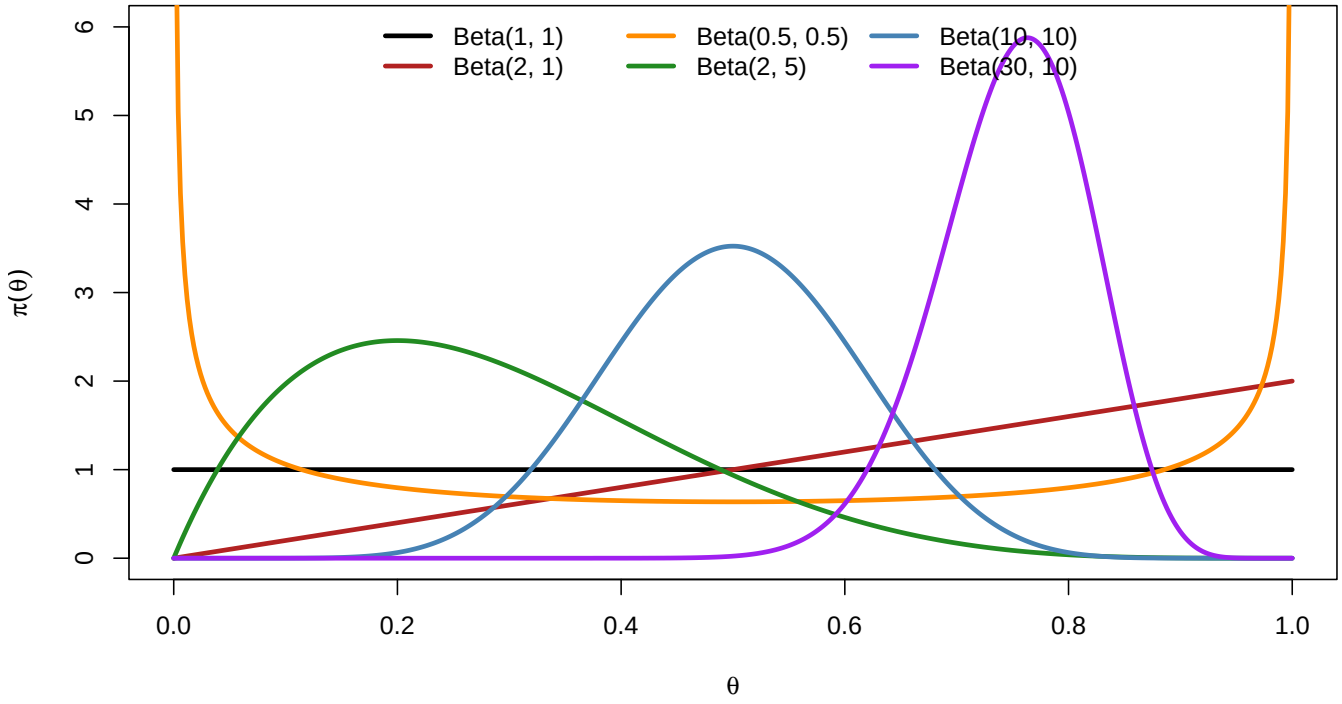


Figure 7.3: Shows the range of prior beliefs about a probability that the Beta family can express, to guide the choice of a prior. Six Beta priors spanning flat, U-shaped, and concentrated shapes. Beta(1,1) is uniform; Beta(0.5,0.5) is U-shaped, favouring θ near the boundaries; Beta(2,5) and Beta(30,10) are increasingly concentrated, at $\theta \approx 2/7$ and $\theta \approx 0.75$ respectively.

7.3.2 Likelihood and posterior

With $P(y_i \mid \theta) = \theta^{y_i}(1-\theta)^{1-y_i}$ and $\mathbf{y} = (y_1, \dots, y_n)$, the likelihood depends on the data only through the total number of successes:

$$L(\theta) = \prod_{i=1}^{n} \theta^{y_i}(1-\theta)^{1-y_i} = \theta^{n_1}(1-\theta)^{n_0}, \qquad n_1 = \sum_i y_i, \;\; n_0 = n - n_1. \tag{7.9}$$

Multiplying by the prior and collecting powers of θ and $1-\theta$,

$$f(\theta) = \pi(\theta)L(\theta) \;\propto\; \theta^{\alpha_1 - 1}(1-\theta)^{\alpha_0 - 1} \cdot \theta^{n_1}(1-\theta)^{n_0} = \theta^{(\alpha_1 + n_1) - 1}(1-\theta)^{(\alpha_0 + n_0) - 1}. \tag{7.10}$$

This unnormalized posterior is, up to the constant $P(\mathbf{y})$, exactly the Beta kernel with updated parameters $\alpha_1' = \alpha_1 + n_1$, $\alpha_0' = \alpha_0 + n_0$ — and since a Beta density is the *only* normalized distribution with that kernel,

$$\theta \mid \mathbf{y} \sim \text{Beta}(\alpha_1 + n_1,\; \alpha_0 + n_0) \tag{7.11}$$

follows immediately, with no integration needed. This is what it means for the Beta prior to be **conjugate** to the Bernoulli likelihood: updating the prior to the posterior is nothing but arithmetic on the two parameters, $\alpha_1 \to \alpha_1 + n_1$, $\alpha_0 \to \alpha_0 + n_0$, and the marginal likelihood $P(\mathbf{y})$ never has to be computed explicitly, because matching the two Beta kernels pins down the normalizing constant for free. Few models have this property; it is a stroke of algebraic luck rather than a general feature of Bayesian inference, which is exactly why the rest of this book is needed.

7.3.3 Posterior summaries

The pseudo-data reading of α_1, α_0 carries through to the posterior mean, which is an exact weighted average of the prior mean and the MLE $\hat{\theta}_{\text{MLE}} = n_1/n$:

$$E(\theta \mid \mathbf{y}) = \frac{\alpha_1 + n_1}{\alpha_0 + \alpha_1 + n} = \underbrace{\frac{\alpha_0 + \alpha_1}{\alpha_0 + \alpha_1 + n}}_{w} \cdot \frac{\alpha_1}{\alpha_0 + \alpha_1} + (1 - w) \cdot \hat{\theta}_{\text{MLE}}. \tag{7.12}$$

The weight w on the prior mean is the prior's pseudo-sample-size relative to the total (prior plus real) sample size; as $n \to \infty$, $w \to 0$ and the posterior mean converges to the MLE regardless of the prior. The posterior variance,

$$V(\theta \mid \mathbf{y}) = \frac{(\alpha_1 + n_1)(\alpha_0 + n_0)}{(n + \alpha_0 + \alpha_1)^2(n + \alpha_0 + \alpha_1 + 1)} = O\Big(\frac{1}{n}\Big), \tag{7.13}$$

shrinks at the same $1/n$ rate as the sampling variance of the MLE, $\hat{\theta}_{\text{MLE}} \dot{\sim} N(\theta_0,\, 1/(nI(\theta_0)))$ from the MLE chapters — for large n the two approaches to uncertainty agree, with the prior's influence vanishing.

7.3.4 The marginal likelihood in closed form

Because conjugacy gives the *whole* posterior in closed form, it also gives the marginal likelihood $P(\mathbf{y}) = \int_0^1 f(\theta)\, d\theta$ in closed form, by integrating the unnormalized Beta kernel directly against its own normalizing constant:

$$P(\mathbf{y}) = \frac{\Gamma(\alpha_0 + \alpha_1)}{\Gamma(\alpha_0)\Gamma(\alpha_1)} \int_0^1 \theta^{(\alpha_1 + n_1) - 1}(1 - \theta)^{(\alpha_0 + n_0) - 1}\, d\theta = \frac{\Gamma(\alpha_0 + \alpha_1)}{\Gamma(\alpha_0)\Gamma(\alpha_1)} \cdot \frac{\Gamma(\alpha_1 + n_1)\,\Gamma(\alpha_0 + n_0)}{\Gamma(\alpha_0 + \alpha_1 + n)}, \tag{7.14}$$

using $\int_0^1 \theta^{a-1}(1 - \theta)^{b-1}\, d\theta = B(a, b) = \Gamma(a)\Gamma(b)/\Gamma(a + b)$. On the log scale, with $\log B(a, b) = \log\Gamma(a) + \log\Gamma(b) - \log\Gamma(a + b)$,

$$\log P(\mathbf{y}) = \log B(\alpha_1 + n_1,\ \alpha_0 + n_0) - \log B(\alpha_1,\ \alpha_0). \tag{7.15}$$

This exact Beta–Bernoulli conjugate identity is the same trick used for Gamma–Poisson and Normal–Normal below: matching a recognizable kernel against its own normalizing constant hands over $P(\mathbf{y})$ for free, with no numerical integration anywhere.

7.3.5 Posterior predictive, in closed form

For a new $y_{n+1} \mid \theta \sim \text{Bern}(\theta)$, the general predictive formula from earlier in this chapter simplifies beautifully in the conjugate case, because $E(\theta \mid \mathbf{y})$ *is* the predictive success probability:

$$P(y_{n+1} = 1 \mid \mathbf{y}) = \int_0^1 \theta\, P(\theta \mid \mathbf{y})\, d\theta = E(\theta \mid \mathbf{y}), \qquad y_{n+1} \mid \mathbf{y} \sim \text{Bern}\big(E(\theta \mid \mathbf{y})\big). \tag{7.16}$$

This exact equality ($E(\theta \mid \mathbf{y})$ doing double duty as both a point estimate of θ and the predictive probability of success) is special to Bernoulli data; the Poisson–Gamma and Normal–Normal examples below give predictive distributions that are genuinely wider than their plug-in sampling models.

7.3.6 Worked numbers: the complete inference output

Take a near-flat Beta$(1, 1)$ prior and $n = 10$ trials with $y = 7$ successes. The table below collects everything derived above — marginal likelihood, posterior, its mean/sd and credible interval, and the predictive distribution — in one place, and the same five quantities are computed the same way for the Gamma–Poisson and Normal–Normal examples that follow:

```
a1 <- 1; a0 <- 1                # Beta(1, 1): uniform prior
n_bb <- 10; y_bb <- 7           # 7 successes out of 10 trials
post_a1 <- a1 + y_bb; post_a0 <- a0 + (n_bb - y_bb)
bb_logmarg <- lbeta(post_a1, post_a0) - lbeta(a1, a0)   # log P(y), exact
bb_mean <- post_a1 / (post_a1 + post_a0)
bb_sd   <- sqrt(post_a1 * post_a0 / ((post_a1 + post_a0)^2 * (post_a1 + post_a0 + 1)))
bb_ci   <- qbeta(c(0.025, 0.975), post_a1, post_a0)
bb_pred_mean <- bb_mean                        # predictive: Bernoulli(bb_mean)
bb_pred_sd   <- sqrt(bb_pred_mean * (1 - bb_pred_mean))
```

Table 7.1: Collects every quantity reported in a conjugate Beta-Binomial analysis, from the prior and MLE to the posterior, credible interval, and predictive distribution. Complete inference output, Beta(1,1) prior with n = 10, y = 7.

quantity	value
prior mean	0.500
MLE (y/n)	0.700
log marginal likelihood	-7.185
posterior distribution	Beta(8, 4)
posterior mean	0.667
posterior sd	0.131
95% credible interval	(0.390, 0.891)
predictive distribution	Bernoulli(0.667)
predictive mean	0.667
predictive sd	0.471

With a uniform prior the posterior mean 0.667 sits between the prior mean 0.5 and the MLE 0.7, pulled almost all the way to the MLE because the prior carries only $\alpha_0 + \alpha_1 = 2$ pseudo-observations against 10 real ones. The 95% credible interval, $(0.390, 0.891)$, is read directly off the Beta CDF (`qbeta`) — no simulation, no quadrature, because conjugacy already handed over the exact posterior family. The predictive distribution Bern(0.667) has the same mean as the posterior but a nonzero sd (0.471), because even a Bernoulli outcome is uncertain even once θ is known exactly.

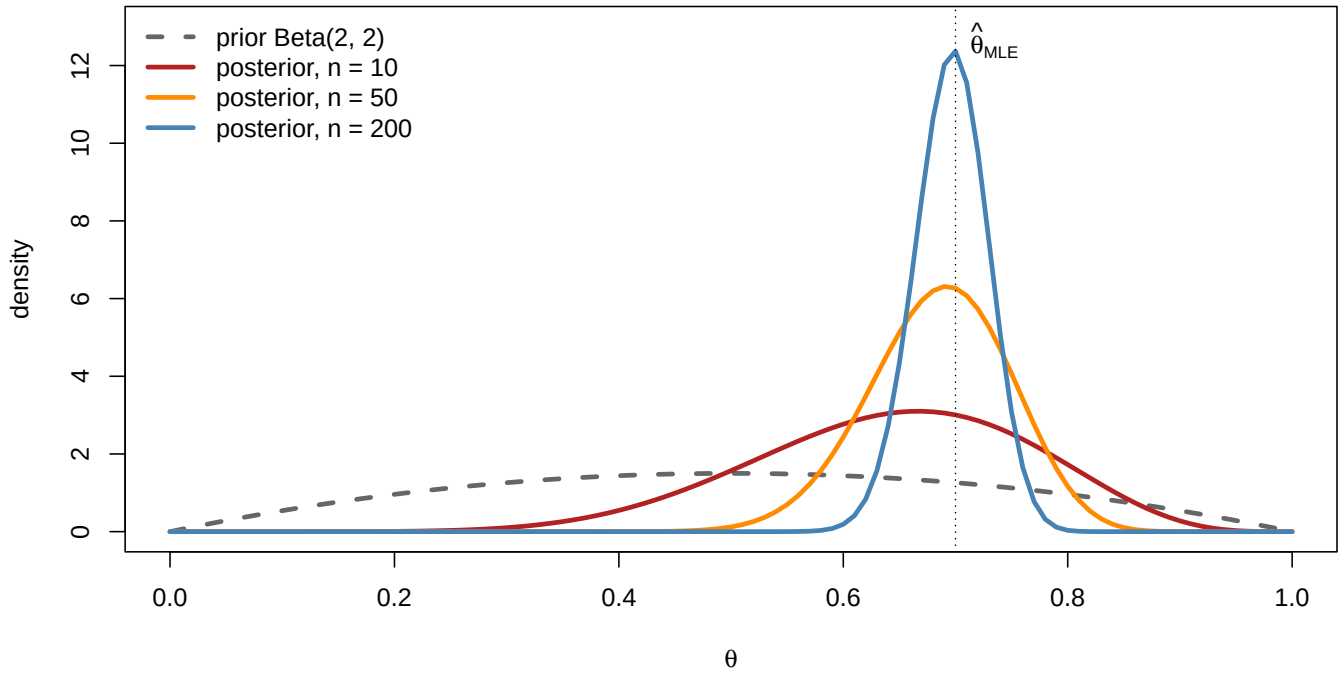


Figure 7.4: Shows how the influence of the prior fades as data accumulate, so that the posterior concentrates around the MLE. Prior Beta(2,2) (dashed) and posteriors after $n = 10, 50, 200$ observations with $\bar{y} = 0.7$ held fixed. The posterior narrows at rate $1/\sqrt{n}$ and moves onto the MLE as n grows, exactly the $w \to 0$ argument above made visible.

7.4 Example: Poisson Data with a Gamma Prior

7.4.1 Prior

Now let $y_1, \dots, y_n \mid \lambda \overset{iid}{\sim} \text{Poisson}(\lambda)$, $\lambda > 0$ a rate, with a Gamma(α, β) prior (shape α, rate β),

$$\pi(\lambda) = \frac{\beta^\alpha}{\Gamma(\alpha)} \lambda^{\alpha-1} e^{-\beta\lambda}, \qquad \lambda > 0, \qquad E(\lambda) = \frac{\alpha}{\beta}, \qquad V(\lambda) = \frac{\alpha}{\beta^2}. \tag{7.17}$$

The pseudo-data reading again applies, with the two parameters playing different roles than in the Beta case: α behaves like a prior total count and β like a prior total *exposure* (a number of pseudo time-units or pseudo-observations already seen with rate λ), so α/β is a prior average rate — the same "counts per exposure" structure as $\hat{\lambda}_{\text{MLE}} = \sum y_i / n$, with β playing the role n plays in the likelihood.

7.4.2 Likelihood and posterior

With $P(y_i \mid \lambda) = \lambda^{y_i} e^{-\lambda} / y_i!$, the likelihood depends on the data only through the total count $S = \sum_i y_i$:

$$L(\lambda) = \prod_{i=1}^{n} \frac{\lambda^{y_i} e^{-\lambda}}{y_i!} = \frac{\lambda^S e^{-n\lambda}}{\prod_i y_i!}, \qquad S = \sum_{i=1}^{n} y_i. \tag{7.18}$$

Multiplying by the prior and collecting powers of λ and $e^{-\lambda}$,

$$f(\lambda) = \pi(\lambda)L(\lambda) \;\propto\; \lambda^{\alpha-1}e^{-\beta\lambda} \cdot \lambda^{S}e^{-n\lambda} = \lambda^{(\alpha+S)-1}e^{-(\beta+n)\lambda}, \tag{7.19}$$

the kernel of a Gamma$(\alpha + S,\ \beta + n)$ density. The Gamma prior is therefore **conjugate** to the Poisson likelihood, exactly as Beta was to Bernoulli, and by the same argument (matching a recognizable kernel pins down the normalizing constant automatically):

$$\lambda \mid \mathbf{y} \sim \text{Gamma}(\alpha + S,\ \beta + n). \tag{7.20}$$

The posterior mean has precisely the weighted-average structure seen in the Bernoulli case, with β now playing the role $\alpha_0 + \alpha_1$ played there:

$$E(\lambda \mid \mathbf{y}) = \frac{\alpha + S}{\beta + n} = \underbrace{\frac{\beta}{\beta + n}}_{w} \cdot \frac{\alpha}{\beta} + (1 - w) \cdot \underbrace{\frac{S}{n}}_{\hat{\lambda}_{\text{MLE}}}, \tag{7.21}$$

a weighted average of the prior mean and the MLE, with prior weight $w = \beta/(\beta + n) \to 0$ as $n \to \infty$ — the same shrinkage story as Beta–Bernoulli, now driven by β (prior pseudo-exposure) instead of $\alpha_0 + \alpha_1$ (prior pseudo-sample-size).

7.4.3 The marginal likelihood in closed form

Because conjugacy gives the *whole* posterior in closed form, it also gives the marginal likelihood $P(\mathbf{y}) = \int f(\lambda)\, d\lambda$ in closed form, by integrating the unnormalized Gamma kernel directly against its own normalizing constant:

$$P(\mathbf{y}) = \left(\prod_{i=1}^{n} \frac{1}{y_i!}\right) \frac{\beta^{\alpha}}{\Gamma(\alpha)} \int_0^{\infty} \lambda^{(\alpha+S)-1}e^{-(\beta+n)\lambda}\, d\lambda = \left(\prod_{i=1}^{n} \frac{1}{y_i!}\right) \frac{\beta^{\alpha}\,\Gamma(\alpha + S)}{\Gamma(\alpha)\,(\beta + n)^{\alpha+S}}, \tag{7.22}$$

using $\int_0^{\infty} \lambda^{k-1}e^{-c\lambda}\, d\lambda = \Gamma(k)/c^{k}$. On the log scale,

$$\log P(\mathbf{y}) = \alpha \log \beta - \log \Gamma(\alpha) + \log \Gamma(\alpha + S) - (\alpha + S) \log(\beta + n) - \sum_{i=1}^{n} \log(y_i!). \tag{7.23}$$

This exact Gamma–Poisson conjugate identity is what makes the Poisson-rate model a useful *test case* for approximate methods: the Laplace approximation chapter later in this book applies its mode-and-curvature approximation to this same model and checks the result against precisely this formula, since here — unlike in most applications — the truth is known.

7.4.4 Posterior predictive, in closed form

For a new count $y_{n+1} \mid \lambda \sim \text{Poisson}(\lambda)$, integrating the Poisson sampling density against the Gamma$(\alpha+S, \beta+n)$ posterior gives a **Negative Binomial** predictive distribution,

$$P(y_{n+1} \mid \mathbf{y}) = \int_0^{\infty} \frac{\lambda^{y_{n+1}}e^{-\lambda}}{y_{n+1}!} \frac{(\beta + n)^{\alpha+S}}{\Gamma(\alpha + S)} \lambda^{(\alpha+S)-1}e^{-(\beta+n)\lambda}\, d\lambda = \text{NegBin}\Big(\text{size} = \alpha + S,\ \ p = \frac{\beta + n}{\beta + n + 1}\Big), \tag{7.24}$$

by the same Gamma-integral identity used for the marginal likelihood above. Its mean matches the posterior mean of λ exactly (as it must, by the tower property), but its variance is *larger* than that of a plug-in Poisson at the posterior mean rate:

$$E(y_{n+1} \mid \mathbf{y}) = \frac{\alpha + S}{\beta + n}, \qquad V(y_{n+1} \mid \mathbf{y}) = \frac{\alpha + S}{\beta + n}\left(1 + \frac{1}{\beta + n}\right) = E(y_{n+1} \mid \mathbf{y}) \cdot \left(1 + \tfrac{1}{\beta+n}\right). \tag{7.25}$$

A plug-in Poisson at $\lambda = E(\lambda \mid \mathbf{y})$ would have variance *equal* to its mean — the extra factor $1+1/(\beta+n)$ is entirely the posterior uncertainty about λ leaking into the prediction, exactly the "wider than any single $P(y_{n+1} \mid \theta_j)$" phenomenon from the general framework above, now with an explicit formula for *how much* wider.

7.4.5 Worked numbers: the complete inference output

Take Gamma$(2, 1)$ as the prior (prior mean 2, i.e. a mild belief that the rate is around 2, with pseudo-exposure $\beta = 1$) and $n = 8$ simulated counts from a true rate of 3. The table collects the same five quantities as the Beta–Bernoulli table above — marginal likelihood, posterior, its mean/sd and credible interval, and the predictive distribution:

```
alpha <- 2; beta <- 1                          # Gamma(2, 1) prior
set.seed(42)
n_pg <- 8; y_pg <- rpois(n_pg, 3)              # 8 counts from a true rate of 3
S_pg <- sum(y_pg)
post_alpha <- alpha + S_pg; post_beta <- beta + n_pg
pg_logmarg <- alpha * log(beta) - lgamma(alpha) + lgamma(post_alpha) -
  post_alpha * log(post_beta) - sum(lgamma(y_pg + 1))     # log P(y), exact
pg_mean <- post_alpha / post_beta
pg_sd   <- sqrt(post_alpha) / post_beta
pg_ci   <- qgamma(c(0.025, 0.975), post_alpha, post_beta)
pg_nb_prob   <- post_beta / (post_beta + 1)    # predictive: NegBin(post_alpha, pg_nb_prob)
pg_pred_mean <- pg_mean
pg_pred_sd   <- sqrt(pg_mean * (1 + 1 / post_beta))
```

Table 7.2: Collects every quantity reported in a conjugate Poisson-Gamma analysis, from the prior and MLE to the posterior, credible interval, and predictive distribution. Complete inference output, Gamma(2,1) prior, n = 8 Poisson counts.

quantity	value
data $y_{1:8}$	5, 6, 2, 5, 3, 3, 4, 1
sum S	29
prior mean	2.000
MLE (S/n)	3.625
log marginal likelihood	-17.065
posterior distribution	Gamma(31, 9)
posterior mean	3.444
posterior sd	0.619
95% credible interval	(2.340, 4.759)
predictive distribution	NegBin(31, 0.900)
predictive mean	3.444
predictive sd	1.956

Table 7.2: Collects every quantity reported in a conjugate Poisson-Gamma analysis, from the prior and MLE to the posterior, credible interval, and predictive distribution. Complete inference output, Gamma(2,1) prior, n = 8 Poisson counts.

quantity	value

The MLE, $29/8 = 3.625$, and the prior mean, 2, are pulled toward each other by the posterior mean 3.444 — mostly toward the MLE, since the prior's pseudo-exposure $\beta = 1$ is small next to the real $n = 8$. Compare this to the true generating rate of 3: both the MLE and the posterior mean overshoot it here (this particular simulated sample happened to run a bit high), and the 95% credible interval $(2.340, 4.759)$ still comfortably covers it. With $\beta + n = 9$, the predictive sd 1.957 exceeds the plug-in Poisson sd $\sqrt{3.444} = 1.856$ by the factor $\sqrt{1 + 1/9}$ — the same overdispersion story made numeric.

7.4.6 Overdispersion, visualized

The gap is small here because n is already large relative to the prior, but it widens as n shrinks or the prior gets vaguer (smaller β). The figure below makes the same point on the density scale: the predictive has visibly heavier tails than a plug-in Poisson with the same mean.

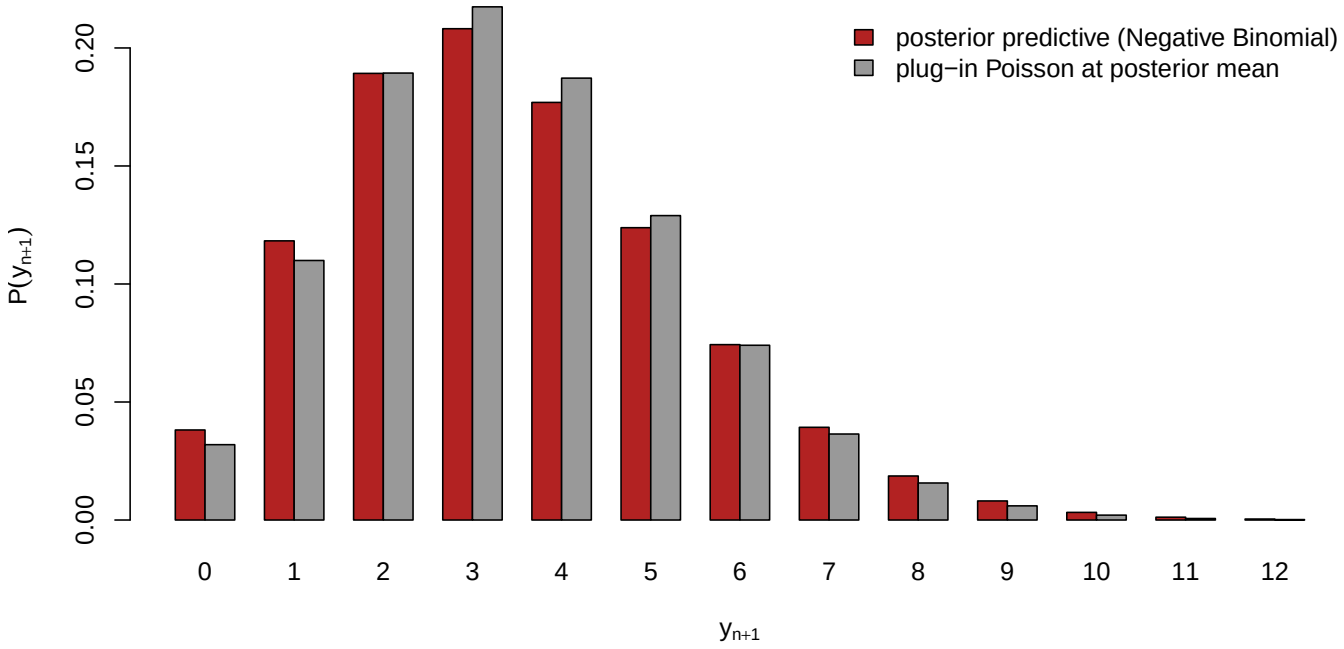


Figure 7.5: Shows why the posterior predictive distribution is more dispersed than a plug-in prediction: it also reflects the remaining uncertainty about the Poisson rate. Posterior predictive $P(y_{n+1} \mid \mathbf{y})$ (Negative Binomial, red) against a plug-in Poisson at the posterior mean rate (gray). Same mean, but the predictive has heavier tails: the two bars noticeably diverge above $y_{n+1} = 7$, and $V(y_{n+1} \mid \mathbf{y})$ exceeds the plug-in variance by the factor $1 + 1/(\beta + n) = 1.111$ derived above.

7.5 Example: Normal Data with a Normal Prior (Known Variance)

7.5.1 Prior

Now let $y_1, \dots, y_n \mid \mu \overset{iid}{\sim} N(\mu, \sigma^2)$, with σ^2 **known/fixed** and only the mean $\mu \in \mathbb{R}$ unknown. Give μ a Normal prior, parametrized through a prior pseudo-sample-size $\kappa_0 > 0$ so that it lines up with the pseudo-data reading used in the two examples above:

$$\pi(\mu) = N\Big(\mu_0,\ \frac{\sigma^2}{\kappa_0}\Big), \qquad E(\mu) = \mu_0, \qquad V(\mu) = \frac{\sigma^2}{\kappa_0}. \tag{7.26}$$

μ_0 is the prior guess for μ, and κ_0 counts how many pseudo-observations (each with the known variance σ^2) that guess is worth: large κ_0 means a confident, concentrated prior; $\kappa_0 \to 0$ recovers a flat, uninformative prior on μ.

7.5.2 Likelihood and posterior

With $P(y_i \mid \mu) = (2\pi\sigma^2)^{-1/2} \exp\{-(y_i - \mu)^2/(2\sigma^2)\}$, the likelihood depends on the data only through the sample mean $\bar{y} = \frac{1}{n}\sum_i y_i$:

$$L(\mu) \ \propto\ \exp\left\{-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \mu)^2\right\} \ \propto\ \exp\left\{-\frac{n(\mu - \bar{y})^2}{2\sigma^2}\right\}. \tag{7.27}$$

Multiplying by the prior and completing the square in μ,

$$f(\mu) = \pi(\mu)L(\mu) \ \propto\ \exp\left\{-\frac{(\mu - \mu_0)^2}{2\sigma^2/\kappa_0} - \frac{n(\mu - \bar{y})^2}{2\sigma^2}\right\} \ \propto\ \exp\left\{-\frac{(\mu - \mu_n)^2}{2\sigma^2/\kappa_n}\right\}, \tag{7.28}$$

the kernel of a Normal density with

$$\kappa_n = \kappa_0 + n, \qquad \mu_n = \frac{\kappa_0\mu_0 + n\bar{y}}{\kappa_0 + n}. \tag{7.29}$$

The Normal prior is therefore **conjugate** to the Normal likelihood (known variance), exactly as Beta was to Bernoulli and Gamma to Poisson:

$$\mu \mid \mathbf{y} \sim N\Big(\mu_n,\ \frac{\sigma^2}{\kappa_n}\Big). \tag{7.30}$$

As in the other two models, the posterior mean is a pseudo-sample-size weighted average of the prior mean and the MLE $\widehat{\mu}_{\text{MLE}} = \bar{y}$,

$$\mu_n = \underbrace{\frac{\kappa_0}{\kappa_0 + n}}_{w} \cdot \mu_0 + (1 - w) \cdot \bar{y}, \tag{7.31}$$

with prior weight $w = \kappa_0/(\kappa_0 + n) \to 0$ as $n \to \infty$, and the posterior "pseudo-sample-sizes" simply **add**: $\kappa_n = \kappa_0 + n$ — posterior precision (inverse variance, in units of $1/\sigma^2$) is prior precision plus data precision.

7.5.3 The marginal likelihood in closed form

Because conjugacy gives the *whole* posterior in closed form, it also gives the marginal likelihood $P(\mathbf{y}) = \int f(\mu)\, d\mu$ in closed form. Writing the likelihood with its full normalizing constant, $L(\mu) = (2\pi\sigma^2)^{-n/2} \exp\{-SS/(2\sigma^2)\} \exp\{-n(\mu - \bar{y})^2/(2\sigma^2)\}$ with $SS = \sum_{i=1}^{n}(y_i - \bar{y})^2$, and integrating the same Gaussian-kernel product used to find the posterior above (a standard Gaussian-product identity: $\int \exp\{-a(\mu - m_1)^2/2 - b(\mu - m_2)^2/2\}\, d\mu = \sqrt{2\pi/(a+b)}\, \exp\{-ab(m_1 - m_2)^2/(2(a+b))\}$) gives

$$P(\mathbf{y}) = (2\pi\sigma^2)^{-n/2} \sqrt{\frac{\kappa_0}{\kappa_n}}\, \exp\left\{ -\frac{SS}{2\sigma^2} - \frac{\kappa_0 n(\mu_0 - \bar{y})^2}{2\sigma^2 \kappa_n} \right\}. \tag{7.32}$$

On the log scale,

$$\log P(\mathbf{y}) = -\frac{n}{2}\log(2\pi\sigma^2) + \frac{1}{2}\log\frac{\kappa_0}{\kappa_n} - \frac{SS}{2\sigma^2} - \frac{\kappa_0 n(\mu_0 - \bar{y})^2}{2\sigma^2\kappa_n}. \tag{7.33}$$

As in the other two models, matching the Normal kernel of $f(\mu)$ against its own normalizing constant hands over $P(\mathbf{y})$ with no numerical integration.

7.5.4 Posterior predictive, in closed form

For a new $y_{n+1} \mid \mu \sim N(\mu, \sigma^2)$, integrating the Normal sampling density against the $N(\mu_n, \sigma^2/\kappa_n)$ posterior — a sum of independent Normals — gives another Normal predictive distribution,

$$y_{n+1} \mid \mathbf{y} \sim N\Big(\mu_n,\ \sigma^2\big(1 + \tfrac{1}{\kappa_n}\big)\Big), \tag{7.34}$$

with mean equal to the posterior mean of μ (as it must, by the tower property) and variance **larger** than the known sampling variance σ^2 by exactly the factor $1 + 1/\kappa_n$ — the same form as the Gamma–Poisson predictive-variance inflation above, now for a continuous outcome: the extra $1/\kappa_n$ is the leftover posterior uncertainty about μ leaking into the prediction.

7.5.5 Worked numbers: the complete inference output

Take a weakly informative prior, $\mu_0 = 0$ with $\kappa_0 = 1$ (worth one pseudo-observation), known $\sigma = 2$, and $n = 15$ observations simulated from a true mean of 3. The table collects the same five quantities as the Beta–Bernoulli and Gamma–Poisson tables above:

```
sigma_nn <- 2                                   # known/fixed sigma
mu0 <- 0; kappa0 <- 1                           # N(0, sigma^2/1) prior
set.seed(123)
n_nn <- 15; y_nn <- rnorm(n_nn, 3, sigma_nn)    # 15 draws from a true mean of 3
ybar_nn <- mean(y_nn)
SS_nn <- sum((y_nn - ybar_nn)^2)
post_kappa <- kappa0 + n_nn
post_mu  <- (kappa0 * mu0 + n_nn * ybar_nn) / post_kappa
post_var <- sigma_nn^2 / post_kappa
post_sd  <- sqrt(post_var)
```

```
nn_ci    <- qnorm(c(0.025, 0.975), post_mu, post_sd)
nn_logmarg <- -n_nn / 2 * log(2 * pi * sigma_nn^2) + 0.5 * log(kappa0 / post_kappa) -
  SS_nn / (2 * sigma_nn^2) -
  kappa0 * n_nn * (mu0 - ybar_nn)^2 / (2 * sigma_nn^2 * post_kappa)   # log P(y), exact
pred_var <- post_var + sigma_nn^2              # predictive: N(post_mu, pred_var)
pred_sd  <- sqrt(pred_var)
```

Table 7.3: Collects every quantity reported in a conjugate Normal-Normal analysis, from the prior and MLE to the posterior, credible interval, and predictive distribution. Complete inference output, $N(0, \sigma^2/1)$ prior with known $\sigma = 2$, n = 15.

quantity	value
prior mean	0.000
MLE ($\bar{y}$)	3.305
log marginal likelihood	-31.850
posterior distribution	N(3.098, 0.500^2)
posterior mean	3.098
posterior sd	0.500
95% credible interval	(2.118, 4.078)
predictive distribution	N(3.098, 2.062^2)
predictive mean	3.098
predictive sd	2.062

With only $\kappa_0 = 1$ pseudo-observation against $n = 15$ real ones, the posterior mean 3.098 sits almost on top of the MLE $\bar{y} = 3.305$, barely pulled toward the prior mean 0 — the same shrinkage story as the two examples above, now driven by κ_0 instead of $\alpha_0 + \alpha_1$ or β. The predictive sd 2.062 exceeds the known $\sigma = 2$ by the factor $\sqrt{1 + 1/\kappa_n} = \sqrt{1.062}$ derived above — the continuous analogue of the Gamma–Poisson overdispersion factor.

7.5.6 Overdispersion, visualized

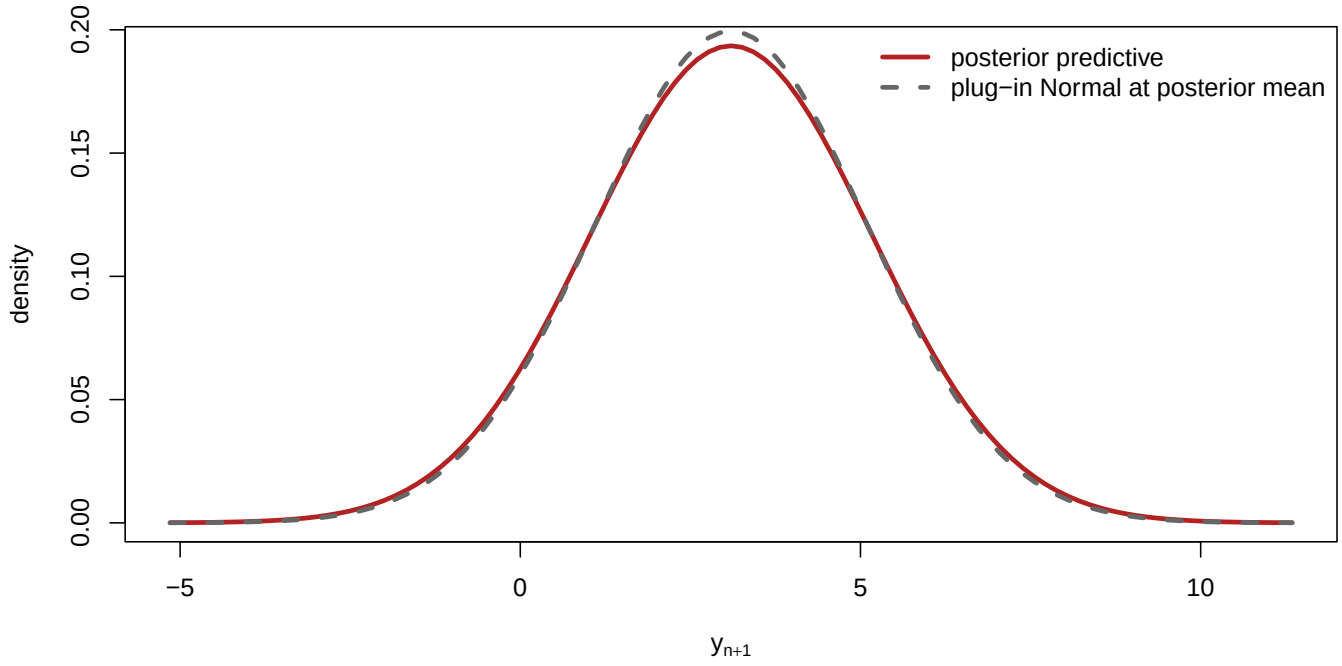


Figure 7.6: Shows why the posterior predictive distribution is wider than a plug-in prediction: it also reflects the remaining uncertainty about the mean. Posterior predictive $N(\mu_n, \sigma^2(1 + 1/\kappa_n))$ (red, solid) against a plug-in $N(\mu_n, \sigma^2)$ at the posterior mean (gray, dashed). Same mean, but the predictive is visibly wider — the extra spread is entirely the remaining posterior uncertainty about μ.

7.6 Shinylive App for Conjugate Bayesian Updating

There is a shinylive app to let you explore conjugate Bayesian updating, to see how the prior, the data, and the sample size combine into the posterior.

7.7 Where Computation Enters

For the Beta–Bernoulli, Gamma–Poisson, and Normal–Normal models, everything above is closed-form: the posterior, its mean and variance, credible intervals, and even the marginal likelihood and predictive distribution. Conjugacy is a special convenience, restricted to a handful of prior/likelihood pairings; for a general model, the same quantities require genuine numerical work:

Quantity	Requires
Normalizing constant $P(\mathbf{y}) = \int L(\theta)\pi(\theta)\, d\theta$	integration over θ
Posterior mean, variance, probabilities	integration over θ
Posterior mode	optimization
Posterior predictive $\int P(y_{n+1} \mid \theta)P(\theta \mid \mathbf{y})\, d\theta$	integration over θ

Quantity	Requires
Credible intervals	posterior quantiles

The strategy depends on the dimension of θ. For **one-dimensional** θ, a deterministic grid of evaluation points — **numerical quadrature**, the next chapter — is cheap and accurate, and **Laplace's method** (the chapter after that) offers a grid-free alternative built from just the posterior mode and its curvature, at the cost of assuming the posterior is roughly bell-shaped. For **multi-dimensional** θ, a grid becomes exponentially expensive, and the standard approach is Monte Carlo: approximate every integral by an average over posterior draws $\theta_1, \ldots, \theta_J \sim P(\theta \mid \mathbf{y})$, as prediction already did above. Producing those draws — by **rejection sampling**, **importance sampling**, and finally **Markov chain Monte Carlo** — is the subject of the remaining chapters of this book.

7.8 Summary

- Bayesian inference treats θ as random, combines a prior $\pi(\theta)$ with a likelihood $L(\theta)$ via Bayes' rule, and reports a *distribution* $P(\theta \mid \mathbf{y}) \propto f(\theta) = L(\theta)\pi(\theta)$ rather than a single point estimate.
- Every standard summary — posterior mean/median/mode, credible intervals, predictive distributions — is an integral or an optimization over that posterior, which is why Bayesian inference is a computational subject.
- The Beta–Bernoulli, Gamma–Poisson, and Normal–Normal (known variance) pairs are **conjugate**: the posterior is available in closed form (Beta, Gamma, and Normal respectively), the posterior mean is an exact weighted average of the prior mean and the MLE with the prior's pseudo-sample-size as weight, and the marginal likelihood and predictive distribution are closed-form too (Negative Binomial for Gamma–Poisson, Normal with inflated variance for Normal–Normal) — no integration needed anywhere.
- The Gamma–Poisson predictive distribution is visibly overdispersed relative to a plug-in Poisson at the posterior mean rate, by an exact factor $1 + 1/(\beta + n)$: real posterior uncertainty about θ widens predictions, a general feature the closed form makes precise here.
- Conjugacy is the exception, not the rule. The rest of this book develops general-purpose tools for the same four computations — quadrature, Laplace approximation, rejection sampling, importance sampling, and MCMC — for models where no such algebraic shortcut exists.

8 Numerical Quadrature

```
library(metRology)
```

The previous chapter laid out the Bayesian framework: a prior $\pi(\theta)$, a likelihood $L(\theta)$, and a posterior $P(\theta \mid \mathbf{y}) \propto f(\theta) = L(\theta)\pi(\theta)$ known only up to its normalizing constant, the marginal likelihood $P(\mathbf{y}) = \int f(\theta)\, d\theta$. Outside the conjugate examples there, every posterior summary — that normalizing constant, a posterior mean, a credible interval, a predictive probability — is an *area* under f with no closed form: a tail probability, a posterior expectation, or a normalizing constant. (The two chapters before that found the *mode* of a log-likelihood or log-posterior by optimization instead — useful, but a different problem.) This chapter develops **numerical quadrature** — approximating an integral by a weighted sum of function values at a finite set of points — and applies it to the central example of the rest of the book, the marginal likelihood of a Bayesian model. The next chapter (Laplace's method) revisits the same example with a method based on the mode instead of a grid.

8.1 The Problem and Transformations

8.1.1 The general setup

The goal is to approximate

$$A = \int_a^b f(x)\, dx \tag{8.1}$$

for finite a, b by evaluating f at n points $x_1, \ldots, x_n$ and forming a weighted sum

$$\tilde{A}_n = \sum_{i=1}^{n} w_i\, f(x_i). \tag{8.2}$$

The quadrature rules below differ only in how the points and weights are chosen. Typical statistical uses are probabilities $P(a < X < b)$, expectations $E[h(X)]$, and normalizing constants such as the marginal likelihood $P(y) = \int L(\theta)\pi(\theta)\, d\theta$ in Bayesian inference.

8.1.2 Infinite ranges: the logistic transformation

A grid needs a finite interval, so when the natural range of integration is $(-\infty, \infty)$ we first change variables. The logistic function maps $x \in (-\infty, \infty)$ to $y \in (0, 1)$,

$$y = \frac{1}{1 + e^{-x}}, \qquad x = t(y) = \log \frac{y}{1 - y}, \qquad t'(y) = \frac{1}{y(1 - y)}, \tag{8.3}$$

and the standard change-of-variables identity gives

$$\int_{-\infty}^{\infty} f(x)\,dx = \int_0^1 f(t(y))\,t'(y)\,dy = \int_0^1 g(y)\,dy, \qquad g(y) = \frac{f(\log \frac{y}{1-y})}{y(1-y)}. \tag{8.4}$$

Any quadrature rule for a finite interval now applies to g on $(0,1)$. Two consequences matter later in the chapter:

- Points equally spaced in y correspond to points in x that are dense near $x = 0$ and increasingly sparse in the tails — the grid automatically concentrates effort where the logistic curve is steep.
- The factor $1/\{y(1-y)\}$ blows up at $y = 0$ and $y = 1$, so g is typically undefined at the endpoints even when f is perfectly well behaved. This is the reason the midpoint rule, which never evaluates the integrand at the endpoints of a panel, is the natural choice here.

For a half-line $(0,\infty)$ the same idea applies with $x = -\log(1-y)$ or $x = y/(1-y)$ in place of the logistic map.

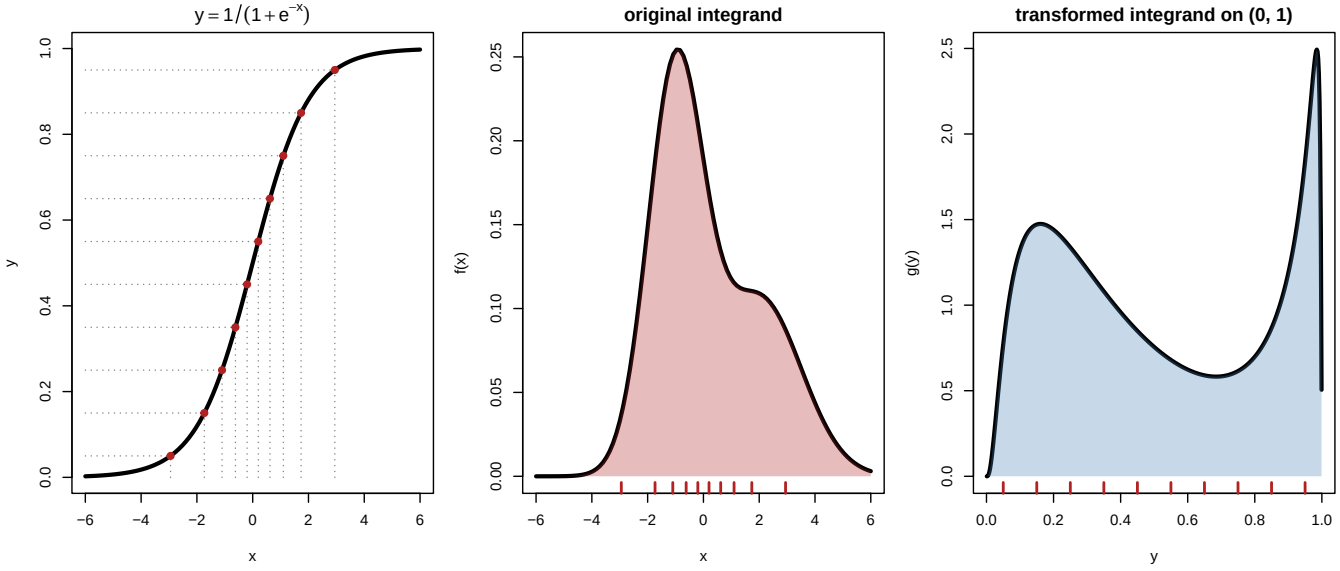


Figure 8.1: Shows how a change of variable maps an integral over the whole real line onto a finite interval, so that a rule with equally spaced points can be applied. Equally spaced points in y (red ticks, right panel) map to unequally spaced points in x (red ticks, middle panel). The two shaded areas are equal: the transform changes where we look, not the value of the integral.

8.1.3 Probabilities as a special case

For X with density φ, e.g. $\varphi(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$,

$$P(a < X < b) = \int_a^b \varphi(x)\,dx = \int_{-\infty}^{\infty} \varphi^{(c)}(x)\,dx, \qquad \varphi^{(c)}(x) = \varphi(x)\,I(a < x < b). \tag{8.5}$$

Either integrate φ directly over the finite interval $[a,b]$, or integrate the *truncated* function $\varphi^{(c)}$ over the whole line, letting the indicator do the cutting. The second view is the more useful one: the same machinery — transform to $(0,1)$, apply a quadrature rule — computes $P(X \in S)$ for an arbitrary set S and, more generally, $E[h(X)] = \int h(x)\varphi(x)\,dx$ for an arbitrary h.

8.2 Quadrature Rules on a Finite Interval

All three rules below partition $[a, b]$ into n panels of width $h_n = (b-a)/n$ and differ only in how f is approximated *within* each panel.

8.2.1 Midpoint rule

Approximate f on each panel by its value at the panel's midpoint (a constant, i.e. a rectangle):

$$\tilde{A}_n = h_n \sum_{i=1}^{n} f\Big(a + (i - \tfrac{1}{2})h_n\Big). \tag{8.6}$$

For smooth f the error is $A - \tilde{A}_n = \dfrac{(b-a)h_n^2}{24} f''(\xi) = O(h_n^2)$ for some $\xi \in (a, b)$. The rule never evaluates f at the panel edges, so in particular it never touches a or b — convenient when f is singular or undefined at the endpoints, exactly the situation created by the logistic transform above.

8.2.2 Trapezoidal rule

Approximate f on $[x_{i-1}, x_i]$ by the straight line through the two endpoint values, so each panel contributes a trapezoid, $\frac{1}{2}[f(x_{i-1}) + f(x_i)](x_i - x_{i-1})$. Summing over panels, interior points are shared by two neighbouring panels and each contributes once (not twice) to the total:

$$\tilde{A}_n = \tfrac{1}{2}h_n\big[f(a) + f(b)\big] + h_n \sum_{i=1}^{n-1} f(a + ih_n). \tag{8.7}$$

The error is $A - \tilde{A}_n = -\dfrac{(b-a)h_n^2}{12} f''(\xi) = O(h_n^2)$ — the same order as the midpoint rule, but with twice the constant and the opposite sign. It is exact for linear f.

8.2.3 Simpson's rule

Take $n = 2m$ even. On each *pair* of panels $[x_0, x_1]$ with $x_1 - x_0 = 2h_n$, replace f by the quadratic through the two endpoints and the midpoint; integrating that quadratic exactly gives

$$\int_{x_0}^{x_1} f(x)\,dx \approx \frac{x_1 - x_0}{6}\Big[f(x_0) + 4f(\tfrac{x_0+x_1}{2}) + f(x_1)\Big]. \tag{8.8}$$

Summing over the m panel-pairs, with each even interior point shared by two pairs,

$$\tilde{A}_{2m} = \frac{h_n}{3}\left[f(a) + f(b) + 4\sum_{i=1}^{m} f(a + (2i-1)h_n) + 2\sum_{i=1}^{m-1} f(a + 2ih_n)\right]. \tag{8.9}$$

The error is $A - \tilde{A}_{2m} = -\dfrac{(b-a)h_n^4}{180} f^{(4)}(\xi) = O(h_n^4)$, two orders faster than the other two rules, and Simpson's rule is exact for cubics. It can also be written as a weighted average of the other two rules on the same panel pair: Simpson $= \frac{2}{3}$ midpoint $+ \frac{1}{3}$ trapezoid.

Table 8.1 summarizes the three rules.

Table 8.1: Summarizes how the midpoint, trapezoidal, and Simpson rules differ in the way they approximate the integrand and in their accuracy. The three quadrature rules compared

Rule	Approximates f on a panel by	Uses endpoints?	Error	Exact for
Midpoint	constant at panel midpoint	no	$O(h_n^2)$	linear
Trapezoidal	line through panel endpoints	yes	$O(h_n^2)$	linear
Simpson	quadratic through 3 points (2 panels)	yes	$O(h_n^4)$	cubic

8.2.4 Implementation and accuracy

All three rules are one line of vectorized R each — no explicit loop over panels is needed:

```
midpoint  <- function(f, a, b, n) {
    h <- (b - a) / n
    h * sum(f(a + (seq_len(n) - 0.5) * h))
}
trapezoid <- function(f, a, b, n) {
    h <- (b - a) / n
    x <- a + (0:n) * h
    h * (sum(f(x)) - 0.5 * (f(a) + f(b)))
}
simpson   <- function(f, a, b, n) {
    stopifnot(n %% 2 == 0)
    h <- (b - a) / n
    x <- a + (0:n) * h
    h / 3 * sum(f(x) * c(1, rep(c(4, 2), n / 2 - 1), 4, 1))
}
```

We check all three against a case with a known answer, $P(0.5 < Z < 2)$ for $Z \sim N(0, 1)$:

```
exact <- pnorm(2) - pnorm(0.5)
errs <- sapply(list(midpoint = midpoint, trapezoid = trapezoid, simpson = simpson),
               function(rule) rule(dnorm, 0.5, 2, 10) - exact)
knitr::kable(t(errs), digits = 10, row.names = FALSE)
```

Table 8.2: Compares the accuracy of the midpoint, trapezoidal, and Simpson rules on the same integral with the same small number of panels. Error of each rule with only $n = 10$ panels, relative to `pnorm`.

midpoint	trapezoid	simpson
-6.4163e-05	0.0001280126	-1.6895e-06

With only 10 panels, Simpson's rule is already accurate to about 10^{-7}, several orders of magnitude better than the other two.

```
ns  <- 2^(1:9)
err <- sapply(list(midpoint, trapezoid, simpson),
              function(rule) sapply(ns, function(n) abs(rule(dnorm, 0.5, 2, n) - exact)))
matplot(ns, err, type = "b", log = "xy", pch = 19, lty = 1, lwd = 2,
        col = c("steelblue", "firebrick", "forestgreen"),
        xlab = "number of panels n", ylab = "absolute error")
legend("bottomleft", bty = "n", lwd = 2, pch = 19,
       col = c("steelblue", "firebrick", "forestgreen"),
       legend = c("midpoint, slope -2", "trapezoid, slope -2", "Simpson, slope -4"))
```

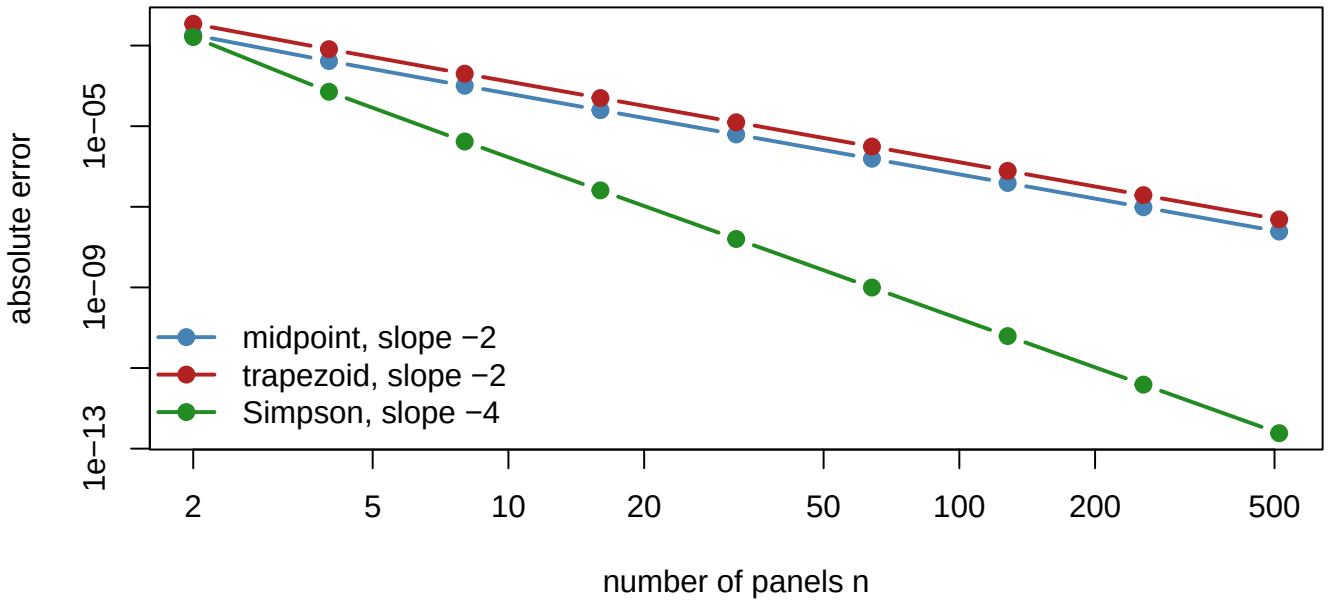


Figure 8.2: Shows how the error of each quadrature rule shrinks as the number of panels grows, which reveals the order of the rule. Absolute error versus number of panels, log-log scale. The slopes are the exponents of $1/n$: -2 for midpoint and trapezoid, -4 for Simpson, until floating-point rounding error dominates near 10^{-15}.

For a smooth, unimodal integrand like a normal density, Simpson's rule dominates. The rest of this chapter nonetheless uses the **midpoint rule**, because the application of interest — Bayesian marginal likelihood after a logistic transform — produces an integrand that is undefined exactly at the endpoints 0 and 1, which only the midpoint rule avoids evaluating.

8.3 Application: The Marginal Likelihood

8.3.1 Setup

With likelihood $L(\theta) = \prod_{i=1}^{n} P(y_i \mid \theta)$ and prior $\pi(\theta)$, Bayes' rule is

$$P(\theta \mid y) = \frac{L(\theta)\pi(\theta)}{P(y)}, \qquad P(y) = \int L(\theta)\pi(\theta)\, d\theta = E_{\pi}[L(\theta)]. \tag{8.10}$$

$P(y)$, the **marginal likelihood** (the likelihood averaged over the prior), is the normalizing constant of the posterior and the basis of Bayes factors for comparing models. It has no closed form outside conjugate settings, so we approximate it by quadrature. Because it depends on the *whole* prior — not just where the likelihood happens to be large — its value is far more sensitive to prior choice than a posterior mean is, as the next figure shows.

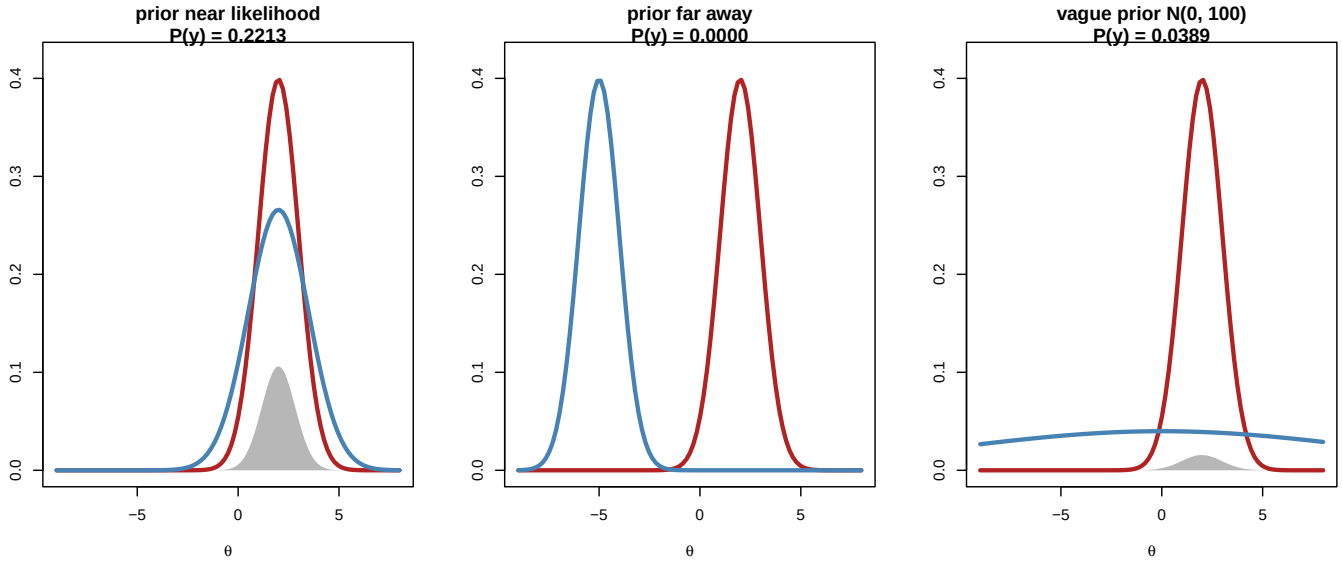


Figure 8.3: Illustrates that the marginal likelihood $P(y)$ is the integral of the likelihood times the prior, and that it changes with the prior even when the likelihood is fixed. The same likelihood (red) combined with three priors (blue). $P(y)$ is the area under $L(\theta)\pi(\theta)$ (shaded), computed here by `integrate` for a single parameter as a sanity check.

A prior centred where the likelihood is large gives a much bigger marginal likelihood than one that is far away or merely diffuse — which is exactly why $P(y)$ is useful for comparing models, and why we must be able to compute it even when the parameter is multivariate and there is no `integrate`.

8.3.2 A two-parameter example: mean and variance unknown

Let $y_1, \dots, y_n \mid \mu, \sigma \overset{iid}{\sim} t_\nu(\mu, \sigma)$, a location-scale t distribution with ν degrees of freedom (the special case $\nu = \infty$ is the normal). Reparametrize the scale as $w = \log \sigma$ so that both parameters range over $(-\infty, \infty)$, and put independent normal priors on them:

$$\mu \sim N(\mu_0, \sigma_\mu^2), \qquad w \sim N(w_0, \sigma_w^2). \tag{8.11}$$

The marginal likelihood is the two-dimensional integral

$$P(y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} L(\mu, w)\,\pi(\mu, w)\,d\mu\,dw, \qquad L(\mu, w) = \prod_{i=1}^{n} t_\nu\!\left(y_i;\ \mu,\ e^{w}\right), \tag{8.12}$$

which has no closed form because the normal prior on w is not conjugate to the likelihood. We handle it with a **product midpoint rule**: apply the logistic transform to each of μ and w separately,

$$\mu = \log\frac{u}{1-u}, \qquad w = \log\frac{v}{1-v}, \qquad d\mu\,dw = \frac{du\,dv}{u(1-u)\,v(1-v)}, \tag{8.13}$$

giving

$$P(y) = \int_0^1\int_0^1 g(u, v)\,du\,dv, \qquad g(u, v) = \frac{L(\mu(u), w(v))\,\pi(\mu(u), w(v))}{u(1-u)\,v(1-v)}, \tag{8.14}$$

and evaluate g on an $n \times n$ grid $u_j = (j - \tfrac{1}{2})h$, $v_k = (k - \tfrac{1}{2})h$ with $h = 1/n$:

$$P(y) \approx h^2 \sum_{j=1}^{n}\sum_{k=1}^{n} g(u_j, v_k). \tag{8.15}$$

The midpoint rule again earns its place here: $1/\{u(1-u)v(1-v)\} \to \infty$ at the boundary of the square, and the midpoint grid never touches it.

8.3.3 Computing on the log scale

$L(\theta)$ is a product of n densities and underflows to exactly 0 in floating point for even moderate n, so every calculation below is carried out on the log scale, and sums of exponentials are combined with **log-sum-exp**:

$$\log\sum_i e^{\ell_i} = M + \log\sum_i e^{\ell_i - M}, \qquad M = \max_i \ell_i. \tag{8.16}$$

Subtracting the maximum before exponentiating makes the largest term exactly $e^0 = 1$, so nothing overflows; terms far below M underflow harmlessly to zero instead of corrupting the sum.

```
log_sum_exp <- function(lx) {
    m <- max(lx)
    m + log(sum(exp(lx - m)))
}
```

8.3.4 An efficient implementation

The functions below compute $\log g(u, v)$ on the whole grid at once with `outer()`, rather than looping in R or nesting two calls to a generic 1-D integrator (the nested approach used in earlier drafts of this material is correct but re-evaluates the logistic transform and re-dispatches a function call n^2 times through two layers of `sapply`; building the grid directly is both clearer and several times faster for the grid sizes used here).

```
### logistic link and its log-derivative, both vectorized
logit_inv     <- function(u) log(u / (1 - u))
log_der_logit <- function(u) -log(u) - log(1 - u)

### log likelihood of a location-scale t sample (df = Inf is the normal)
log_lik <- function(x, mu, w, df = Inf) sum(dt.scaled(x, df, mu, exp(w), log = TRUE))

### log joint prior on (mu, w)
log_prior <- function(mu, w, mu_0, sigma_mu, w_0, sigma_w) {
    dnorm(mu, mu_0, sigma_mu, log = TRUE) + dnorm(w, w_0, sigma_w, log = TRUE)
}

### log of the transformed integrand g(u, v) at one grid point
log_g <- function(u, v, x, mu_0, sigma_mu, w_0, sigma_w, df = Inf) {
    mu <- logit_inv(u); w <- logit_inv(v)
    log_lik(x, mu, w, df) + log_prior(mu, w, mu_0, sigma_mu, w_0, sigma_w) +
        log_der_logit(u) + log_der_logit(v)
}

### log marginal likelihood by the product midpoint rule on an n x n grid
log_marlik_mid <- function(x, mu_0, sigma_mu, w_0, sigma_w, n, df = Inf) {
    h <- 1 / n
    grid <- (seq_len(n) - 0.5) * h
    G <- outer(grid, grid, Vectorize(function(u, v)
        log_g(u, v, x, mu_0, sigma_mu, w_0, sigma_w, df)))
    log_sum_exp(G) + 2 * log(h)
}

### log marginal likelihood by naive Monte Carlo (for validating the above)
log_marlik_mc <- function(x, mu_0, sigma_mu, w_0, sigma_w, iters_mc, df = Inf) {
    mus <- rnorm(iters_mc, mu_0, sigma_mu)
    ws  <- rnorm(iters_mc, w_0, sigma_w)
    v_log_lik <- vapply(seq_len(iters_mc), function(i) log_lik(x, mus[i], ws[i], df),
↪   numeric(1))
    log_sum_exp(v_log_lik) - log(iters_mc)
}
```

`log_marlik_mc` draws θ from the prior and averages the likelihood — a direct Monte Carlo estimate of $E_\pi[L(\theta)]$ — and serves purely as an independent check on the quadrature answer, since it makes no use of a grid or a transformation and so cannot share their failure modes.

8.3.5 Visualizing the grid and the integrand

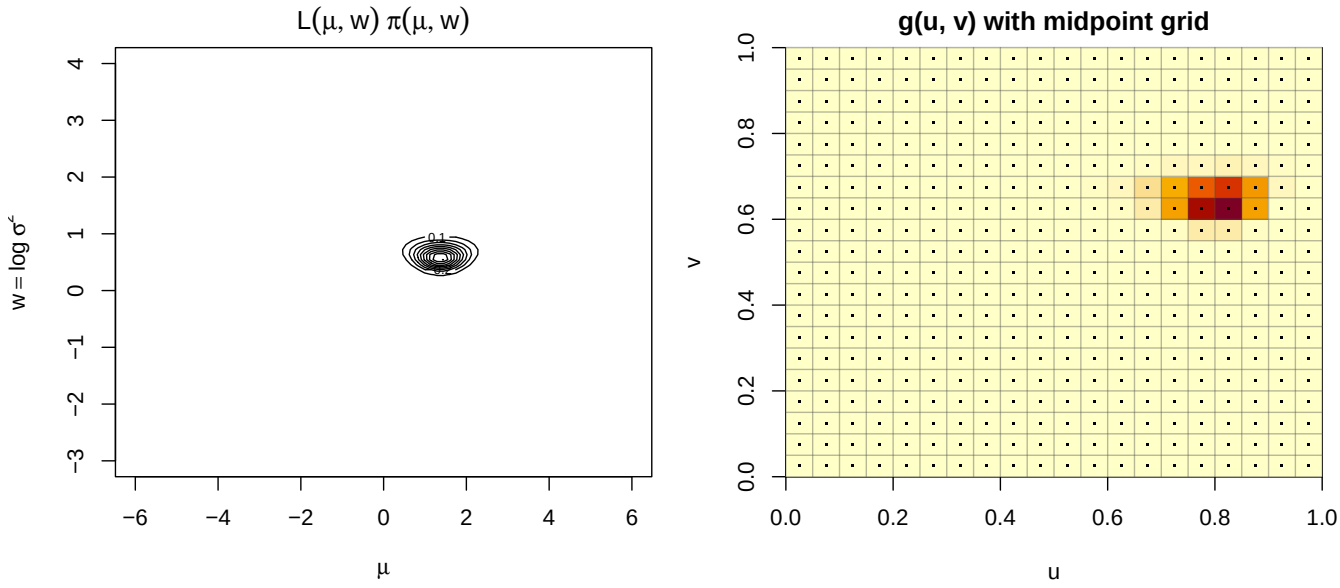


Figure 8.4: Illustrates how a two-dimensional posterior integral is computed by transforming it to the unit square and applying a product midpoint grid. Left: the posterior $L(\mu, w)\pi(\mu, w)$ on its natural scale. Right: the transformed integrand $g(u, v)$ on the unit square, with the midpoint grid at $n = 20$ overlaid. Equal probability mass under the two surfaces corresponds to equal volume, even though the shapes differ.

There is a shinylive app that draws both surfaces as contour heatmaps for normal data with unknown μ and σ, shows the midpoint grid on each, and compares the midpoint estimate of $\log P(y)$ with a reference value as the data location, sample size, grid size and priors change.

8.3.6 Checking the quadrature against Monte Carlo

```
set.seed(2)
x1 <- rt.scaled(50, mean = 2, sd = 2, df = 2)
x2 <- rt.scaled(100, mean = 2, sd = 2, df = Inf)

comparison <- data.frame(
    dataset = c("n=50, t (df=2) data", "n=100, normal data"),
    midpoint_n100 = c(log_marlik_mid(x1, 0, 10, 0, 10, 100),
                      log_marlik_mid(x2, 0, 10, 0, 10, 100)),
    monte_carlo   = c(log_marlik_mc(x1, 0, 10, 0, 10, 1e5),
                      log_marlik_mc(x2, 0, 10, 0, 10, 1e5))
)
knitr::kable(comparison, digits = 4)
```

Table 8.3: Compares quadrature with naive Monte Carlo for computing the log marginal likelihood, to show that a deterministic grid is much more accurate in low dimensions. Estimated log marginal likelihood: product midpoint rule versus naive Monte Carlo, for two simulated data sets.

dataset	midpoint_n100	monte_carlo
n=50, t (df=2) data	-173.6227	-173.5375
n=100, normal data	-221.7068	-221.6300

The two independent methods agree to well within a tenth of a log-unit, which is reassuring: they share no code path, so agreement is evidence that neither the grid nor the Monte Carlo sample is too coarse for this posterior.

8.3.7 Convergence as the grid is refined

```
ns_grid <- seq(10, 90, by = 10)
est <- vapply(ns_grid, function(k) log_marlik_mid(x2, 0, 10, 0, 10, k), numeric(1))
knitr::kable(data.frame(`grid points per axis` = ns_grid, `log P(y)` = est,
                        check.names = FALSE), digits = 4)
```

Table 8.4: Shows how the quadrature estimate converges as the grid is refined, which indicates how many points are needed. Estimated log marginal likelihood as the number of grid points per axis increases, for the $n = 100$ normal-data example. By $n = 40$ per axis the estimate has stabilized to the precision shown.

grid points per axis	log P(y)
10	-222.7297
20	-221.5545
30	-221.7318
40	-221.7069
50	-221.7067
60	-221.7068
70	-221.7068
80	-221.7068
90	-221.7068

8.3.8 Comparing priors

Holding the data fixed, the marginal likelihood falls as the prior on μ either narrows around the wrong value or spreads out indefinitely:

```
set.seed(3)
x <- rnorm(100)
mu0_vals   <- c(0, -5)
```

```
sigma_vals <- c(0.01, 0.1, 1, 10, 100, 1000)
log_P_y <- outer(mu0_vals, sigma_vals,
                 Vectorize(function(m, s) log_marlik_mid(x, m, s, 0, 1, 100)))
rownames(log_P_y) <- mu0_vals
colnames(log_P_y) <- sigma_vals
knitr::kable(log_P_y, digits = 2, align = rep("r", length(sigma_vals)))
```

Table 8.5: Shows how the log marginal likelihood, the quantity used for Bayesian model comparison, changes with the prior mean and prior standard deviation. Log marginal likelihood for data centred at 0: rows are the prior mean μ_0, columns the prior sd σ_μ (w-prior fixed at N(0,1)).

	0.01	0.1	1	10	100	1000
0	-129.39	-128.95	-130.97	-133.27	-135.57	-137.88
-5	-739.84	-316.30	-143.43	-133.40	-135.57	-137.88

With $\mu_0 = 0$ (the right neighbourhood for data centred at 0), the marginal likelihood is highest for a moderately informative prior and degrades slowly as the prior widens. With $\mu_0 = -5$ (the wrong neighbourhood), a *narrow* prior is actively harmful — it insists on a region the data reject — while a wide prior recovers because it eventually puts enough mass back near the truth. This is the general shape of Bayesian model comparison: strong, correct priors win; strong, wrong priors lose; vague priors are safe but not optimal.

8.3.9 Comparing likelihoods

The marginal likelihood also compares different likelihoods for the same data — here, whether ν (the degrees of freedom of the t model) matches how the data were actually generated:

```
dfs <- c(Inf, 2, 1, 0.5)
set.seed(4)
x_normal <- rt.scaled(100, mean = 2, sd = 2, df = Inf)
x_heavy  <- rt.scaled(100, mean = 2, sd = 2, df = 2)
model_compare <- data.frame(
    df_fitted        = dfs,
    normal_data      = vapply(dfs, function(d) log_marlik_mid(x_normal, 0, 10, 0, 10, 100, df
     ↪  = d), numeric(1)),
    heavy_tailed_data = vapply(dfs, function(d) log_marlik_mid(x_heavy,  0, 10, 0, 10, 100, df
     ↪  = d), numeric(1))
)
knitr::kable(model_compare, digits = 3,
             col.names = c("df of fitted model", "data simulated as normal", "data simulated as
              ↪  t(df=2)"))
```

Table 8.6: Shows how the marginal likelihood can be used to compare models, here t models with different degrees of freedom fitted to Gaussian and to heavy-tailed data. Log marginal likelihood under four values of the t-model degrees of freedom nu, fit to data simulated from a Gaussian and from a heavy-tailed t(df=2).

df of fitted model	data simulated as normal	data simulated as t(df=2)
Inf	-210.675	-300.951
2.0	-220.056	-261.663
1.0	-231.060	-265.646
0.5	-251.042	-281.555

Each column favours the ν that actually generated it: normal data are best explained by $\nu = \infty$, and $t(\mathrm{df} = 2)$ data are best explained by $\nu = 2$, with the wrong tail weight costing several log-likelihood units either way. As the earlier prior comparison showed, the prior also shifts these numbers — but the *ranking* of the correct model over the incorrect ones survives even under a badly misspecified prior on (μ, w), which is why marginal likelihoods remain useful for model comparison despite their prior-sensitivity.

8.4 Difficulties of Numerical Quadrature

8.4.1 When the grid misses the posterior's mode

The likelihood $L(\mu) = N(\hat{\mu};\, \mu,\, 1/n_{\text{eff}})$ treats $\hat{\mu}$ — the box you can type into — as the sample mean of n_{eff} observations from $N(\mu, 1)$; the prior is $\pi(\mu) = N(\mu; 0, 5^2)$. The marginal likelihood is

$$P(y) = \int_{-\infty}^{\infty} z(\mu)\, d\mu, \qquad z(\mu) = \pi(\mu) L(\mu). \tag{8.17}$$

Because this is a normal-normal conjugate pair, $P(y)$ has a closed form — but the app instead gets it by numerical integration, shown as the "true" value in the legend. To approximate $P(y)$ with a finite quadrature rule, apply the same logistic transform used throughout this chapter, $u = 1/(1 + e^{-\mu})$, so that $d\mu = du/\{u(1-u)\}$ and

$$P(y) = \int_0^1 \frac{z(\mathrm{logit}(u))}{u(1-u)}\, du. \tag{8.18}$$

The midpoint rule with n points $u_i = (i - \frac{1}{2})/n$ turns this into a sum of value-times-Jacobian terms,

$$P(y) \approx \frac{1}{n} \sum_{i=1}^{n} \frac{z(\mu_i)}{u_i(1-u_i)}, \qquad \mu_i = \mathrm{logit}(u_i), \tag{8.19}$$

and that sum is exactly what the picture shows: drag the slider to change n, and each grey point is one term of it — placed at μ_i by the guide lines through the logistic curve, then dropped onto $z(\mu)$ to reveal the $z(\mu_i)$ factor it contributes, before being scaled by the Jacobian $1/\{u_i(1-u_i)\}$. Move μ away from 0 and every term lands where $z(\mu) \approx 0$, however many points you add — which is why the quadrature estimate in the legend drifts far from the true value. This app runs live only in the HTML edition of this book; readers of other formats can follow the link below it.

8.4.1.1 Shinylive App for a Quadrature Grid Missing the Posterior Mode

There is a shinylive app to show how a fixed quadrature grid can miss the posterior mode entirely, so that the computed marginal likelihood is badly wrong. The same app also shows the Laplace-normalized grid of the next unit, which fixes the problem.

8.4.2 The same failure in the full two-parameter model

The app fixes σ to isolate the mechanism, but the product midpoint rule over (μ, w) introduced earlier in this chapter — `log_marlik_mid` — fails the same way once both the mean and the variance are unknown, and `log_marlik_mc` again serves as the independent check. Table 8.7 compares the two for data centred at $\mu = -5$ and at $\mu = -20$ (both with $\sigma = 2$, $n = 100$), against a prior on μ centred at 0 with $\sigma_\mu = 10$ — reasonably vague, but not recentred on either dataset.

```
set.seed(6)
x_m5  <- rnorm(100, -5, 2)
x_m20 <- rnorm(100, -20, 2)

mode_far_tbl <- data.frame(
    data        = c("mu = -5", "mu = -20"),
    midpoint    = c(log_marlik_mid(x_m5,  0, 10, 0, 10, 100),
                    log_marlik_mid(x_m20, 0, 10, 0, 10, 100)),
    monte_carlo = c(log_marlik_mc(x_m5,  0, 10, 0, 10, 1e5),
                    log_marlik_mc(x_m20, 0, 10, 0, 10, 1e5))
)
knitr::kable(mode_far_tbl, digits = 3,
             col.names = c("data", "midpoint (n = 100/axis)", "Monte Carlo (N = 1e5)"))
```

Table 8.7: Shows a failure mode of grid quadrature and naive Monte Carlo: when the posterior mass lies far from where the grid or the prior draws are placed, both estimates are poor. Log marginal likelihood for data centred at mu = -5 and mu = -20 (sigma = 2, n = 100), prior mu_0 = 0, sigma_mu = 10, w_0 = 0, sigma_w = 10: the product midpoint rule (n = 100 points per axis) against a naive Monte Carlo run of 100,000 draws from the prior.

data	midpoint (n = 100/axis)	Monte Carlo (N = 1e5)
mu = -5	-222.488	-222.673
mu = -20	-420.335	-213.503

At $\mu = -5$ the two methods should still be close, since the logistic grid still has some points in the region the posterior occupies; at $\mu = -20$ the posterior for μ has been pushed close enough to the boundary of the transformed square that the midpoint estimate can disagree with the Monte Carlo answer by a large margin on the log scale — the two-dimensional analogue of exactly what the app above shows in one dimension.

8.4.3 The curse of dimensionality

A product grid with n points per axis needs n^d evaluations in d dimensions:

Table 8.8: Quantifies the curse of dimensionality: the number of grid points needed by a product rule grows exponentially with the dimension. Grid size grows exponentially with dimension

d	$n = 20$	$n = 50$
1	20	50
2	400	2,500
5	3.2×10^6	3.1×10^8
10	1.0×10^{13}	9.8×10^{16}

Meanwhile the posterior occupies only a tiny, often tilted, region inside that grid — most evaluations are wasted on points where $L(\theta)\pi(\theta) \approx 0$ — and the error of a product rule, expressed in terms of the total number of evaluations $N = n^d$, degrades to $O(N^{-2/d})$.

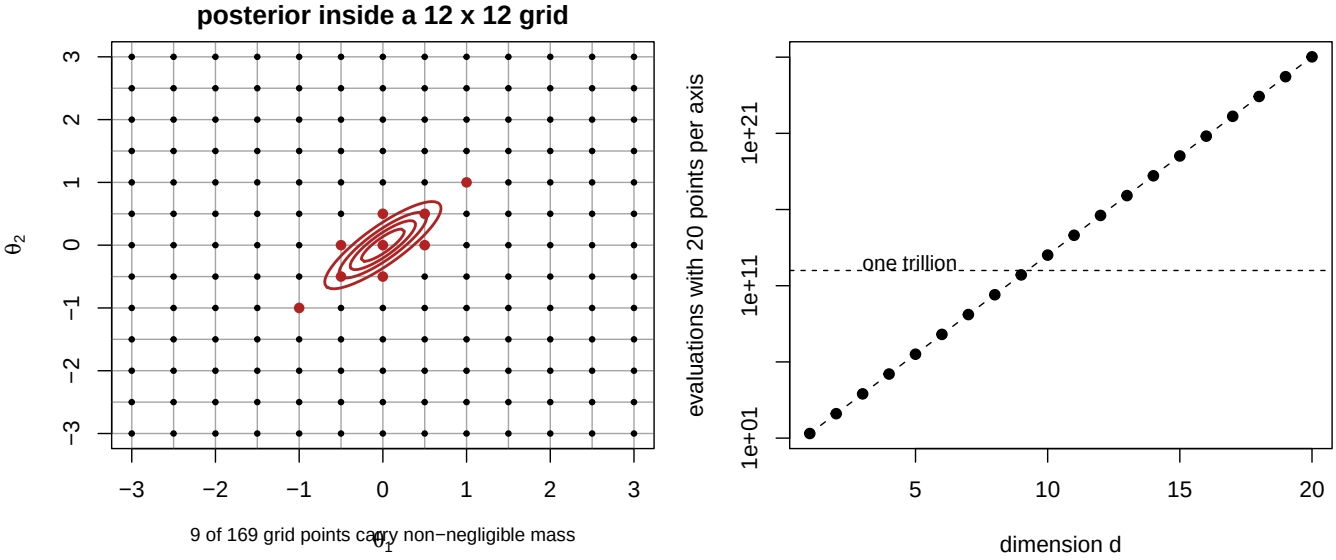


Figure 8.5: Illustrates the curse of dimensionality for quadrature: a grid wastes most of its points on negligible regions, and the number of evaluations explodes with the dimension. Left: a correlated 2-D posterior inside a 12x12 grid; only the highlighted points carry non-negligible mass. Right: evaluations needed for a 20-point-per-axis grid as dimension increases.

Conclusion. Quadrature is a reliable, cheap tool for $d \lesssim 3$, and it is what the rest of this book uses whenever a low-dimensional integral needs checking against another method. Beyond that, Monte Carlo methods — whose error is $O(N^{-1/2})$ *regardless of dimension* — take over, at the cost of losing the very fast convergence rates of Simpson's rule; this trade-off motivates the sampling methods developed from the next few chapters onward.

8.5 Summary

- Quadrature approximates $\int_a^b f\,dx$ by a weighted sum of f at grid points; infinite ranges are handled by a change of variable such as the logistic transform, always carrying along the Jacobian $t'(y)$.
- The midpoint and trapezoidal rules have $O(h_n^2)$ error; Simpson's rule has $O(h_n^4)$. The midpoint rule's extra virtue — never evaluating the integrand at a panel edge — matters whenever the transformed integrand is singular at the boundary, as it is here.
- The marginal likelihood $P(y) = \int L(\theta)\pi(\theta)\,d\theta$ is the key statistical use case in this book: always compute it on the log scale with log-sum-exp, and validate a quadrature answer against an independent Monte Carlo estimate.
- Quadrature fails when the integrand is peaked relative to the grid, or when a transform squeezes the mass of interest into a region the grid cannot resolve; both problems are cured by locating the mass first (optimization) and centring the grid or transform on it — the idea taken up systematically by the Laplace approximation in the next chapter.
- Product grids cost n^d evaluations, so quadrature is practical only for small d; higher-dimensional problems require Monte Carlo methods.

9 Laplace Approximation

The previous chapter approximated an integral by a weighted sum of function values on a grid. A grid gets expensive fast: an n-point-per-axis product grid costs n^d evaluations, so quadrature is only practical for $d \lesssim 3$. The **Laplace approximation** takes a completely different route: instead of evaluating the integrand everywhere, it looks only at the single most important point — the mode — and replaces the whole integrand there by the Gaussian that matches its height and curvature. No grid is needed at any dimension, at the price of assuming the integrand is roughly bell-shaped.

9.1 Laplace Approximation Formula

The goal and the idea. Many Bayesian calculations are posterior expectations

$$E_f[a(\theta)] = \frac{\int a(\theta)\, f(\theta)\, d\theta}{\int f(\theta)\, d\theta}, \tag{9.1}$$

where $\theta \in \mathbb{R}^p$, $a(\theta)$ is some quantity of interest (a parameter, a prediction, an indicator), and $f(\theta)$ is an **unnormalized** posterior, typically $f(\theta) = L(\theta)\pi(\theta)$. Both numerator and denominator are p-dimensional integrals.

Write $h(\theta) = -\log f(\theta)$, so $f(\theta) = e^{-h(\theta)}$. For independent data, h is (up to an additive constant) minus the log-likelihood plus minus the log-prior — a sum of n terms — so as n grows it becomes large and sharply curved at its minimum. This is the same large-sample regularity that makes the MLE asymptotically normal (seen in the maximum-likelihood chapters); here it makes the *posterior* asymptotically normal too. The Laplace approximation makes this precise by expanding h to second order at its minimum and integrating the resulting Gaussian exactly — no grid, no sampling, just one optimization and one Hessian.

Derivation. Let $\theta_0 = \arg\min_\theta h(\theta)$, the posterior mode, found numerically (`optim(..., hessian = TRUE)` or `nlm(..., hessian = TRUE)` in R — exactly the optimizers used to find the MLE in earlier chapters). Expand h around θ_0:

$$h(\theta) = h(\theta_0) + \underbrace{\nabla h(\theta_0)^\top}_{=\,0 \text{ at the minimum}} (\theta - \theta_0) + \tfrac{1}{2}(\theta - \theta_0)^\top H (\theta - \theta_0) + O(\|\theta - \theta_0\|^3), \qquad H = \nabla^2 h(\theta_0). \tag{9.2}$$

H, the Hessian of $-\log f$ at the mode, is positive definite at a genuine minimum; for $f = L\pi$ it is the observed information plus the curvature of the log-prior. Dropping the cubic remainder and exponentiating,

$$f(\theta) \approx \tilde{f}(\theta) = f(\theta_0) \exp\Big\{ -\tfrac{1}{2}(\theta - \theta_0)^\top H (\theta - \theta_0) \Big\}. \tag{9.3}$$

The exponential factor is, up to its normalizing constant, the density of $N_p(\theta_0, H^{-1})$,

$$\varphi_H(\theta) = (2\pi)^{-p/2}\, |H|^{1/2} \exp\Big\{ -\tfrac{1}{2}(\theta - \theta_0)^\top H (\theta - \theta_0) \Big\}, \tag{9.4}$$

so multiplying and dividing by $(2\pi)^{-p/2}|H|^{1/2}$ splits $\tilde{f}$ into a constant times a density:

$$\tilde{f}(\theta) = c \cdot \varphi_H(\theta), \qquad c = f(\theta_0)\,(2\pi)^{p/2}\,|H|^{-1/2}. \tag{9.5}$$

Two results follow immediately, since φ_H is a properly normalized density that integrates to 1:

$$\boxed{\theta \mid \text{data} \;\approx\; N_p(\theta_0,\; H^{-1})} \qquad\qquad \boxed{\int f(\theta)\,d\theta \;\approx\; f(\theta_0)\,(2\pi)^{p/2}\,|H|^{-1/2}} \tag{9.6}$$

Geometric meaning of $|H|$. The second result says that the integral is the **height** of f at its mode times a **volume**, $(2\pi)^{p/2}|H|^{-1/2}$. The volume is that of a contour of $\tilde{f}$. The contours are the ellipsoids

$$E_r = \{\theta : (\theta-\theta_0)^\top H(\theta-\theta_0) \le r^2\}. \tag{9.7}$$

If $\lambda_1, \ldots, \lambda_p$ are the eigenvalues of H, E_r has semi-axes of length $r/\sqrt{\lambda_j}$ along the eigenvectors of H, so its volume is

$$\text{vol}(E_r) = V_p\, r^p \prod_{j=1}^{p} \frac{1}{\sqrt{\lambda_j}} = V_p\, r^p\, |H|^{-1/2}, \qquad V_p = \frac{\pi^{p/2}}{\Gamma(p/2+1)}, \tag{9.8}$$

where V_p is the volume of the unit ball in $\mathbb{R}^p$. Setting $\text{vol}(E_r) = (2\pi)^{p/2}|H|^{-1/2}$ and solving for r gives

$$r^* = \sqrt{2}\,\Gamma(p/2+1)^{1/p}, \qquad \int f(\theta)\,d\theta \approx f(\theta_0)\cdot \text{vol}\,(E_{r^*}). \tag{9.9}$$

So the Laplace approximation replaces the bell-shaped f by a flat-topped cylinder of height $f(\theta_0)$ over the contour ellipsoid E_{r^*}, which has the same volume as the Gaussian $\tilde{f}$. For $p=1$, $r^* = \sqrt{\pi/2} \approx 1.25$ and E_{r^*} is the interval $\theta_0 \pm r^*/\sqrt{H}$, of length $\sqrt{2\pi/H}$; for $p=2$, $r^* = \sqrt{2}$ and E_{r^*} is the ellipse of area $\pi r^{*2}|H|^{-1/2} = 2\pi|H|^{-1/2}$. As p grows, $r^* \approx \sqrt{p/e}$, far inside the typical radius $\sqrt{p}$ of the normal distribution, so E_{r^*} holds less and less of the posterior probability.

A large $|H|$ means strong curvature, tight contours and a small integral; a small $|H|$ means a flat, wide peak and a large integral, for the same height $f(\theta_0)$. Figure 9.1 shows this for two Hessians that differ by a factor of 4, so $|H|$ differs by $4^2 = 16$ and the area of E_{r^*} by 4.

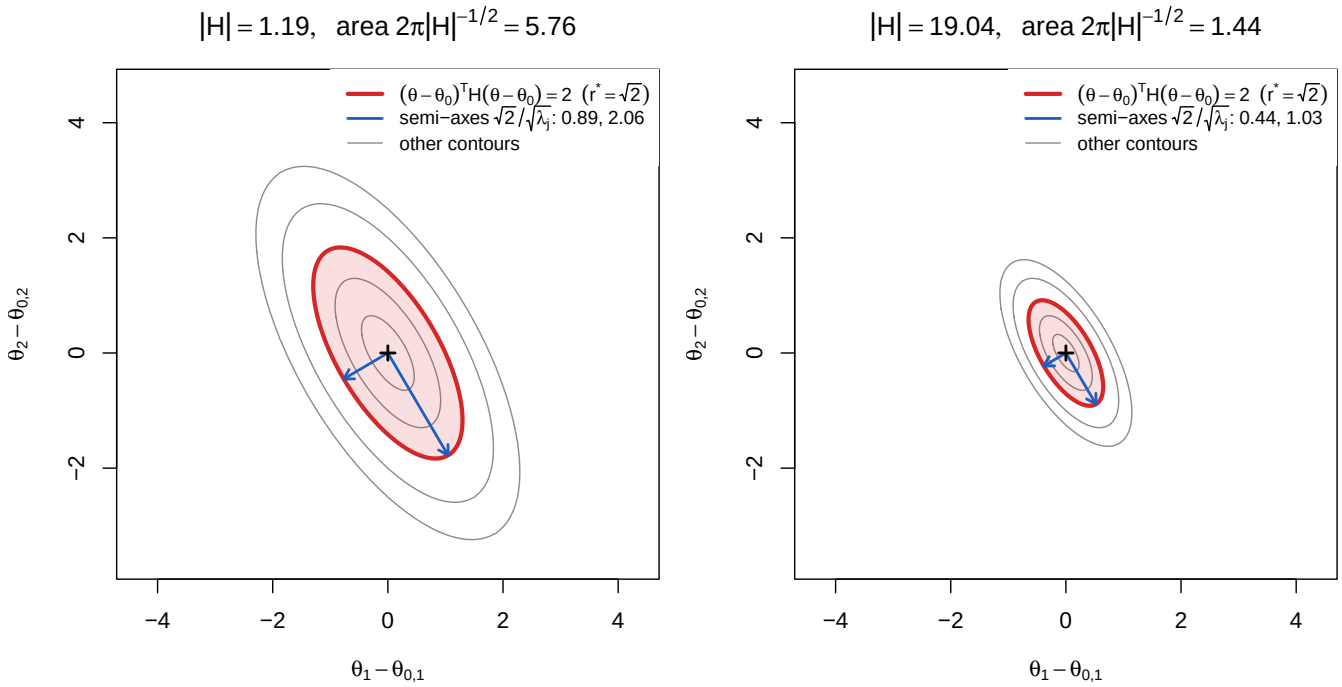


Figure 9.1: Illustrates the geometric meaning of |H| in the Laplace approximation: the integral is the peak height times the area of a contour ellipse, which is proportional to |H|^(-1/2). Contours of exp{-(theta - theta_0)' H (theta - theta_0) / 2} for two Hessians with the same peak height, H (left) and 4H (right), drawn on the same axes. The shaded ellipse is the contour (theta - theta_0)' H (theta - theta_0) = r^2 *with r* = sqrt(2), whose area is 2 pi |H|^(-1/2): a cylinder of height f(theta_0) over it has the same volume as the Gaussian. Its semi-axes (blue) have lengths sqrt(2)/sqrt(lambda_j) along the eigenvectors of H. Multiplying H by 4 shrinks every semi-axis by 2, the area by 4, and so the Laplace integral by 4.

On the $-2\log$ scale, the scale of deviances and information criteria, the second result reads

$$-2\log\int f(\theta)\,d\theta \;\approx\; -2\log f(\theta_0) + \log|H| - p\log(2\pi) \;=\; 2\,h(\theta_0) + \log|H| - p\log(2\pi), \tag{9.10}$$

and for a marginal likelihood, where $f(\theta) = L(\theta)\,\pi(\theta)$ and $P(y) = \int f(\theta)\,d\theta$,

$$-2\log P(y) \;\approx\; -2\log L(\theta_0) - 2\log\pi(\theta_0) + \log|H| - p\log(2\pi). \tag{9.11}$$

The first term measures fit at the mode; the other three penalize through the prior density at the mode and the volume of the posterior, $|H|^{-1/2}$. This is the form from which BIC is derived in Section 9.3.1.

Only the mode and the Hessian of $-\log f$ at the mode are needed — both come for free from any optimizer that returns a Hessian. The relative error of the second result is $O(1/n)$ for regular models; the error of the first is governed by how skewed the true posterior actually is.

9.1.1 Example: A Poisson Rate with a Gamma Prior

To see the approximation against ground truth, use a model with a known exact answer: the Gamma–Poisson conjugate pair from the Bayesian Inference chapter. For $y_1, \ldots, y_n \mid \lambda \sim \text{Poisson}(\lambda)$ and $\lambda \sim \text{Gamma}(\alpha, \beta)$,

the posterior is exactly Gamma($\alpha + \sum y_i,\ \beta + n$), and the marginal likelihood derived there in closed form, $\log P(\mathbf{y}) = \alpha \log \beta - \log \Gamma(\alpha) + \log \Gamma(\alpha + S) - (\alpha + S) \log(\beta + n) - \sum_i \log(y_i!)$ with $S = \sum_i y_i$, is exactly `exact_log_ml` below — so both the exact log marginal likelihood and the exact posterior are available to measure the Laplace error directly.

```
alpha <- 2; beta <- 1

### Laplace approximation: mode, Hessian, and the two boxed results above
laplace_poisson <- function(y) {
  h  <- function(l) -(sum(dpois(y, l, log = TRUE)) + dgamma(l, alpha, beta, log = TRUE))
  op <- optim(mean(y) + 0.5, h, method = "L-BFGS-B", lower = 1e-6, hessian = TRUE)
  c(mode = op$par, se = sqrt(1 / op$hessian[1, 1]),
    log_ml = -op$value + 0.5 * log(2 * pi) - 0.5 * log(op$hessian[1, 1]))
}

### exact log marginal likelihood, from the Gamma-Poisson conjugate identity
exact_log_ml <- function(y) {
  n <- length(y); s <- sum(y)
  alpha * log(beta) - lgamma(alpha) + lgamma(alpha + s) -
    (alpha + s) * log(beta + n) - sum(lfactorial(y))
}
```

```
set.seed(42)
res <- t(sapply(c(5, 20, 100, 500), function(n) {
  y <- rpois(n, 3)
  lap <- laplace_poisson(y)
  c(n = n, laplace_log_ml = lap["log_ml"], exact_log_ml = exact_log_ml(y),
    error = lap["log_ml"] - exact_log_ml(y),
    laplace_mode = lap["mode"], posterior_mean = (alpha + sum(y)) / (beta + n))
}))
knitr::kable(res, digits = 4, row.names = FALSE,
             col.names = c("n", "Laplace log P(y)", "exact log P(y)", "error",
                           "Laplace mode", "posterior mean"))
```

Table 9.1: Assesses the accuracy of the Laplace approximation by comparing it with the exact marginal likelihood in a conjugate example where the exact answer is known. Laplace log marginal likelihood versus the exact value, and the posterior mean versus the Laplace mode, as n grows. The error shrinks roughly like 1/n: it falls by about a factor of 4-5 each time n increases by that factor.

n	Laplace log P(y)	exact log P(y)	error	Laplace mode	posterior mean
5	-11.3822	-11.3784	-0.0038	3.6667	3.8333
20	-44.2274	-44.2263	-0.0011	3.6190	3.6667
100	-197.0022	-197.0019	-0.0003	2.9406	2.9505
500	-958.5983	-958.5983	-0.0001	2.8842	2.8862

The error is a few thousandths at $n = 5$ and shrinks to a few ten-thousandths by $n = 500$; the Laplace mode also

converges to the posterior mean as the Gamma posterior becomes more symmetric. Plotting the exact posterior against its Laplace approximation makes this concrete:

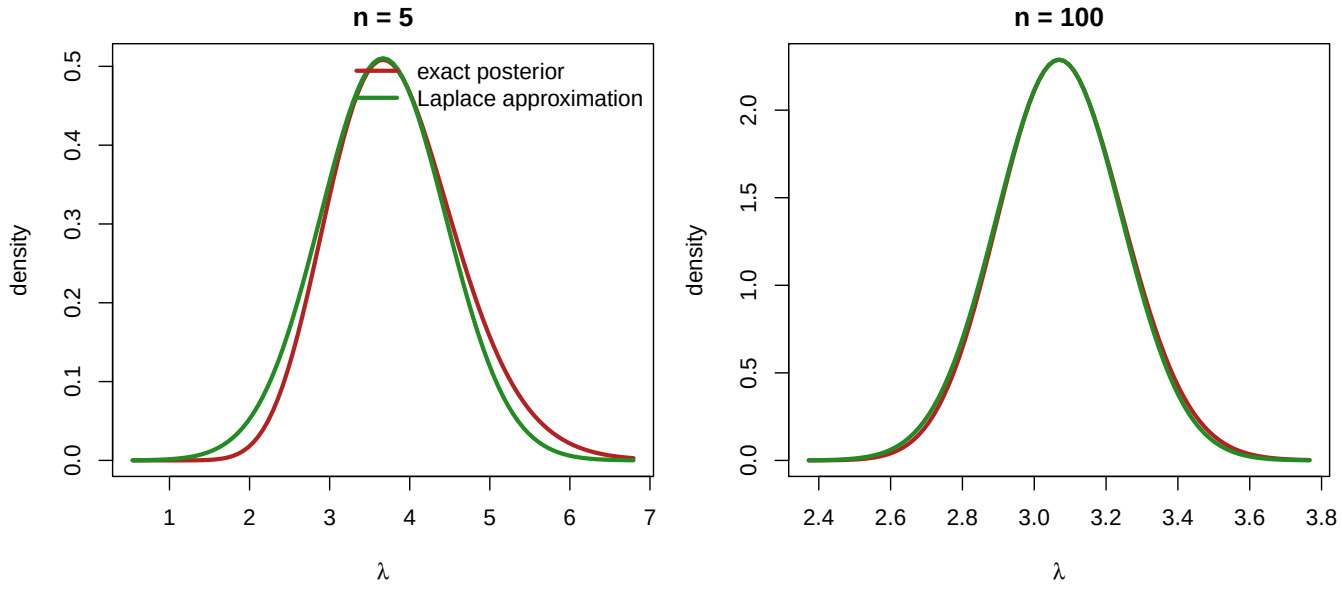


Figure 9.2: Shows how the Laplace approximation replaces a posterior by a normal density centred at the mode, and how well it fits at small and large sample sizes. Exact Gamma posterior (red) and the Laplace normal approximation (green) for n=5 and n=100. At n=5 the exact posterior is visibly right-skewed and the Gaussian is a rough match; by n=100 the two are nearly indistinguishable, illustrating the O(1/n) convergence in the table above.

9.1.2 Linear functionals: posterior mean and credible intervals

If $a(\theta) = A^\top \theta + a_0$ is linear, replacing f by $\tilde{f}$ gives $E_f[a(\theta)] \approx A^\top \theta_0 + a_0$: the posterior mean is approximated by the posterior mode, and $\mathrm{Var}(A^\top \theta) \approx A^\top H^{-1} A$ gives Wald-style credible intervals $\theta_{0j} \pm z_{\alpha/2}\sqrt{(H^{-1})_{jj}}$ — exactly the Bayesian counterpart of an MLE plus its Wald standard error from the maximum-likelihood chapters.

```
set.seed(42)
n <- 20; y <- rpois(n, 3); lap <- laplace_poisson(y)
a1 <- alpha + sum(y); b1 <- beta + n
comp <- data.frame(
  quantity = c("Laplace (mode / se)", "exact posterior (mean / sd)"),
  center = c(lap["mode"], a1 / b1),
  spread = c(lap["se"], sqrt(a1) / b1),
  lower95 = c(lap["mode"] - 1.96 * lap["se"], qgamma(0.025, a1, b1)),
  upper95 = c(lap["mode"] + 1.96 * lap["se"], qgamma(0.975, a1, b1))
)
knitr::kable(comp, digits = 4)
```

Table 9.2: Compares interval estimates based on the Laplace approximation with the exact posterior intervals, to see how much accuracy is lost. Laplace mode/SE-based 95% interval versus the exact Gamma posterior mean/sd-based interval, n = 20.

	quantity	center	spread	lower95	upper95
mode	Laplace (mode / se)	3.6190	0.4151	2.8054	4.4327
	exact posterior (mean / sd)	3.6667	0.4179	2.8937	4.5298

The two intervals agree closely even at $n = 20$: the mild right-skew of the Gamma posterior visible in Figure 9.2 has not yet noticeably distorted a Wald-style interval built from a symmetric Gaussian.

9.1.3 Example: Computing ML of Normal Data with an Unknown Mean

To compare the three methods on an apples-to-apples problem, reuse the single-parameter example from the end of Unit 08, where one unknown mean has to be integrated out.

The model. The data are $n = 3$ observations $y_1, \dots, y_n \mid \mu \overset{iid}{\sim} N(\mu, 1)$ with known variance 1. Their mean $\hat{\mu} = \bar{y}$ is sufficient for μ, and $\hat{\mu} \mid \mu \sim N(\mu, 1/n)$, so we take the likelihood to be the density of $\hat{\mu}$,

$$L(\mu) = N(\hat{\mu};\, \mu,\, 1/n) = \sqrt{\frac{n}{2\pi}}\, \exp\Big\{ -\frac{n}{2}(\hat{\mu} - \mu)^2 \Big\}. \tag{9.12}$$

(The full likelihood $\prod_i N(y_i; \mu, 1)$ differs from this only by a factor that does not involve μ.) The prior is $\pi(\mu) = N(\mu;\, 0,\, \sigma_\pi^2)$ with $\sigma_\pi = 5$. To see how each method copes as the data move away from where the prior puts its mass, $\hat{\mu}$ is set to 0, -5 and -20, i.e. 0, 1 and 4 prior standard deviations from the prior mean.

The target. The marginal likelihood is the integral of the unnormalized posterior $f(\mu) = L(\mu)\, \pi(\mu)$ over the whole real line,

$$P(y) = \int_{-\infty}^{\infty} f(\mu)\, d\mu, \qquad f(\mu) = N(\hat{\mu};\, \mu,\, 1/n)\, N(\mu;\, 0,\, \sigma_\pi^2). \tag{9.13}$$

Because the prior and likelihood are a normal-normal conjugate pair, everything is available in closed form. The posterior is

$$\mu \mid y \sim N(m,\, v), \qquad v = \Big(n + \frac{1}{\sigma_\pi^2}\Big)^{-1} = \frac{1}{3.04} \approx 0.574^2, \qquad m = v\, n\, \hat{\mu} \approx 0.987\, \hat{\mu}, \tag{9.14}$$

a narrow bump that follows the data, and integrating μ out gives $\hat{\mu} \sim N(0,\, 1/n + \sigma_\pi^2)$ marginally, so the **true** value is

$$P(y) = N(\hat{\mu};\, 0,\, 1/n + \sigma_\pi^2). \tag{9.15}$$

Laplace. Laplace attacks the integral by optimization instead of a grid. Let $h(\mu) = -\log f(\mu)$; `nlm()` finds its minimizer $\hat{\mu}_L$ and the second derivative $H = h''(\hat{\mu}_L)$, and the boxed formula with $p = 1$ gives

$$\log P(y) \approx \log f(\hat{\mu}_L) + \tfrac{1}{2} \log(2\pi) - \tfrac{1}{2} \log H. \tag{9.16}$$

Here $\hat{\mu}_L = m$ and $H = 1/v$, and since f is exactly a normal kernel, the Laplace approximation is exact for every $\hat{\mu}$.

Quadrature on a logistic grid. The midpoint rule needs a finite interval, so, as in Unit08, the real line is first mapped to $(0, 1)$ by the logistic transform

$$u = \frac{1}{1 + e^{-\mu}}, \qquad \mu = \operatorname{logit}(u) = \log \frac{u}{1 - u}, \qquad \frac{d\mu}{du} = \frac{1}{u(1 - u)}. \tag{9.17}$$

Substituting $\mu = \operatorname{logit}(u)$, $d\mu = du/\{u(1-u)\}$ turns the target into an integral over the unit interval,

$$P(y) = \int_{-\infty}^{\infty} f(\mu)\, d\mu = \int_0^1 g(u)\, du, \qquad g(u) = \frac{f(\operatorname{logit}(u))}{u(1 - u)}. \tag{9.18}$$

The midpoint rule with N equal cells of width $1/N$, evaluated at the cell centres $u_i = (i - \frac{1}{2})/N$, then gives

$$P(y) \approx \frac{1}{N} \sum_{i=1}^{N} g(u_i) = \frac{1}{N} \sum_{i=1}^{N} \frac{f(\mu_i)}{u_i(1 - u_i)}, \qquad \mu_i = \operatorname{logit}(u_i), \tag{9.19}$$

with $N = 100$ below, computed on the log scale with log-sum-exp. The grid is fixed in advance: its points μ_i are dense near $\mu = 0$ and sparse far from it. The posterior at $\hat{\mu} = -20$ sits near $u \approx e^{-20} \approx 2 \times 10^{-9}$, in an interval only a few times 10^{-9} wide in u, far inside the first cell, whose centre is $u_1 = 0.005$, i.e. $\mu_1 \approx -5.3$.

Monte Carlo from the prior. Writing the target as an expectation under the prior, $P(y) = \int L(\mu)\, \pi(\mu)\, d\mu = E_\pi[L(\mu)]$, the naive Monte Carlo estimator of Unit08 draws $\mu^{(1)}, \ldots, \mu^{(M)} \overset{iid}{\sim} \pi$ and averages the likelihood,

$$P(y) \approx \frac{1}{M} \sum_{m=1}^{M} L(\mu^{(m)}), \tag{9.20}$$

with $M = 10^6$. Only draws that land under the posterior bump contribute; at $\hat{\mu} = -20$ that is about $\Phi(-3.8) \approx 7 \times 10^{-5}$ of the draws, roughly a hundred out of a million.

```
n_obs <- 3; sigma_prior <- 5                  # n = 3 observations; same setup as the Unit08
 ↪  app

### true value: exact, since f is a normal-normal conjugate kernel, so marginally
### mu_hat ~ N(0, 1/n_obs + sigma_prior^2). (integrate(f, -Inf, Inf) is not safe here:
### for mu_hat far from 0 it misses the narrow spike of f and returns a wrong value.)
log_mar_true_mu <- function(mu_hat) dnorm(mu_hat, 0, sqrt(1 / n_obs + sigma_prior^2), log =
 ↪  TRUE)

### Laplace: p = 1 case of the boxed formula, mode and Hessian from nlm()
laplace_mu <- function(mu_hat) {
  h  <- function(mu) -(dnorm(mu_hat, mu, 1 / sqrt(n_obs), log = TRUE) +
                         dnorm(mu, 0, sigma_prior, log = TRUE))
  op <- nlm(h, p = mu_hat, hessian = TRUE)
  c(mode = op$estimate, log_ml = -op$minimum + 0.5 * log(2 * pi) - 0.5 * log(op$hessian[1, 1]))
}

### naive Monte Carlo: draw mu from the prior, average the likelihood
log_sum_exp <- function(lx) { m <- max(lx); m + log(sum(exp(lx - m))) }
log_mar_mc_mu <- function(mu_hat, iters) {
  mus <- rnorm(iters, 0, sigma_prior)
```

```
  log_sum_exp(dnorm(mu_hat, mus, 1 / sqrt(n_obs), log = TRUE)) - log(iters)
}

### midpoint rule on the logistic-transformed (0, 1) grid, as in Unit08
log_mar_mid_mu <- function(mu_hat, n) {
  h  <- 1 / n
  u  <- (seq_len(n) - 0.5) * h
  mu <- log(u / (1 - u))
  log_f <- dnorm(mu_hat, mu, 1 / sqrt(n_obs), log = TRUE) +
    dnorm(mu, 0, sigma_prior, log = TRUE) - log(u * (1 - u))
  log_sum_exp(log_f) + log(h)
}
```

```
set.seed(7)
mu_hats <- c(0, -5, -20)
three_way <- t(sapply(mu_hats, function(m) {
  c(True = log_mar_true_mu(m),
    Laplace = laplace_mu(m)["log_ml"],
    Monte_Carlo_pi = log_mar_mc_mu(m, 1e6),
    Quadrature = log_mar_mid_mu(m, 100))
}))
knitr::kable(cbind(`mu` = mu_hats, round(three_way, 3)), row.names = FALSE,
             col.names = c("mu", "True", "Laplace", "Monte Carlo (pi)", "Quadrature (N=100)"))
```

Table 9.3: Compares the Laplace approximation with grid quadrature and naive Monte Carlo against the true value as the posterior moves away from the prior mean, to show where each method succeeds or fails. Log marginal likelihood by four methods, for data locations mu_hat = 0, -5, -20 (n = 3 observations with sd 1, prior N(0, 5^2) on mu). Quadrature uses a 100-point midpoint grid; Monte Carlo uses 1e6 prior draws; both are unchanged from Unit08's app.

mu	True	Laplace	Monte Carlo (pi)	Quadrature (N=100)
0	-2.535	-2.535	-2.537	-2.535
-5	-3.028	-3.028	-3.027	-2.707
-20	-10.430	-10.430	-10.626	-327.191

At $\mu = 0$ the posterior sits right where both the logistic grid and the prior put their effort, so all four numbers should agree closely. As μ moves to -5 and then -20, the fixed grid — centred at 0 by construction — squeezes the posterior into an ever-thinner sliver of the unit interval, and quadrature is the first to break away from the true value, exactly the failure diagnosed at the end of Unit08. Naive Monte Carlo is built on the same fragile assumption in a different guise: it needs prior draws *near* μ to contribute any likelihood at all, so once μ is several prior standard deviations from 0, a fixed budget of draws from π can also become too sparse there to be trusted, even though nothing about the Monte Carlo estimator's derivation assumed μ would be small. Laplace has neither failure mode: it optimizes to find the mode wherever it actually is and builds its approximation *around that point*, so it automatically "recentres" itself for every μ — the same fix Unit08 proposed by hand for quadrature, done here for free by the optimizer.

What the quadratic approximation looks like in two dimensions. In this example $\log f(\mu)$ is exactly a quadratic in μ, so its quadratic approximation, $\log f(\hat{\mu}_L) - \frac{1}{2}H(\mu-\hat{\mu}_L)^2$, is not an approximation at all, which is why Laplace is exact here. That is no longer true once the variance is unknown too. Let the data be $y_1, \dots, y_n \overset{iid}{\sim} N(\mu, \sigma^2)$ with $w = \log\sigma$, priors $\mu \sim N(0, 5^2)$ and $w \sim N(0, 5^2)$, and $f(\mu, w) = L(\mu, w)\,\pi(\mu)\,\pi(w)$ (the model of Section 9.2.4). With $\hat{\theta} = (\hat{\mu}, \hat{w})$ the mode of f and H the 2×2 Hessian of $-\log f$ there, Laplace replaces

$$\log f(\theta) \quad \text{by} \quad \log f(\hat{\theta}) - \tfrac{1}{2}(\theta - \hat{\theta})^\top H\,(\theta - \hat{\theta}), \qquad \theta = (\mu, w), \tag{9.21}$$

whose contours are ellipses. Figure 9.3 compares the contours of the two at the same levels, for simulated data with true $\mu = 1$, $\sigma = 2$.

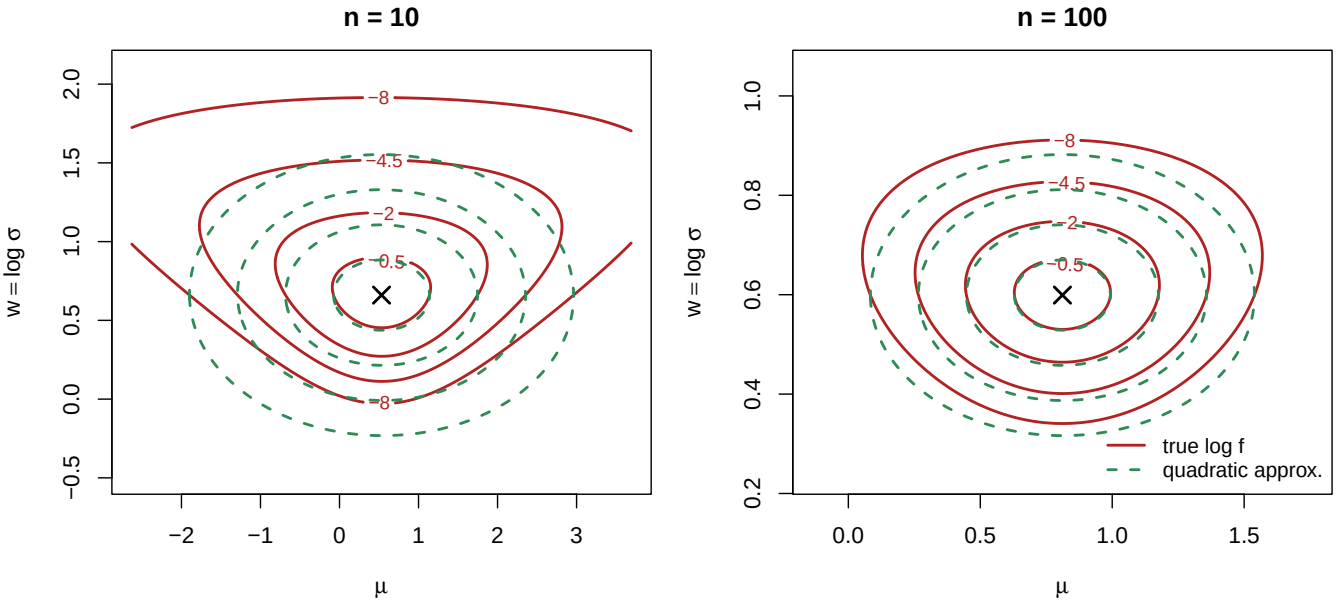


Figure 9.3: Shows where the quadratic (Laplace) approximation of a two-parameter log posterior is accurate and where it is not. Contours of $\log f(\mu, w) - \log f(\hat{\theta})$ (red, solid) and of its quadratic approximation $-\frac{1}{2}(\theta - \hat{\theta})^\top H\,(\theta - \hat{\theta})$ (green, dashed) at the levels $-r^2/2$ for $r = 1, 2, 3, 4$, for $n = 10$ (left) and $n = 100$ (right) observations from $N(1, 2^2)$, with $w = \log\sigma$ and priors $\mu \sim N(0, 5^2)$, $w \sim N(0, 5^2)$. The cross marks the mode $\hat{\theta}$. Near the mode the two sets of contours agree; further out the true contours are bent into a banana, rising in w as μ moves away from the mode, while the quadratic approximation keeps them elliptical. The disagreement is large at $n = 10$ and much smaller at $n = 100$.

At $n = 10$ the inner contour ($r = 1$), which holds most of the posterior mass, is matched well, but the outer contours of $\log f$ form a banana: when μ is far from $\bar{y}$, the data are best explained by a larger σ, so the ridge of f curves upward in w, and the posterior has a heavier tail towards large w than the ellipse allows. At $n = 100$ the curvature at the mode is 10 times larger and the region where f has appreciable mass shrinks to a neighbourhood where the quadratic expansion is accurate, so the two sets of contours nearly coincide. This is the $O(1/n)$ accuracy of Laplace seen in picture form, and the reason the Laplace-adapted rule of Section 9.2.4 still has a small error in this model.

9.2 Quadrature Adapted by the Laplace Approximation

A fixed logistic transform ($u = 1/(1 + e^{-\mu})$) fails when the posterior mass moves far from $\mu = 0$. The remedy is to let the Laplace approximation tell the quadrature rule where to look. Laplace approximates the posterior by $N(\hat{\mu}_L, s_L^2)$, where $\hat{\mu}_L$ is the mode of f and $s_L = 1/\sqrt{H}$ with H the second derivative of $-\log f$ at $\hat{\mu}_L$. The Laplace-adapted midpoint rule below places its grid points according to this normal distribution, so it follows the posterior wherever it sits.

9.2.1 Univraite Laplace-adapted midpoint rule

A general transformation. The midpoint rule needs a finite interval to partition, so the real line is first mapped to $(0, 1)$. Let t be any strictly increasing, differentiable map from $(-\infty, \infty)$ onto $(0, 1)$, that is, the CDF of a continuous distribution, with density $t'(\mu) > 0$. Substituting

$$u = t(\mu), \qquad \mu = t^{-1}(u), \qquad d\mu = \frac{du}{t'(t^{-1}(u))} \tag{9.22}$$

turns the target into an integral over the unit interval,

$$P(y) = \int_{-\infty}^{\infty} f(\mu)\, d\mu = \int_0^1 \frac{f(t^{-1}(u))}{t'(t^{-1}(u))}\, du, \tag{9.23}$$

and the midpoint rule with N equal cells and centres $u_i = (i - \frac{1}{2})/N$ gives

$$P(y) \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f(\mu_i)}{t'(\mu_i)}, \qquad \mu_i = t^{-1}(u_i). \tag{9.24}$$

The integrand over u is f divided by the density t'. It is nearly constant, and a few points are enough, when t' has the same shape as f; it is a narrow spike, which a fixed grid can miss, when t' puts its mass somewhere else. A fixed choice such as the logistic CDF $t(\mu) = 1/(1 + e^{-\mu})$, with $t'(\mu) = t(\mu)\{1 - t(\mu)\}$, matches f only when the posterior happens to sit near 0 with a scale near 1.

The Laplace choice. The best available description of the shape of f is its Laplace approximation, which for $p = 1$ is

$$\tilde{f}(\mu) = f(\hat{\mu}_L) \exp\Big\{ -\frac{(\mu - \hat{\mu}_L)^2}{2s_L^2} \Big\}, \qquad s_L = H^{-1/2}, \tag{9.25}$$

where $\hat{\mu}_L$ is the mode of f and H the second derivative of $-\log f$ there. Normalized, $\tilde{f}$ is the density of $N(\hat{\mu}_L, s_L^2)$; write

$$\phi(x;\, \hat{\mu}_L,\, s_L) = \frac{1}{s_L\sqrt{2\pi}} \exp\Big\{ -\frac{(x - \hat{\mu}_L)^2}{2s_L^2} \Big\}, \qquad \Phi(x;\, \hat{\mu}_L,\, s_L) = \int_{-\infty}^{x} \phi(v;\, \hat{\mu}_L,\, s_L)\, dv \tag{9.26}$$

for this density and its CDF, and $\Phi^{-1}(u;\, \hat{\mu}_L,\, s_L)$ for the quantile function (`dnorm`, `pnorm` and `qnorm` with `mean = mu_L, sd = s_L` in R). Taking t to be the CDF of the normalized $\tilde{f}$,

$$t(\mu) = \Phi(\mu;\, \hat{\mu}_L,\, s_L), \qquad t^{-1}(u) = \Phi^{-1}(u;\, \hat{\mu}_L,\, s_L), \qquad t'(\mu) = \phi(\mu;\, \hat{\mu}_L,\, s_L), \tag{9.27}$$

the general integral becomes

$$P(y) = \int_0^1 g(u)\,du, \qquad g(u) = \frac{f(\mu(u))}{\phi(\mu(u);\,\hat{\mu}_L,\,s_L)}, \qquad \mu(u) = \Phi^{-1}(u;\,\hat{\mu}_L,\,s_L). \tag{9.28}$$

The integrand $g(u)$ is the posterior kernel divided by the Laplace normal density at the same point. If the posterior were exactly $N(\hat{\mu}_L, s_L^2)$, g would be constant on $(0,1)$, equal to the Laplace approximation $f(\hat{\mu}_L)/\phi(\hat{\mu}_L;\,\hat{\mu}_L,\,s_L) = \sqrt{2\pi}\,s_L f(\hat{\mu}_L)$. The midpoint rule on the same cells then gives

$$P(y) \approx \frac{1}{N}\sum_{i=1}^{N} g(u_i) = \frac{1}{N}\sum_{i=1}^{N} \frac{f(\mu_i)}{\phi(\mu_i;\,\hat{\mu}_L,\,s_L)}, \qquad \mu_i = \Phi^{-1}(u_i;\,\hat{\mu}_L,\,s_L). \tag{9.29}$$

Compared with the logistic map, only t has changed: the grid points μ_i are now the quantiles of the Laplace normal, so they sit under the posterior wherever it is. The error decreases like $1/N^2$, as for any midpoint rule, only when g is smooth and bounded on $[0,1]$. When the posterior has heavier tails than its Laplace normal approximation, $g(u)$ grows without bound as $u \to 0$ or 1, because $\Phi^{-1}(u;\,\hat{\mu}_L,\,s_L)$ is steep there and the outermost cells, each standing for a whole tail, sample it poorly; convergence is then much slower.

The table below applies the rule to the example of Section 9.1.3. Because $f(\mu)$ is a normal–normal kernel here, the posterior is exactly normal, $g(u)$ is exactly constant, and the rule is exact with any number of points, wherever the data sit. The two-parameter example below, where the posterior of $(\mu, \log\sigma)$ is not exactly normal, shows the rule at work when g is only approximately constant.

```
### Laplace-adapted midpoint rule: mu_i = qnorm(u_i, mu_L, s_L), g(u_i) = f(mu_i) / dnorm(mu_i,
↪  mu_L, s_L)
log_mar_adapted <- function(mu_hat, n) {
  lf <- function(mu) dnorm(mu_hat, mu, 1 / sqrt(n_obs), log = TRUE) + dnorm(mu, 0, sigma_prior,
↪  log = TRUE)
  h  <- function(mu) -lf(mu)
  op <- nlm(h, p = mu_hat, hessian = TRUE)
  m  <- op$estimate; sL <- 1 / sqrt(op$hessian[1, 1])
  u  <- (seq_len(n) - 0.5) / n
  mu <- qnorm(u, m, sL)
  log_sum_exp(lf(mu) - dnorm(mu, m, sL, log = TRUE)) - log(n)
}
adapted <- t(sapply(mu_hats, function(m) c(
  True                 = log_mar_true_mu(m),
  Laplace              = unname(laplace_mu(m)["log_ml"]),
  `Logistic midpoint (N=100)` = log_mar_mid_mu(m, 100),
  `Laplace-adapted midpoint (N=5)` = log_mar_adapted(m, 5))))
knitr::kable(cbind(mu = mu_hats, round(adapted, 3)), row.names = FALSE)
```

Table 9.4: Log marginal likelihood for data locations mu_hat = 0, -5, -20 (setup as in the table above): the logistic midpoint rule with 100 points, and the Laplace-adapted midpoint rule with only 5 points.

mu	True	Laplace	Logistic midpoint (N=100)	Laplace-adapted midpoint (N=5)
0	-2.535	-2.535	-2.535	-2.535
-5	-3.028	-3.028	-2.707	-3.028
-20	-10.430	-10.430	-327.191	-10.430

Table 9.4: Log marginal likelihood for data locations mu_hat = 0, -5, -20 (setup as in the table above): the logistic midpoint rule with 100 points, and the Laplace-adapted midpoint rule with only 5 points.

mu	True	Laplace	Logistic midpoint (N=100)	Laplace-adapted midpoint (N=5)

9.2.2 Example: Computing ML of Normal Data with an Unknown Mean

This example compares the logistic midpoint rule with the Laplace-adapted midpoint rule on the normal-mean model of Section 9.1.3, and can be explored in a shinylive app. The model is $n = 3$ observations from $N(\mu, 1)$ with sample mean $\hat{\mu}$, so $L(\mu) = N(\hat{\mu};\, \mu,\, 1/3)$, prior $\pi(\mu) = N(\mu;\, 0,\, 5^2)$, and $f(\mu) = L(\mu)\,\pi(\mu)$. Sliders set the data location $\hat{\mu}$ (from -5 to 25) and the number N of grid points. The app computes $P(y) = \int f(\mu)\, d\mu$ by the two midpoint rules, each written as an integral over $u \in (0, 1)$:

- *Logistic grid* (top row), with $\mu = \text{logit}(u)$:

$$P(y) = \int_0^1 \frac{f(\,\text{logit}(u))}{u(1-u)}\, du \;\approx\; \frac{1}{N}\sum_{i=1}^{N} \frac{f(\,\text{logit}(u_i))}{u_i(1-u_i)}. \tag{9.30}$$

- *Laplace-adapted grid* (bottom row), with $\mu = \Phi^{-1}(u;\, \hat{\mu}_L,\, s_L)$ and $\hat{\mu}_L, s_L$ recomputed for each $\hat{\mu}$:

$$P(y) = \int_0^1 \frac{f(\Phi^{-1}(u;\, \hat{\mu}_L,\, s_L))}{\phi(\Phi^{-1}(u;\, \hat{\mu}_L,\, s_L);\, \hat{\mu}_L,\, s_L)}\, du \;\approx\; \frac{1}{N}\sum_{i=1}^{N} \frac{f(\mu_i)}{\phi(\mu_i;\, \hat{\mu}_L,\, s_L)}, \qquad \mu_i = \Phi^{-1}(u_i;\, \hat{\mu}_L,\, s_L). \tag{9.31}$$

In each row the left plot shows $f(\mu)$ with the grid points μ_i, and the right plot shows the integrand over u with the midpoint rectangles, whose total area is the estimate; the true value $N(\hat{\mu};\, 0,\, 1/3 + 5^2)$ is shown for comparison. As $\hat{\mu}$ moves away from 0, the logistic integrand becomes a spike near $u = 1$ that falls between grid points and the estimate collapses, while the Laplace-adapted integrand stays flat and a few rectangles give the right area.

9.2.3 Multivariate Laplace-adapted midpoint rule

For $\theta \in \mathbb{R}^p$ the Laplace approximation of f is

$$\tilde{f}(\theta) = f(\hat{\theta})\, \exp\Big\{ -\tfrac{1}{2}(\theta - \hat{\theta})^\top H\, (\theta - \hat{\theta}) \Big\}, \tag{9.32}$$

whose normalized form is the density $\phi_p(\theta;\, \hat{\theta},\, H^{-1})$ of $N_p(\hat{\theta}, H^{-1})$. Applying the one-dimensional map to each coordinate of θ separately would ignore the correlations in H^{-1}: a product grid aligned with the θ axes puts many points where the tilted posterior ellipse has no mass. The remedy is to standardize θ first, so that the Laplace normal becomes $N_p(0, I)$, whose coordinates are independent, and only then map each coordinate to $(0, 1)$.

Step 1: standardize with the Cholesky factor. Factor the Laplace covariance as

$$H^{-1} = C C^\top, \tag{9.33}$$

with C triangular (the Cholesky factor), and change variables to

$$\theta = \hat{\theta} + Cz, \qquad z = C^{-1}(\theta - \hat{\theta}), \qquad d\theta = |\det C|\, dz. \tag{9.34}$$

Since $C^\top H C = C^\top (CC^\top)^{-1} C = I$, the quadratic form becomes $(\theta-\hat{\theta})^\top H (\theta-\hat{\theta}) = z^\top z$, so in the z coordinates the Laplace approximation is spherical,

$$\tilde{f}(\hat{\theta} + Cz) = f(\hat{\theta})\, e^{-z^\top z/2} = f(\hat{\theta})\, (2\pi)^{p/2} \prod_{j=1}^{p} \phi(z_j), \tag{9.35}$$

with ϕ the standard normal density. The factor C removes the correlations and rescales every direction to unit spread; $|\det C| = |H|^{-1/2}$ is the volume factor of Section 9.1. Any square root of H^{-1}, such as the symmetric one from the eigendecomposition, would make $z^\top z$ spherical too; the Cholesky factor is simply the cheapest to compute. In R, `chol(H)` returns an upper-triangular R with $H = R^\top R$, and $C = R^{-1}$ (`backsolve(R, diag(p))`) satisfies $CC^\top = (R^\top R)^{-1} = H^{-1}$, with $|\det C| = 1/\prod_j R_{jj}$.

Step 2: map each coordinate to $(0,1)$. Apply the standard normal CDF to each z_j,

$$u_j = \Phi(z_j), \qquad z_j = \Phi^{-1}(u_j), \qquad dz_j = \frac{du_j}{\phi(z_j)}, \qquad j = 1, \dots, p. \tag{9.36}$$

Combining the two steps, the target becomes an integral over the unit cube,

$$P(y) = \int_{\mathbb{R}^p} f(\theta)\, d\theta = \int_{(0,1)^p} g(u)\, du, \qquad g(u) = \frac{|\det C|\, f(\hat{\theta} + C\, z(u))}{\prod_{j=1}^{p} \phi(z_j(u))}, \qquad z(u) = (\Phi^{-1}(u_1), \dots, \Phi^{-1}(u_p)). \tag{9.37}$$

Because $\phi_p(\hat{\theta}+Cz;\, \hat{\theta},\, H^{-1}) = (2\pi)^{-p/2}|H|^{1/2} e^{-z^\top z/2} = \prod_j \phi(z_j) \,/\, |\det C|$, the integrand is again the posterior kernel divided by the Laplace normal density,

$$g(u) = \frac{f(\theta(u))}{\phi_p(\theta(u);\, \hat{\theta},\, H^{-1})}, \qquad \theta(u) = \hat{\theta} + C\, z(u), \tag{9.38}$$

exactly as for $p = 1$, where $C = s_L$. If f were exactly $\tilde{f}$, g would be constant on the cube, equal to the Laplace approximation $f(\hat{\theta})(2\pi)^{p/2}|H|^{-1/2}$.

Step 3: product midpoint rule. With k cells per axis and centres $u_{(i)} = (i - \tfrac{1}{2})/k$, the product rule uses the $N = k^p$ points

$$\theta_{\mathbf{i}} = \hat{\theta} + C\, z_{\mathbf{i}}, \qquad z_{\mathbf{i}} = (\Phi^{-1}(u_{(i_1)}), \dots, \Phi^{-1}(u_{(i_p)})), \qquad \mathbf{i} = (i_1, \dots, i_p) \in \{1, \dots, k\}^p, \tag{9.39}$$

and gives

$$P(y) \approx \frac{1}{k^p} \sum_{\mathbf{i}} g(u_{\mathbf{i}}) = \frac{|\det C|}{k^p} \sum_{\mathbf{i}} \frac{f(\theta_{\mathbf{i}})}{\prod_{j=1}^{p} \phi(z_{\mathbf{i}j})}. \tag{9.40}$$

The grid $z_{\mathbf{i}}$ is a square lattice of normal quantiles; C maps it onto a sheared lattice that follows the orientation and spread of the posterior ellipse. As in one dimension, the sum is computed on the log scale, $\log g = \log f(\theta_{\mathbf{i}}) + \log|\det C| - \sum_j \log \phi(z_{\mathbf{i}j})$, followed by log-sum-exp. Two limitations carry over or get worse with p: the number

of points k^p grows exponentially, so the product rule is practical only for small p; and when the posterior has heavier tails than its Laplace normal, g grows towards the faces of the cube and convergence in k is slow.

Figure 9.4 follows the two maps for $p = 2$, $k = 8$, $\hat{\theta} = (1, 2)$ and

$$H = \begin{pmatrix} 2.78 & -2.22 \\ -2.22 & 2.78 \end{pmatrix}, \qquad H^{-1} = \begin{pmatrix} 1 & 0.8 \\ 0.8 & 1 \end{pmatrix}, \qquad C = \begin{pmatrix} 1 & 0 \\ 0.8 & 0.6 \end{pmatrix}. \tag{9.41}$$

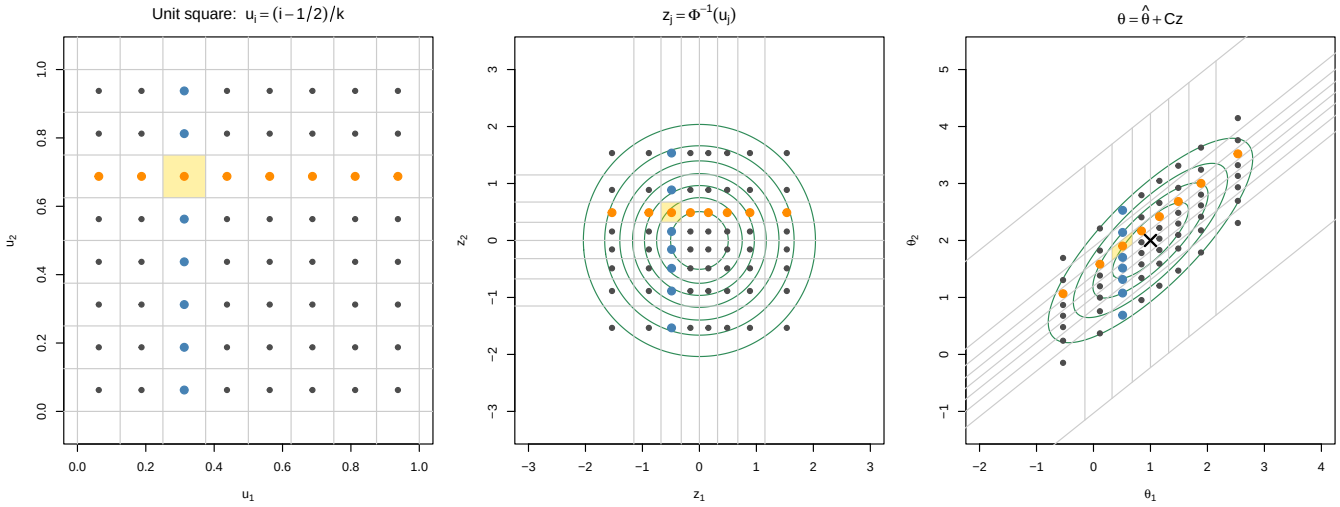


Figure 9.4: Shows how the Laplace-adapted product grid is built: equal cells on the unit square are mapped to normal quantiles, then sheared by the Cholesky factor onto the posterior ellipse. Left: the k x k = 8 x 8 midpoint grid u_i on (0,1)^2, with cell edges in gray. Middle: z = (Phi^{-1}(u_1), Phi^{-1}(u_2)), a square lattice of standard normal quantiles, with contours of the N_2(0, I) density; the outermost cells stretch to infinity. Right: theta = theta_hat + C z with theta_hat = (1, 2) and H^{-1} = C C' having correlation 0.8, with contours of the Laplace normal N_2(theta_hat, H^{-1}); the cross marks theta_hat. One row (orange) and one column (blue) of points, and one cell (shaded), are tracked through both maps: lines of constant z_1 stay vertical, because C is lower triangular, while lines of constant z_2 are tilted along the ellipse.

```
### Laplace-adapted product midpoint rule in p dimensions, k points per axis.
### log_f takes one parameter vector; the optional log_f_rows evaluates log f on every row
### of a matrix at once, which is much faster when k^p is large.
log_mar_adapted_mv <- function(log_f, theta0, k, log_f_rows = NULL) {
  op     <- optim(theta0, function(th) -log_f(th), method = "BFGS", hessian = TRUE)
  th_hat <- op$par; p <- length(th_hat)
  R  <- chol(op$hessian)                          # H = R'R
  C  <- backsolve(R, diag(p))                     # C = R^{-1}, so C C' = H^{-1}
  z1 <- qnorm((seq_len(k) - 0.5) / k)             # normal quantiles on one axis
  Z  <- as.matrix(expand.grid(rep(list(z1), p))) # k^p x p lattice of z
  Th <- sweep(Z %*% t(C), 2, th_hat, "+")         # theta_i = theta_hat + C z_i
  lf <- if (is.null(log_f_rows)) apply(Th, 1, log_f) else log_f_rows(Th)
  log_g <- lf + sum(log(diag(C))) - rowSums(dnorm(Z, log = TRUE))
  log_sum_exp(log_g) - p * log(k)
}
```

9.2.4 Example: Computing ML of Normal Data with Unknown Mean and Variance

This example extends the comparison to two parameters, and can be explored in a shinylive app. The data are $y_1, \dots, y_n \mid \mu, \sigma \overset{iid}{\sim} N(\mu, \sigma^2)$, simulated with the slider "data location" as the true mean, $\sigma = 2$, and n set by a slider (10 to 200). With $w = \log \sigma$, so that both parameters live on the whole real line, the likelihood is

$$L(\mu, w) = \prod_{i=1}^{n} N(y_i;\, \mu,\, e^{2w}) = (2\pi)^{-n/2}\, e^{-nw} \exp\Big\{ -\frac{e^{-2w}}{2} \sum_{i=1}^{n} (y_i - \mu)^2 \Big\}, \tag{9.42}$$

the priors are independent, $\mu \sim N(0, \sigma_\mu^2)$ and $w \sim N(0, \sigma_w^2)$ (both standard deviations set by sliders), $f(\mu, w) = L(\mu, w)\, \pi(\mu)\, \pi(w)$, and the target is the double integral $P(y) = \iint f(\mu, w)\, d\mu\, dw$. Three methods are compared with a reference value from a fine grid centred on the posterior mode:

- *Product midpoint rule on a logistic grid*, with $\mu = \operatorname{logit}(u)$ and $w = \operatorname{logit}(v)$ mapped separately:

$$P(y) = \int_0^1 \int_0^1 \frac{f(\operatorname{logit}(u),\, \operatorname{logit}(v))}{u(1-u)\, v(1-v)}\, du\, dv \;\approx\; \frac{1}{N^2} \sum_{i=1}^{N} \sum_{l=1}^{N} \frac{f(\operatorname{logit}(u_i),\, \operatorname{logit}(v_l))}{u_i(1-u_i)\, v_l(1-v_l)}, \tag{9.43}$$

  with $u_i = (i - \tfrac{1}{2})/N$ and $v_l = (l - \tfrac{1}{2})/N$.
- *Laplace approximation* with $p = 2$: $\log P(y) \approx \log f(\hat\theta) + \log(2\pi) - \tfrac{1}{2} \log |H|$, where $\hat\theta = (\hat\mu, \hat w)$ is the posterior mode and H the 2×2 Hessian of $-\log f$ there.
- *Product midpoint rule on a Laplace-adapted grid*, the $p = 2$ case of Section 9.2.3. With the Cholesky factor $H^{-1} = CC^\top$, $\theta = (\mu, w) = \hat\theta + Cz$ and $u = \Phi(z_1)$, $v = \Phi(z_2)$,

$$P(y) = \int_0^1 \int_0^1 \frac{|\det C|\, f(\hat\theta + C\, z(u, v))}{\phi(\Phi^{-1}(u))\, \phi(\Phi^{-1}(v))}\, du\, dv, \qquad z(u, v) = (\Phi^{-1}(u),\, \Phi^{-1}(v)), \tag{9.44}$$

  and with k points per axis,

$$P(y) \approx \frac{|\det C|}{k^2} \sum_{i=1}^{k} \sum_{l=1}^{k} \frac{f(\hat\theta + C\, z(u_i, v_l))}{\phi(\Phi^{-1}(u_i))\, \phi(\Phi^{-1}(v_l))}, \qquad u_i = \frac{i - \frac{1}{2}}{k}, \quad v_l = \frac{l - \frac{1}{2}}{k}. \tag{9.45}$$

The top row of plots shows f on the (μ, w) plane with the logistic grid mapped back, and the logistic integrand on the unit square; the bottom row shows the same for the Laplace-adapted grid. A table lists $\log P(y)$, the error against the reference and the number of function evaluations for each method, and the effective number of grid points for each midpoint rule. Moving the data location to -5 or -10 makes the logistic integrand a thin spike near the edge of the square, and the logistic rule fails however many points are used. The Laplace-adapted integrand stays nearly flat in the middle of the square, so its error stays close to that of the Laplace approximation wherever the data sit. Its error shrinks only slowly as k grows, because the posterior of (μ, w) has somewhat heavier tails than its Laplace normal approximation and the integrand rises towards the edges of the square.

9.2.5 Example: Choosing the Number of Predictors in Logistic Regression

The marginal likelihood can compare models of different dimension, because integrating over β automatically penalizes parameters the data do not need. This example uses `log_mar_adapted_mv` to choose the number of predictors in a logistic regression.

Data. $n = 200$ observations with 20 candidate predictors $x_{i1}, \dots, x_{i,20} \overset{iid}{\sim} N(0,1)$ and

$$y_i \mid x_i \sim \text{Bernoulli}(\,\text{logit}^{-1}(x_{i1} - x_{i2})), \tag{9.46}$$

so only the first two predictors carry signal; there is no intercept.

Models. Model M_p uses the first p predictors, for $p = 1, 2, 3, 4, 5, 10, 20$, so the models are nested and M_2 is the true one. With $\eta_i = \sum_{j=1}^{p} x_{ij}\beta_j$, the likelihood and prior are

$$L_p(\beta) = \prod_{i=1}^{n} \frac{e^{y_i \eta_i}}{1 + e^{\eta_i}}, \qquad \beta_1, \dots, \beta_p \overset{iid}{\sim} N(0, \sigma_\beta^2), \tag{9.47}$$

$f_p(\beta) = L_p(\beta)\,\pi(\beta)$, and the target is the p-dimensional integral

$$P(y \mid M_p) = \int_{\mathbb{R}^p} f_p(\beta)\, d\beta. \tag{9.48}$$

Computation. Each $P(y \mid M_p)$ is computed by the Laplace-adapted product midpoint rule of Section 9.2.3. Because the grid has k^p points, k must shrink as p grows: $k = 10$ for $p \le 3$, then 6, 5, 3 and 2 for $p = 4, 5, 10, 20$; the largest grid, $2^{20} \approx 10^6$ points, is evaluated in blocks with matrix products. With $k = 1$ the only point is $u = (\frac{1}{2}, \dots, \frac{1}{2})$, that is $z = 0$ and $\beta = \hat{\beta}$, and the rule reduces exactly to the Laplace approximation, which is shown for comparison. Two prior standard deviations are compared, $\sigma_\beta = 1$ and $\sigma_\beta = 10$.

```
### data: 20 candidate predictors, only x1 and x2 carry signal
set.seed(2024)
n_lr <- 200; p_max <- 20
X_lr <- matrix(rnorm(n_lr * p_max), n_lr, p_max)
y_lr <- rbinom(n_lr, 1, plogis(drop(X_lr %*% c(1, -1, rep(0, p_max - 2)))))

### log P(y | M_p) by the Laplace-adapted product midpoint rule, k points per axis
log_ml_logit <- function(p, sigma_beta, k) {
  Xp <- X_lr[, seq_len(p), drop = FALSE]
  log_f <- function(b) {
    eta <- drop(Xp %*% b)
    sum(y_lr * eta - log1p(exp(eta))) + sum(dnorm(b, 0, sigma_beta, log = TRUE))
  }
  log_f_rows <- function(B) {                       # one row of B per grid point, in blocks
    out <- numeric(nrow(B))
    for (I in split(seq_len(nrow(B)), ceiling(seq_len(nrow(B)) / 2e4))) {
      Bi <- B[I, , drop = FALSE]
      E  <- Xp %*% t(Bi)                            # n x |I| linear predictors
      out[I] <- colSums(y_lr * E - log1p(exp(E))) + rowSums(dnorm(Bi, 0, sigma_beta, log =
↪  TRUE))
    }
    out
  }
  log_mar_adapted_mv(log_f, rep(0, p), k, log_f_rows)
}
```

```r
p_vals  <- c(1, 2, 3, 4, 5, 10, 20)
k_vals  <- c(10, 10, 10, 6, 5, 3, 2)              # k^p kept at most about 10^6
sb_vals <- c(1, 10)
logml_adapt <- sapply(sb_vals, function(sb) mapply(function(p, k) log_ml_logit(p, sb, k),
 ↪  p_vals, k_vals))
logml_lap   <- sapply(sb_vals, function(sb) sapply(p_vals, function(p) log_ml_logit(p, sb, 1)))
dimnames(logml_adapt) <- dimnames(logml_lap) <- list(p = p_vals, sigma_beta = sb_vals)

### BIC = -2 log L(beta_mle) + p log n, AIC = -2 log L(beta_mle) + 2 p, and the training
### loss -2 log L(beta_mle), all on the same scale as -2 log P(y | M_p); none uses the prior
loglik_mle <- sapply(p_vals, function(p)
  as.numeric(logLik(glm(y_lr ~ X_lr[, seq_len(p), drop = FALSE] - 1, family = binomial))))
ic <- cbind(bic = -2 * loglik_mle + p_vals * log(n_lr),
            aic = -2 * loglik_mle + 2 * p_vals,
            dev = -2 * loglik_mle)
```

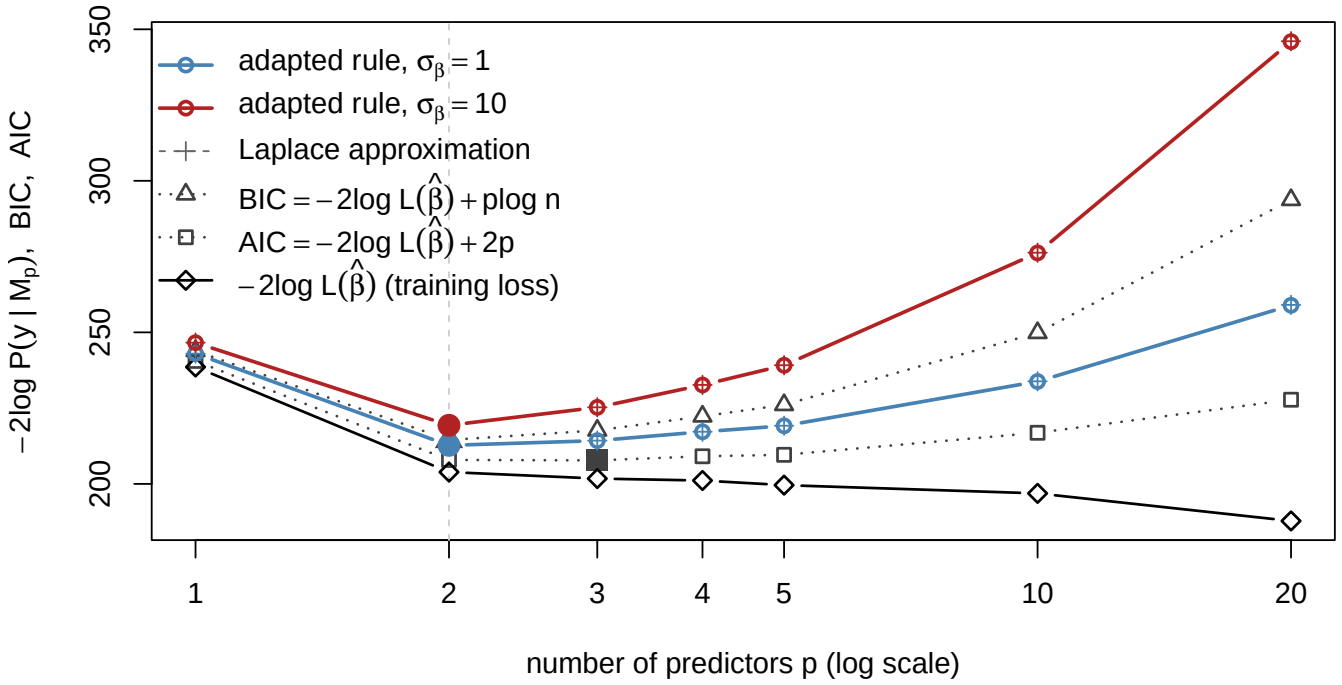


Figure 9.5: Shows that the marginal likelihood picks the true number of predictors in a logistic regression, for two prior scales, and compares it with BIC, AIC and the training loss. $-2\log P(y \mid M_p)$ for models using the first p of 20 predictors ($n = 200$; only x_1 and x_2 carry signal), with prior $\beta_j \sim N(0, \sigma_\beta^2)$ for $\sigma_\beta = 1$ and 10. Solid coloured lines with circles: the Laplace-adapted product midpoint rule; dashed coloured lines with plus signs: the Laplace approximation alone (the same rule with $k = 1$). The two differ by at most about 0.14, so each plus sign sits inside its circle. Dotted gray lines: $\text{BIC} = -2\log L(\hat\beta) + p\log n$ (triangles) and $\text{AIC} = -2\log L(\hat\beta) + 2p$ (squares), computed from the maximum likelihood fit and the same for both priors. Black line with diamonds: the training loss $-2\log L(\hat\beta)$ with no penalty, which keeps decreasing as predictors are added and so always favours the largest model. On every other curve the filled symbol marks the minimum. Both marginal-likelihood curves and BIC are smallest at $p = 2$, and the marginal likelihood rises faster for the more diffuse prior. AIC, with its lighter penalty of 2 per parameter, is smallest at $p = 3$, but only by 0.14 (207.76 against 207.90 at $p = 2$).

```
knitr::kable(data.frame(p = p_vals, k = k_vals, points = k_vals^p_vals,
                        adapted_1 = logml_adapt[, 1], laplace_1 = logml_lap[, 1],
                        adapted_10 = logml_adapt[, 2], laplace_10 = logml_lap[, 2]),
             digits = 3, format.args = list(big.mark = ","),
             col.names = c("p", "k", "grid points",
                           "adapted, _ = 1", "Laplace, _ = 1",
                           "adapted, _ = 10", "Laplace, _ = 10"))
```

Table 9.5: log P(y | M_p) for the logistic regression models in the figure above: the Laplace-adapted product midpoint rule (k points per axis, k^p grid points) and the Laplace approximation (the same rule with k = 1).

	p	k	grid points	adapted, σ_β = 1	Laplace, σ_β = 1	adapted, σ_β = 10	Laplace, σ_β = 10
1	1	10	10	-121.496	-121.496	-123.294	-123.295
2	2	10	100	-106.355	-106.356	-109.666	-109.668
3	3	10	1,000	-107.134	-107.139	-112.624	-112.630
4	4	6	1,296	-108.595	-108.603	-116.316	-116.324
5	5	5	3,125	-109.582	-109.594	-119.567	-119.579
10	10	3	59,049	-116.894	-116.924	-138.108	-138.141
20	20	2	1,048,576	-129.465	-129.527	-172.961	-173.029

On the $-2\log P(y \mid M_p)$ scale of the figure, smaller is better. Both marginal-likelihood curves drop sharply from $p = 1$ to $p = 2$, because the model with only x_1 misses a real effect and fits the data much worse. Both reach their minimum at the true $p = 2$. They then rise as useless predictors are added: each extra coefficient improves the fit only slightly, while the prior spreads its probability over a range of β_j the data do not need. The rise is steeper for $\sigma_\beta = 10$, because a more diffuse prior wastes more probability on implausible values. The choice of prior variance changes how strongly extra parameters are penalized, so it matters for model comparison in a way it rarely does for estimation. For this posterior the adapted rule and the Laplace approximation agree to within about 0.07 even at $p = 20$, so the ranking of the models is the same under either method. That agreement does not mean both are accurate: at $p = 20$ the grid has only $k = 2$ points per axis, too few to correct the Laplace approximation, and importance sampling in Section 11.7 shows that both are too low by about 0.6 there. The error is still far smaller than the differences between models.

The figure also shows three quantities that need no prior and no integration, all computed from the maximum likelihood estimate $\hat{\beta}$:

$$-2\log L_p(\hat{\beta}), \qquad \mathrm{BIC}_p = -2\log L_p(\hat{\beta}) + p\log n, \qquad \mathrm{AIC}_p = -2\log L_p(\hat{\beta}) + 2p. \tag{9.49}$$

The training loss $-2\log L_p(\hat{\beta})$ can only decrease as predictors are added, because each model contains the one before it; on its own it always picks $p = 20$. BIC is the large-n simplification of the Laplace approximation (Section 9.3.1): it keeps the training loss and the $p\log n$ term and drops everything of order one, including the prior, so its curve has the same shape as $-2\log P(y \mid M_p)$, lies between the two prior scales, and also picks $p = 2$. AIC charges only 2 per parameter instead of $\log n \approx 5.3$, so its curve rises slowly; here it narrowly prefers $p = 3$, the usual tendency of AIC to admit a spurious predictor when the extra fit is small. AIC targets prediction rather than the probability of the model, so the two criteria answer different questions.

9.3 Remarks on the Laplace Approximation

9.3.1 BIC as a large-n simplification

With $f(\theta) = L(\theta; D)\pi(\theta)$, the Laplace approximation to the log marginal likelihood is

$$\log P(D) \approx \log\left[L(\theta_0; D)\,\pi(\theta_0)\right] + \frac{p}{2}\log(2\pi) - \frac{1}{2}\log|H|. \tag{9.50}$$

For large n the Hessian is dominated by the likelihood, $H \approx n\, I(\theta_0)$ with I the per-observation Fisher information, so $|H| \approx n^p |I(\theta_0)|$ and

$$\log P(D) \approx \log L(\theta_0; D) - \frac{p}{2}\log n + \underbrace{\log \pi(\theta_0) + \frac{p}{2}\log(2\pi) - \frac{1}{2}\log|I(\theta_0)|}_{O(1),\ \text{dropped}}. \tag{9.51}$$

Multiplying by -2 and replacing the posterior mode by the MLE (the two coincide to $O(1/n)$) gives the **Bayesian information criterion**, $\text{BIC} = -2\log L(\hat{\theta}_{\text{MLE}}) + p\log n$ — a model-comparison score that needs no prior at all, because the dropped $O(1)$ terms do not grow with n and so wash out of *differences* between models fit to the same data. For the $n = 20$ Poisson example from the credible-interval check above, $-2 \times$ (Laplace $\log P(y)$) and the MLE-based BIC differ by about 1.68 — a constant offset consistent with the dropped $O(1)$ terms, not something that grows with n.

9.3.2 When Laplace Works Well, and When It Does Not

Works well when n is large relative to p and the posterior is unimodal and roughly symmetric, as in the $n = 100$ panel above. Parameters should live on an unbounded scale — transform first ($\log\sigma$, $\text{logit}\, p$) so the matching Gaussian is not truncated at a boundary, remembering to include the Jacobian of the transform in f (exactly the $w = \log\sigma^2$ reparametrization used throughout this book since the multivariate optimization chapter).

Struggles when the posterior is skewed (small n, a parameter near a boundary) — the mode is then a poor stand-in for the mean and the Gaussian tails are wrong, as the $n = 5$ panel above already shows mildly; when the posterior is multimodal, since the approximation only sees the one mode an optimizer happened to find; and in hierarchical models where p grows with n, so the $O(1/n)$ error does not vanish (nested Laplace approximation, INLA, handles this by integrating out the high-dimensional part analytically).

9.4 Summary

- The Laplace approximation expands $h = -\log f$ to second order at its minimum θ_0 and integrates the resulting Gaussian exactly: $\theta \mid \text{data} \approx N_p(\theta_0, H^{-1})$ and $\int f(\theta)\, d\theta \approx f(\theta_0)(2\pi)^{p/2}|H|^{-1/2}$.
- It needs only an optimizer and a Hessian — the same machinery already used to find the MLE — and its cost does not grow with dimension the way a quadrature grid's does.
- BIC is the large-n simplification of the Laplace marginal likelihood, dropping $O(1)$ terms that do not depend on the sample size.
- On the running Gaussian marginal-likelihood example, Laplace tracked the true value everywhere it was tested, including where quadrature's fixed grid broke down and even where naive Monte Carlo's fixed proposal started to run short of relevant draws — because Laplace re-optimizes for every dataset instead of committing to one grid or one proposal in advance.
- It is deterministic, fast, and accurate to $O(1/n)$, but blind to skewness and multimodality; the Monte Carlo methods of the next chapters remove these limitations at higher computational cost.

10 Rejection Sampling

```
library(ars)
```

Monte Carlo integration estimates $E_f[a(\theta)] = \int a(\theta) f(\theta)\, d\theta$ from iid draws $\theta_1, \ldots, \theta_n \sim f$. The catch is the sampling step itself: R has built-in samplers for standard families, but a posterior $f(\theta) \propto L(\theta)\pi(\theta)$ is typically known only up to a constant, and there is no generic way to draw from it directly. **Rejection sampling** solves this by producing *exact* iid draws from f using draws from an easier distribution g, following the recipe *reject, reweight, repeat*.

10.1 The Envelope and the Algorithm

10.1.1 The envelope condition

Suppose we can find a function $g(x)$ such that

- $g(x) \geq f(x)$ for every x — an **envelope** of f, and
- we can sample from the normalized density $g(x)/A_g$, where $A_g = \int g(x)\, dx$.

Neither f nor g needs to be normalized: f may be an unnormalized posterior with unknown area $A_f = \int f(x)\, dx$, and g a scaled density $M g_0(x)$ for a sampler g_0 we already have. We will see that the acceptance rate of the algorithm is A_f/A_g. So the design problem is finding a g that is both easy to sample and close to f.

10.1.2 Uniform points under a curve

Rejection sampling rests on one fact about points spread uniformly under the graph of a function. For a function $h \geq 0$ with finite area $A_h = \int h(x)\, dx$, write the region under its graph as

$$S_h = \{(x, v) : 0 < v < h(x)\}. \tag{10.1}$$

The area of S_h is A_h, so a point (X, V) **uniform on** S_h has joint density $1/A_h$ on S_h and 0 elsewhere.

Theorem 10.1 (Uniform under a curve)**.** *Let $h \geq 0$ with $0 < A_h = \int h(x)\, dx < \infty$.*

(a) If (X, V) is uniform on S_h, then the marginal density of X is $h(x)/A_h$.

(b) Conversely, if X has density $h(x)/A_h$ and, given X, $V \sim$ Unif$(0, h(X))$, then (X, V) is uniform on S_h.

Proof.

(a) Integrate the joint density over v. For fixed x, the point (x, v) lies in S_h exactly when $0 < v < h(x)$, so

$$p_X(x) = \int_0^{h(x)} \frac{1}{A_h}\, dv = \frac{h(x)}{A_h}. \tag{10.2}$$

(b) The joint density is the marginal of X times the conditional of V given X:

$$p(x, v) = \frac{h(x)}{A_h} \cdot \frac{1}{h(x)}\, \mathbb{1}\{0 < v < h(x)\} = \frac{1}{A_h}\, \mathbb{1}\{(x, v) \in S_h\}, \tag{10.3}$$

which is the uniform density on S_h.

□

Part (a) says the density of X at x is proportional to the **height** of the region at x: a tall column holds more points than a short one. Part (b) gives a way to fill S_h uniformly using a sampler for h/A_h. Figure 10.1 illustrates both parts with the Laplace-shaped function $g(x) = e^{1/2-|x|}$, which has area $A_g = 2e^{1/2}$.

```
g_demo <- function(x) exp(0.5 - abs(x))
A_g <- 2 * exp(0.5)                       # area under g
xs <- seq(-6, 6, length.out = 500)
### sampler for the normalized g, a Laplace(0, 1) density, by inverse CDF
rlaplace <- function(n) { u <- runif(n); ifelse(u < 0.5, log(2 * u), -log(2 * (1 - u))) }

set.seed(2)
### part (a): uniform points in a box, kept if under g
nb <- 30000
xb <- runif(nb, -6, 6); vb <- runif(nb, 0, exp(0.5))
keep <- vb < g_demo(xb); xa <- xb[keep]; va <- vb[keep]
### part (b): X ~ g / A_g, then V | X ~ Unif(0, g(X))
nshow <- 800
xr <- rlaplace(nshow); vr <- runif(nshow) * g_demo(xr)

par(mfrow = c(1, 3), mar = c(4, 4, 2.2, 1))
plot(xs, g_demo(xs), type = "l", lwd = 3, col = "steelblue", ylim = c(0, 1.7),
     xlab = "x", ylab = "v", main = "(a) uniform on S_g, from a box")
points(xa[1:nshow], va[1:nshow], pch = 19, cex = 0.45, col = adjustcolor("black", 0.6))
hist(xa, breaks = seq(-6, 6, by = 0.3), freq = FALSE, col = "grey75", border = "white",
     xlab = "x", main = "(a) histogram of X vs g(x) / A_g")
lines(xs, g_demo(xs) / A_g, lwd = 3, col = "steelblue")
plot(xs, g_demo(xs), type = "l", lwd = 3, col = "steelblue", ylim = c(0, 1.7),
     xlab = "x", ylab = "v", main = "(b) X ~ g / A_g, V | X ~ Unif(0, g(X))")
points(xr, vr, pch = 19, cex = 0.45, col = adjustcolor("black", 0.6))
```

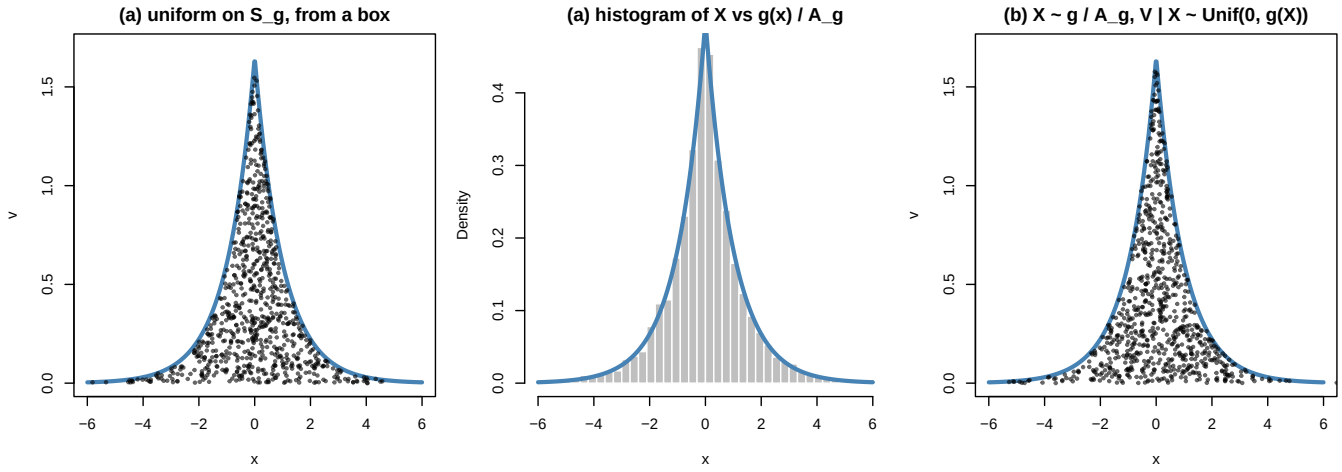


Figure 10.1: Illustrates the theorem that a point uniform under the graph of g has marginal density proportional to g, and conversely. Left: uniform points in the box $[-6, 6] \times [0, e^{1/2}]$, kept when they fall under $g(x) = e^{1/2-|x|}$, so they are uniform on S_g. Middle: the histogram of their x-coordinates matches $g(x)/A_g$, as in part (a). Right: points made the other way, by drawing X from g/A_g and then V uniform on $(0, g(X))$, as in part (b). They fill S_g in the same uniform way as the left panel.

10.1.3 The algorithm in two forms

Form 1 (ratio form). For each draw needed, repeat until acceptance:

1. Draw $X \sim g(x)/A_g$.
2. Draw $U \sim \text{Unif}(0, 1)$.
3. If $U < f(X)/g(X)$, accept X; otherwise return to step 1.

Form 2 (geometric form). For each draw needed, repeat until acceptance:

1. Draw $X \sim g(x)/A_g$.
2. Draw $V \sim \text{Unif}(0, g(X))$, e.g. $V = U\,g(X)$ with $U \sim \text{Unif}(0, 1)$.
3. If $V < f(X)$, i.e. the point (X, V) lies in S_f, accept X; otherwise return to step 1.

The two forms are the same algorithm, since with $V = U\,g(X)$,

$$V < f(X) \iff U < \frac{f(X)}{g(X)}. \tag{10.4}$$

Form 1 is how the sampler is usually coded. Form 2 explains why it works, as shown next. Both evaluate f and g only at the proposed point, so unknown normalizing constants in f (and in g) do not matter, as long as the functions actually used satisfy $g \geq f$.

10.1.4 Why the retained points have density f

By part (b) of Theorem 10.1, steps 1–2 of Form 2 make (X, V) uniform on S_g. Because $g \geq f$, the region S_f lies inside S_g. Step 3 keeps the point only when it lands in S_f. So for any set $B \subseteq S_f$,

$$P\{(X,V) \in B \mid \text{accept}\} = \frac{P\{(X,V) \in B\}}{P\{(X,V) \in S_f\}} = \frac{\text{area}(B)/A_g}{A_f/A_g} = \frac{\text{area}(B)}{A_f}. \tag{10.5}$$

This says a retained (X, V) is **uniform on** S_f. By part (a) of Theorem 10.1 with $h = f$, the retained X has density $f(x)/A_f$, the normalized target. The draw is exact, not approximate.

The same calculation gives the cost. Each proposal is accepted with probability

$$P(\text{accept}) = P\{(X,V) \in S_f\} = \frac{A_f}{A_g}, \tag{10.6}$$

the ratio of the two areas. Proposals are independent, so the number needed per accepted draw is geometric with mean A_g/A_f. A tight envelope wastes few proposals and a loose one wastes many.

Figure 10.2 shows the whole picture for the standard normal target $f(x) = e^{-x^2/2}$ under the same Laplace envelope g. Here $A_f = \sqrt{2\pi} \approx 2.507$ and $A_g = 2e^{1/2} \approx 3.297$, so about 76% of proposals are accepted.

```
f_demo <- function(x) exp(-x^2 / 2)
xs <- seq(-6, 6, length.out = 500)

par(mfrow = c(1, 2), mar = c(4, 4, 2, 1))
plot(xs, g_demo(xs), type = "n", ylim = c(0, 1.75), xlab = "x", ylab = "")
polygon(c(xs, rev(xs)), c(g_demo(xs), rep(0, 500)), col = adjustcolor("steelblue", 0.3), border
 ↪  = NA)
polygon(c(xs, rev(xs)), c(f_demo(xs), rep(0, 500)), col = adjustcolor("firebrick", 0.4), border
 ↪  = NA)
lines(xs, g_demo(xs), col = "steelblue", lwd = 3); lines(xs, f_demo(xs), col = "firebrick", lwd
 ↪  = 3)
legend("topright", bty = "n", fill = adjustcolor(c("steelblue", "firebrick"), 0.35), border =
 ↪  NA,
       legend = c("area under envelope g", "area under target f"))

set.seed(1)
N <- 600
xp <- rlaplace(N); vp <- runif(N) * g_demo(xp); acc <- vp < f_demo(xp)
plot(xs, g_demo(xs), type = "l", lwd = 3, col = "steelblue", ylim = c(0, 1.75), xlab = "x",
 ↪  ylab = "v")
lines(xs, f_demo(xs), lwd = 3, col = "firebrick")
points(xp[!acc], vp[!acc], pch = 4, col = adjustcolor("gray40", 0.7), cex = 0.7)
points(xp[acc], vp[acc], pch = 19, col = adjustcolor("firebrick", 0.7), cex = 0.7)
legend("topright", bty = "n", pch = c(19, 4), col = c("firebrick", "gray40"),
       legend = c(sprintf("accepted (%d)", sum(acc)), sprintf("rejected (%d)", sum(!acc))))
```

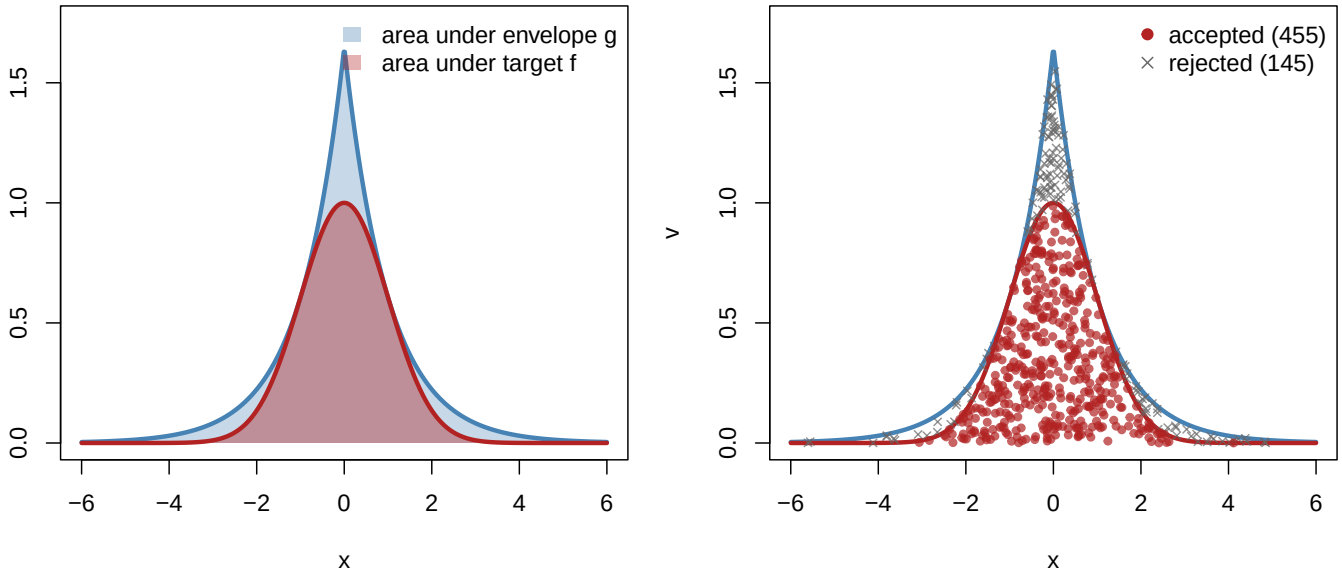


Figure 10.2: Illustrates the rejection sampling algorithm: candidate points are drawn uniformly under an envelope and kept when they fall under the target density. Left: target $f(x) = e^{-x^2/2}$ (unnormalized standard normal) under its Laplace envelope $g(x) = e^{1/2-|x|}$, with the two areas shaded. Right: 600 candidate points drawn uniformly under g; the accepted ones (red, under f too) are uniformly distributed under f, so their x-coordinates are draws from the normalized f.

10.2 Example 1: Sampling a Normal from a Laplace Envelope

The figure above already uses a concrete, closed-form example worth stating in full, because it is simple enough to check every claim exactly.

Target. The unnormalized standard normal density

$$f(x) = e^{-x^2/2}, \qquad x \in \mathbb{R}, \qquad A_f = \sqrt{2\pi}. \tag{10.7}$$

Envelope. A scaled Laplace (double-exponential) density

$$g(x) = e^{1/2-|x|} = 2e^{1/2}\, g_0(x), \qquad g_0(x) = \tfrac{1}{2}e^{-|x|}, \qquad A_g = 2e^{1/2}. \tag{10.8}$$

The normalized envelope g_0 is the standard Laplace density, which is sampled by inversion (`rlaplace` above).

Proof. $g \geq f$ **for every** x**.** Compare the logs:

$$\log g(x) - \log f(x) = \tfrac{1}{2} - |x| + \tfrac{x^2}{2} = \tfrac{1}{2}(|x|^2 - 2|x| + 1) = \tfrac{1}{2}(|x| - 1)^2 \;\geq\; 0, \tag{10.9}$$

with equality exactly at $|x| = 1$, where the two curves in the figure touch. The constant $e^{1/2}$ is the smallest that works: with $e^{a-|x|}$ the gap is $\tfrac{1}{2}(|x|-1)^2 + a - \tfrac{1}{2}$, which is negative at $|x| = 1$ when $a < \tfrac{1}{2}$. □

The exact acceptance rate is

$$P(\text{accept}) = \frac{\int f(x)\,dx}{\int g(x)\,dx} = \frac{\sqrt{2\pi}}{2e^{1/2}} \approx 0.760. \tag{10.10}$$

```
rnorm_reject <- function(n) {
    out <- numeric(0); tries <- 0
    while (length(out) < n) {
        m <- n - length(out)
        x <- rlaplace(m); tries <- tries + m
        out <- c(out, x[runif(m) < f_demo(x) / g_demo(x)])
    }
    list(x = out, acceptance = n / tries)
}
set.seed(2)
res <- rnorm_reject(2e5)
knitr::kable(data.frame(
    quantity = c("acceptance rate", "sample mean", "sample sd"),
    theoretical = c(sqrt(2 * pi) / (2 * sqrt(exp(1))), 0, 1),
    simulated = c(res$acceptance, mean(res$x), sd(res$x))
), digits = 4)
```

Table 10.1: Verifies the rejection sampler for the normal distribution by comparing its acceptance rate and sample moments with theory. Theoretical vs. simulated acceptance rate and sample moments for the Laplace-envelope normal sampler.

quantity	theoretical	simulated
acceptance rate	0.7602	0.7595
sample mean	0.0000	0.0031
sample sd	1.0000	0.9991

Roughly 3 in 4 proposals are kept — a tight, efficient envelope, and the benchmark against which the next example's Cauchy envelope should be judged.

A shinylive app shows how the tightness of the envelope determines the efficiency of rejection sampling: a looser envelope still gives exact draws but rejects more of them.

10.3 Example 2: Sampling a Gamma Distribution

The Laplace envelope above was designed to match a normal target. A different target needs a different envelope. Here is one built for Gamma$(\alpha, 1)$ with $\alpha > 1$. Write $m = \alpha - 1 > 0$ for the mode and $c = 2\alpha - 1 = 2m + 1$.

Target. The unnormalized Gamma density (the kernel, without $1/\Gamma(\alpha)$)

$$f(x) = x^{\alpha-1}e^{-x} = x^m e^{-x}, \qquad x > 0, \tag{10.11}$$

with area $A_f = \Gamma(\alpha)$ and maximum $f(m) = m^m e^{-m}$ at the mode.

Envelope. A Cauchy-shaped curve centred at the mode, scaled to touch f there:

$$g(x) = \frac{m^m e^{-m}}{1 + (x-m)^2/c}, \qquad x \in \mathbb{R}. \tag{10.12}$$

Normalized, g is the density of $m + \sqrt{c}\,C$ with C standard Cauchy, so sampling g needs only `rcauchy`. Its area is $A_g = m^m e^{-m}\,\pi\sqrt{c}$. The heavy Cauchy tails, decaying like x^{-2}, are what let g stay above the exponentially decaying f for large x. A normal-shaped envelope would eventually fall below it.

Proof. $g \geq f$ **for every** $x > 0$**.** For $x \leq 0$ there is nothing to show, since $f = 0$ there. For $x > 0$ consider the log gap

$$D(x) = \log g(x) - \log f(x) = m \log m - m - \log\Big(1 + \frac{(x-m)^2}{c}\Big) - m \log x + x. \tag{10.13}$$

Clearly $D(m) = 0$. Differentiating,

$$D'(x) = 1 - \frac{m}{x} - \frac{2(x-m)}{c + (x-m)^2} = (x-m)\left[\frac{1}{x} - \frac{2}{c + (x-m)^2}\right] = \frac{(x-m)\left[c + (x-m)^2 - 2x\right]}{x\left(c + (x-m)^2\right)}. \tag{10.14}$$

With $c = 2m + 1$ the bracket is a perfect square: $c + (x-m)^2 - 2x = (x-m)^2 - 2(x-m) + 1 = (x-m-1)^2 \geq 0$. So

$$D'(x) = \frac{(x-m)(x-m-1)^2}{x\left(c + (x-m)^2\right)}, \tag{10.15}$$

which has the sign of $x - m$. Hence D decreases on $(0, m)$ and increases on (m, ∞), its minimum is $D(m) = 0$, and $D(x) \geq 0$, i.e. $g(x) \geq f(x)$, for all $x > 0$. This also shows where the scale $c = 2\alpha - 1$ comes from: it is the value that turns the bracket into a square. □

The acceptance rate is therefore

$$P(\text{accept}) = \frac{A_f}{A_g} = \frac{\Gamma(\alpha)}{\pi\sqrt{2\alpha - 1}\,(\alpha-1)^{\alpha-1} e^{-(\alpha-1)}}. \tag{10.16}$$

It is $e/(\pi\sqrt{3}) \approx 0.50$ at $\alpha = 2$, and by Stirling's formula it rises to $1/\sqrt{\pi} \approx 0.564$ as $\alpha \to \infty$.

```
### log Gamma kernel f(x) = x^(alpha-1) exp(-x), without 1/Gamma(alpha)
log_f_gamma <- function(x, alpha) (alpha - 1) * log(x) - x
### log of the Cauchy-shaped envelope g, which touches f at the mode alpha - 1
log_g_gamma <- function(x, alpha) {
    (alpha - 1) * (log(alpha - 1) - 1) - log(1 + (x - (alpha - 1))^2 / (2 * alpha - 1))
}
```

10.3.1 Checking the envelope numerically

The proof uses the **unnormalized** f, and this matters. Up to a constant, `log_g_gamma` is the log-density of $m + \sqrt{c}\,C$:

```
x_check <- seq(0.1, 10, by = 0.37)
alpha_check <- 3.3
log_f_Y <- dcauchy((x_check - (alpha_check - 1)) / sqrt(2 * alpha_check - 1), log = TRUE) -
    log(sqrt(2 * alpha_check - 1))
range(log_g_gamma(x_check, alpha_check) - log_f_Y)   # constant across x confirms the claim
```

```
[1] 1.621804 1.621804
```

Table 10.2 computes the minimum gap $\min_x\{\log g(x) - \log f(x)\}$ on a fine grid, both for the kernel f used in the proof and for the normalized density `dgamma` $= f/\Gamma(\alpha)$.

```
alphas_check <- c(1.1, 1.5, 1.9, 2.0, 2.5, 3, 4.5, 10)
xv <- seq(1e-6, 60, length.out = 2e4)
gap_kernel <- vapply(alphas_check, function(a) min(log_g_gamma(xv, a) - log_f_gamma(xv, a)),
 ↪  numeric(1))
gap_dgamma <- vapply(alphas_check, function(a) min(log_g_gamma(xv, a) - dgamma(xv, a, log =
 ↪  TRUE)), numeric(1))
knitr::kable(data.frame(alpha = alphas_check, gap_kernel = gap_kernel, gap_dgamma =
 ↪  gap_dgamma),
             digits = 5, col.names = c("alpha", "min gap vs kernel f", "min gap vs dgamma"))
```

Table 10.2: Checks that the Cauchy envelope lies above the Gamma target, and shows that this depends on which version of the target is used. Minimum of log g(x) - log f(x) over x > 0, for several alpha. Negative means the envelope is violated somewhere. Against the kernel x^(alpha-1) exp(-x) the gap is never negative, as proved. Against the normalized density it is -log Gamma(alpha), which is negative for 1 < alpha < 2 because Gamma(alpha) < 1 there.

alpha	min gap vs kernel f	min gap vs dgamma
1.1	0	-0.04987
1.5	0	-0.12078
1.9	0	-0.03898
2.0	0	0.00000
2.5	0	0.28468
3.0	0	0.69315
4.5	0	2.45374
10.0	0	12.80183

The gap against the kernel is 0 for every α, attained at the mode, exactly as the proof says. Against `dgamma` it equals $-\log\Gamma(\alpha)$. That is negative for $1 < \alpha < 2$, so g would *not* cover the normalized density there. For $\alpha > 2$ it is positive, so g still covers it but with a wasted factor $\Gamma(\alpha)$, and the acceptance rate drops to $1/A_g$ instead of $\Gamma(\alpha)/A_g$. The lesson is the one stated with the algorithm: the inequality $g \geq f$ must hold for the functions *actually used* in the ratio f/g. Here the right choice is the kernel, which needs no $\Gamma(\alpha)$ at all.

10.3.2 Sampling function

A batch-and-refill implementation is both clearer and much faster than drawing one candidate at a time in an R loop: generate a vector of candidates, vectorize the accept/reject test, keep the survivors, and top up only the shortfall.

```
sample_gamma_rej <- function(n, alpha, max_tries = 2e6) {
    out <- numeric(0); tries <- 0
    while (length(out) < n) {
        if (tries > max_tries) stop("acceptance rate too low: envelope is too loose")
        m <- n - length(out)
        x <- rcauchy(m) * sqrt(2 * alpha - 1) + (alpha - 1)
        tries <- tries + m
        x <- x[x > 0]                              # f = 0 for x <= 0: always rejected
        keep <- log(runif(length(x))) < log_f_gamma(x, alpha) - log_g_gamma(x, alpha)
        out <- c(out, x[keep])
    }
    structure(out[seq_len(n)], accept.rate = n / tries)
}
```

10.3.3 Test cases and efficiency

```
acc_theory <- function(a) exp(lgamma(a) - log(pi) - 0.5 * log(2 * a - 1) - (a - 1) * (log(a -
↪  1) - 1))
set.seed(3)
alphas_test <- c(1.2, 2.5, 4.5, 20)
gamma_tests <- lapply(alphas_test, function(a) sample_gamma_rej(2000, a))
knitr::kable(data.frame(
    alpha = alphas_test,
    accept_theory = acc_theory(alphas_test),
    accept_rate = vapply(gamma_tests, attr, numeric(1), "accept.rate"),
    ks_stat = mapply(function(x, a) suppressWarnings(ks.test(x, "pgamma", a))$statistic,
                     gamma_tests, alphas_test)
), digits = 4, col.names = c("alpha", "exact acceptance", "simulated acceptance",
                             "KS statistic vs true Gamma"))
```

Table 10.3: Shows how the efficiency of the Gamma rejection sampler, measured by its acceptance rate, changes with the shape parameter. The simulated acceptance rate matches the exact rate A_f / A_g. It rises slowly with alpha towards 1/sqrt(pi) = 0.564, and the KS statistics show no departure from the Gamma distribution.

alpha	exact acceptance	simulated acceptance	KS statistic vs true Gamma
1.2	0.4163	0.4195	0.0201
2.5	0.5161	0.5315	0.0160
4.5	0.5404	0.5414	0.0150
20.0	0.5594	0.5534	0.0150

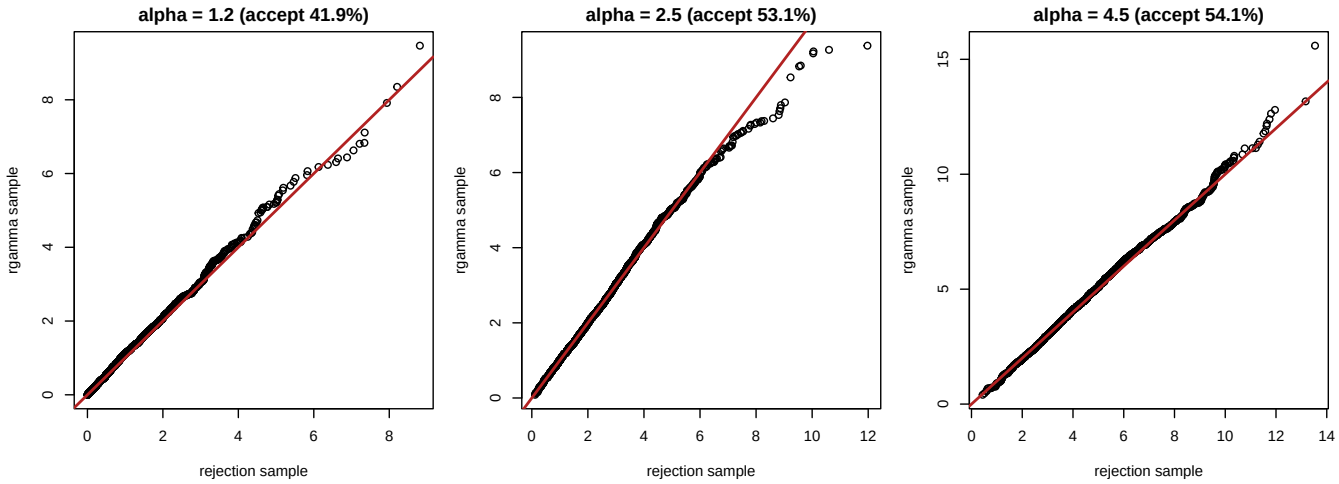


Figure 10.3: Verifies that the Gamma rejection sampler produces draws from the correct distribution. Q-Q plots of rejection-sampled draws against the true Gamma(alpha,1) quantiles, for three of the test cases: the envelope produces exact draws regardless of alpha, only the cost per draw changes.

The Q-Q plots confirm exactness at every α: rejection sampling never trades accuracy for speed, only speed changes. About half of all proposals are rejected for any α, so this envelope is less efficient than the Laplace envelope of Example 1. Efficient random-number generators for the Gamma distribution (as used inside `rgamma` itself) use envelopes that are much tighter; see Marsaglia & Tsang (2000, *ACM TOMS*) for the algorithm R actually uses.

10.3.4 Shinylive App for Rejection Sampling from a Gamma Density with a Cauchy Envelope

There is a shinylive app to show how the efficiency of rejection sampling for the Gamma distribution changes with the shape parameter while the draws remain exact.

10.4 Adaptive Rejection Sampling for Log-Concave Targets

Designing an envelope by hand, as above, is the hard part of rejection sampling. For a broad and important class of targets it can be automated.

10.4.1 Log-concavity and the tangent hull

A density f is **log-concave** if $\log f(x)$ is concave. Many posteriors are: the normal, the gamma with shape ≥ 1, the beta with both parameters ≥ 1, logistic-regression posteriors under normal priors, and any product of log-concave factors (so most conjugate and many non-conjugate exponential- family posteriors qualify).

A concave function lies below every one of its tangent lines. So for tangent lines $\ell_1, \dots, \ell_k$ to $\log f$ at points $x_1 < \cdots < x_k$, the piecewise-linear **upper hull**

$$\log g(x) = \min_j \ell_j(x) \tag{10.17}$$

satisfies $\log g \geq \log f$ everywhere, hence $g = e^{\log g} \geq f$: a valid envelope, built automatically from a handful of tangent points and their slopes (the log-density's derivative), with no hand-crafted formula. Because g is piecewise-exponential, both its normalizing integral and its inverse CDF are closed-form on each piece, so sampling from g is cheap.

Proof. **The tangent-line inequality.** Let $h = \log f$ be concave and differentiable on an interval, and let x_j and x be two points in it. For $t \in (0, 1]$, concavity gives $h(x_j + t(x - x_j)) \geq (1 - t)\, h(x_j) + t\, h(x)$, which rearranges to

$$\frac{h(x_j + t(x - x_j)) - h(x_j)}{t} \;\geq\; h(x) - h(x_j). \tag{10.18}$$

Letting $t \to 0$, the left side tends to $h'(x_j)(x - x_j)$, so $h(x) \leq h(x_j) + h'(x_j)(x - x_j) = \ell_j(x)$. This holds for every j, so $h(x) \leq \min_j \ell_j(x) = \log g(x)$ and $f \leq g$. □

10.4.2 The adaptive algorithm

Adaptive rejection sampling (Gilks & Wild, 1992) refines the hull as it goes:

1. Start with a few tangent points, at least one on each side of the mode.
2. Draw $x \sim g$, $U \sim \text{Unif}(0, 1)$; accept if $U < f(x)/g(x)$.
3. Whenever a point is **rejected**, add it to the tangent set and rebuild the hull.

Each rejection adds a tangent exactly where the envelope was loose, so the acceptance rate rises toward 1 as sampling proceeds — the envelope adapts itself to the target instead of being fixed in advance. Only $\log f$ and its derivative are required (a derivative-free variant uses secants in place of tangents). The `ars` package implements this for any user-supplied log-concave `f`.

10.5 Example 3: Truncated Normal by Adaptive Rejection Sampling

The standard normal is log-concave ($\log f(x) = -x^2/2$ is concave), and truncating to $[l, u]$ only shrinks the domain — it cannot break concavity — so a truncated normal is log-concave too, and `ars` applies to it directly. Explicitly, for $l < u$ (either may be infinite):

Target.

$$f(x) = e^{-x^2/2}\, \mathbb{1}\{l < x < u\}, \qquad h(x) = \log f(x) = -\tfrac{x^2}{2} \;\text{ on } (l, u). \tag{10.19}$$

Envelope (ARS). With tangent points $x_1 < \cdots < x_k$ in $[l, u]$, the tangent at x_j is

$$\ell_j(x) = -\tfrac{x_j^2}{2} - x_j(x - x_j) = \tfrac{x_j^2}{2} - x_j x. \tag{10.20}$$

Adjacent tangents ℓ_j and ℓ_{j+1} cross where $\frac{x_j^2}{2} - x_j z = \frac{x_{j+1}^2}{2} - x_{j+1} z$, that is, at the midpoint

$$z_j = \frac{x_j + x_{j+1}}{2}, \qquad j = 1, \ldots, k - 1. \tag{10.21}$$

The envelope uses tangent j between z_{j-1} and z_j, with $z_0 = l$ and $z_k = u$:

$$g(x) = \begin{cases} \exp\{\frac{x_1^2}{2} - x_1 x\}, & l < x \le z_1, \\ \exp\{\frac{x_2^2}{2} - x_2 x\}, & z_1 < x \le z_2, \\ \quad\vdots & \quad\vdots \\ \exp\{\frac{x_k^2}{2} - x_k x\}, & z_{k-1} < x < u. \end{cases} \tag{10.22}$$

Each piece is an exponential function of x, so g is easy to integrate and to sample piece by piece.

Proof. $g \ge f$**.** On the piece $z_{j-1} < x \le z_j$, $\log g(x) = \ell_j(x)$, and

$$\ell_j(x) - h(x) = \tfrac{x_j^2}{2} - x_j x + \tfrac{x^2}{2} = \tfrac{1}{2}(x - x_j)^2 \;\ge\; 0. \tag{10.23}$$

So $\log g \ge h$ on every piece, i.e. $g \ge f$ on (l, u). This is the tangent-line inequality above, verified directly. (This g is the same as the hull $\min_j \ell_j$: since $\ell_j(x) - \ell_{j+1}(x) = (x_{j+1} - x_j)(x - z_j)$, tangent j lies below tangent $j+1$ exactly when $x \le z_j$, so on each piece the tangent in use is the lowest one.) □

This makes it a clean test case for comparing ARS against the most obvious alternative: sample the *untruncated* $N(0,1)$ and throw away anything outside $[l, u]$. That naive sampler is itself a (trivial) rejection sampler, with envelope $g(x) = e^{-x^2/2}$ on all of $\mathbb{R}$. Here $g \ge f$ because $f = g \cdot \mathbb{1}\{l < x < u\}$, and the ratio f/g is 1 inside (l, u) and 0 outside, so "accept if $U < f/g$" just means "keep x if it lands in (l, u)". It is exactly correct — but its acceptance rate is just $P(l < Z < u)$, which collapses to essentially 0 once $[l, u]$ sits out in the tail. ARS has no such weakness: its hull is built from the *truncated* target from the start, so it never proposes a point outside $[l, u]$ in the first place, and its acceptance rate stays high regardless of where the window sits.

```
### direct rejection sampling for a truncated normal: draw from N(0,1),
### keep values inside [lb, ub]; vectorized batch-and-refill
sample_tnorm_drs <- function(n, lb = -Inf, ub = Inf) {
    out <- numeric(0)
    while (length(out) < n) {
        x <- rnorm(n - length(out))
        out <- c(out, x[x >= lb & x <= ub])
    }
    out
}

### adaptive rejection sampling via the ars package
sample_tnorm_ars <- function(n, lb, ub) {
    logf <- function(x) dnorm(x, log = TRUE)
    fprima <- function(x) -x
    ars(n, f = logf, fprima = fprima, x = c(lb, (lb + ub) / 2, ub),
        lb = TRUE, ub = TRUE, xlb = lb, xub = ub)
}
```

```
set.seed(4)
n <- 1000
windows <- list(c(-5, -4), c(-50, -40), c(100, 110))
timing <- lapply(windows, function(w) {
    t_ars <- system.time(sample_tnorm_ars(n, w[1], w[2]))["elapsed"]
    t_naive <- if (w[1] > -10 && w[2] < 10) system.time(sample_tnorm_drs(n, w[1],
 ↪  w[2]))["elapsed"] else NA
    c(lb = w[1], ub = w[2], ars_seconds = t_ars, naive_seconds = t_naive)
})
knitr::kable(do.call(rbind, timing), digits = 4,
             col.names = c("lower bound", "upper bound", "ars (s)", "naive rejection (s)"))
```

Table 10.4: Compares the cost of adaptive rejection sampling with that of naive rejection sampling for truncated normal distributions. Adaptive rejection sampling (ars) vs naive rejection sampling from an untruncated N(0,1): naive sampling degrades catastrophically as the truncation window moves into the tail, while ars does not, because it never wastes draws outside [lb, ub] in the first place.

lower bound	upper bound	ars (s)	naive rejection (s)
-5	-4	0.002	1.063
-50	-40	0.005	NA
100	110	0.002	NA

For $[-50, -40]$ or $[100, 110]$ the naive sampler's acceptance probability is astronomically small ($P(-50 < Z < -40) \approx 10^{-221}$), so it is left as `NA` rather than run to timeout; `ars` samples these windows in a fraction of a second because its envelope is built from the truncated density itself, never from the untruncated normal.

```
### log(pnorm(ub) - pnorm(lb)), numerically stable even when the naive
### difference underflows to 0 for windows far out in either tail
log_pnorm_diff <- function(lb, ub) {
    if (ub <= 0) {
        log_pu <- pnorm(ub, log.p = TRUE); log_pl <- pnorm(lb, log.p = TRUE)
        log_pu + log1p(-exp(log_pl - log_pu))
    } else if (lb >= 0) {
        log_su <- pnorm(ub, lower.tail = FALSE, log.p = TRUE)
        log_sl <- pnorm(lb, lower.tail = FALSE, log.p = TRUE)
        log_sl + log1p(-exp(log_su - log_sl))
    } else {
        log(pnorm(ub) - pnorm(lb))
    }
}
tnorm_density <- function(x, lb, ub) exp(dnorm(x, log = TRUE) - log_pnorm_diff(lb, ub))

par(mfrow = c(1, 3), mar = c(4, 4, 2, 1))
set.seed(5)
for (w in windows) {
    x <- sample_tnorm_ars(n, w[1], w[2])
    hist(x, breaks = 25, freq = FALSE, col = "steelblue", border = "white",
```

```
        main = sprintf("[%g, %g]", w[1], w[2]), xlab = "x")
    curve(tnorm_density(x, w[1], w[2]), add = TRUE, lwd = 3, col = "firebrick", n = 300)
    if (identical(w, windows[[1]])) {
        legend("topright", bty = "n", lwd = 3, col = "firebrick", cex = 0.8,
               legend = "true density")
    }
}
```

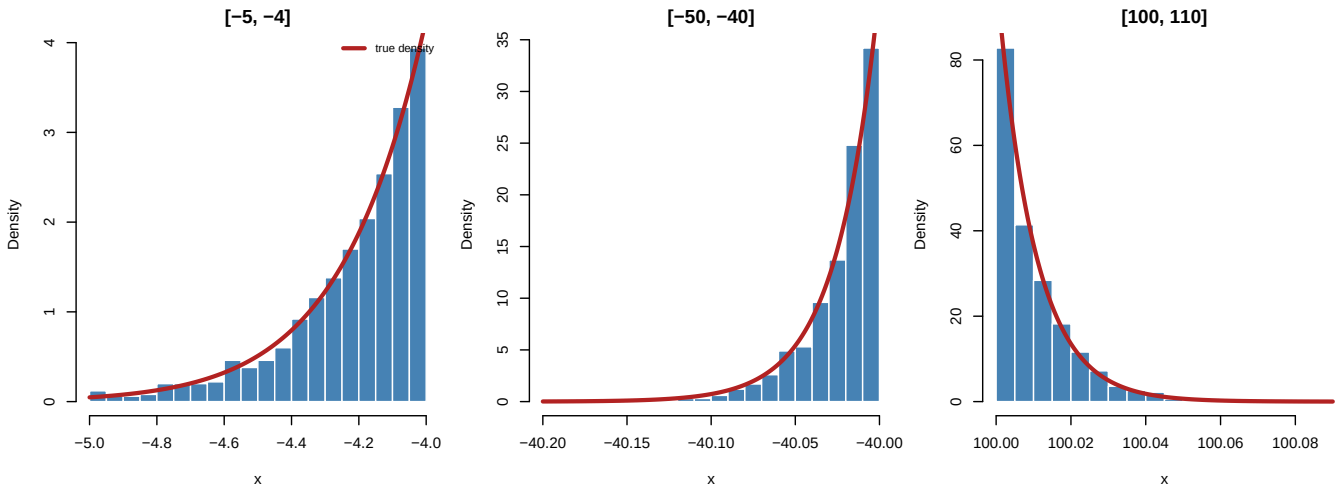


Figure 10.4: Verifies that adaptive rejection sampling produces draws from the correct truncated normal distribution, even far out in the tail. Histograms of adaptive-rejection-sampled draws against the true truncated-normal density (red), in three increasingly extreme truncation windows: ars matches the target exactly regardless of where the window sits, only the cost per draw changes (Table 10.4).

A shinylive app runs this truncated normal example step by step. It shows how adaptive rejection sampling improves its envelope automatically, so that it becomes more efficient as sampling proceeds.

10.6 Example 4: Gamma Distribution by ARS on $\log X$

The Gamma$(\alpha, 1)$ log density $(\alpha - 1)\log x - x$ has second derivative $-(\alpha - 1)/x^2$, so it is concave only when $\alpha \geq 1$. For $\alpha < 1$ the density is not log-concave (it rises to ∞ at $x = 0$) and ARS cannot be applied to X directly. A change of variable fixes this. Let $Y = \log X$. By the Jacobian $dx/dy = e^y$, Y has density

$$f_Y(y) \propto \exp\{h(y)\}, \qquad h(y) = \alpha y - e^y, \tag{10.24}$$

with $h'(y) = \alpha - e^y$ and $h''(y) = -e^y < 0$. So $\log f_Y$ is concave for **every** $\alpha > 0$.

Target. For X truncated to (l, u), with $a = \log l$ and $b = \log u$ (so $a = -\infty$ when $l = 0$ and $b = \infty$ when $u = \infty$),

$$f(y) = \exp\{\alpha y - e^y\}\, \mathbb{1}\{a < y < b\}. \tag{10.25}$$

Envelope (ARS). With tangent points $y_1 < \cdots < y_k$ in (a, b), the tangent at y_j is

$$\ell_j(y) = h(y_j) + h'(y_j)(y - y_j) = (\alpha - e^{y_j})\, y + e^{y_j}(y_j - 1), \tag{10.26}$$

using $h(y_j) - y_j h'(y_j) = e^{y_j}(y_j - 1)$. Adjacent tangents ℓ_j and ℓ_{j+1} cross at

$$z_j = \frac{e^{y_{j+1}}(y_{j+1} - 1) - e^{y_j}(y_j - 1)}{e^{y_{j+1}} - e^{y_j}}, \qquad j = 1, \dots, k-1. \tag{10.27}$$

The envelope uses tangent j between z_{j-1} and z_j, with $z_0 = a$ and $z_k = b$:

$$g(y) = \begin{cases} \exp\{(\alpha - e^{y_1})\, y + e^{y_1}(y_1 - 1)\}, & a < y \le z_1, \\ \exp\{(\alpha - e^{y_2})\, y + e^{y_2}(y_2 - 1)\}, & z_1 < y \le z_2, \\ \quad\vdots & \quad\vdots \\ \exp\{(\alpha - e^{y_k})\, y + e^{y_k}(y_k - 1)\}, & z_{k-1} < y < b. \end{cases} \tag{10.28}$$

Each piece is an exponential function of y. If $a = -\infty$, the first piece must rise with slope $\alpha - e^{y_1} > 0$, i.e. $y_1 < \log \alpha$, so that g decays to the left. If $b = \infty$, the last piece must fall, i.e. $y_k > \log \alpha$. Otherwise g has infinite area and cannot be normalized.

Proof. $g \ge f$**.** On the piece $z_{j-1} < y \le z_j$, $\log g(y) = \ell_j(y)$, and

$$\ell_j(y) - h(y) = e^y - e^{y_j} - e^{y_j}(y - y_j) = e^{y_j}(e^t - 1 - t), \qquad t = y - y_j, \tag{10.29}$$

which is ≥ 0 because $e^t \ge 1 + t$ for all t. So $\log g \ge h$ on every piece, i.e. $g \ge f$ on (a, b). (As in Example 3, this g is the same as the hull $\min_j \ell_j$: since $\ell_j(y) - \ell_{j+1}(y) = (e^{y_{j+1}} - e^{y_j})(y - z_j)$, tangent j lies below tangent $j + 1$ exactly when $y \le z_j$.) □

ARS can sample Y, and $X = e^Y$ is then an exact Gamma draw. Truncating X to (l, u) just truncates Y to $(\log l, \log u)$, which keeps h concave. With $l = 0$ the lower end is $-\infty$, and the left tail of Y is long when α is small: $h(y) \approx \alpha y$ there, an exponential tail with the slow rate α.

It is worth writing ARS out in full here. The `ars` package fails on this target when $\alpha < 1$. For $\alpha = 0.3$ its draws of $\log X$ have mean about -5.3 (over several seeds) instead of the true $E(\log X) = \psi(0.3) = -3.50$, where ψ is the digamma function. For $\alpha = 0.05$ it prints repeated internal errors (`ifault = 5`) and still returns biased draws. Both problems seem to come from the long, nearly linear left tail of h. The sampler below follows the three steps of the adaptive algorithm above. Every rejected point becomes a new tangent. All areas and inverse CDFs are computed on the log scale, so a piece with very little mass cannot overflow or underflow. The squeeze test is left out for clarity, so every proposal costs one evaluation of h.

```
### tangent-line hull for a concave h on (a, b); a = -Inf and/or b = Inf allowed.
### z holds the piece boundaries: a, the tangent intersections, b
build_hull <- function(ys, h, hprime, a, b) {
    ys <- sort(unique(ys)); hy <- h(ys); hpy <- hprime(ys); k <- length(ys)
    z <- c(a, numeric(k - 1), b)
    for (i in seq_len(k - 1))
        z[i + 1] <- (hy[i + 1] - hy[i] - ys[i + 1] * hpy[i + 1] + ys[i] * hpy[i]) /
```

```
                    (hpy[i] - hpy[i + 1])
    list(x = ys, hx = hy, hp = hpy, z = z)
}

### hull value at the two ends of each piece (-Inf at an infinite end,
### where the tangent slopes downhill), and log area under exp(piece)
piece_ends <- function(hull) {
    k <- length(hull$x)
    lo <- hull$hx + hull$hp * (hull$z[1:k] - hull$x)
    hi <- hull$hx + hull$hp * (hull$z[2:(k + 1)] - hull$x)
    lo[is.infinite(hull$z[1:k])] <- -Inf; hi[is.infinite(hull$z[2:(k + 1)])] <- -Inf
    cbind(lo, hi)
}
log_areas <- function(hull, v) {
    m <- pmax(v[, 1], v[, 2]); d <- pmin(v[, 1], v[, 2])
    flat <- abs(hull$hp) < 1e-10
    ifelse(flat, v[, 1] + log(diff(hull$z)), m + log1p(-exp(d - m)) - log(abs(hull$hp)))
}

### one draw from the normalized envelope: pick a piece in proportion to its
### area, then invert that piece's exponential CDF (on the log scale)
sample_hull <- function(hull) {
    v <- piece_ends(hull); la <- log_areas(hull, v)
    j <- sample.int(length(la), 1, prob = exp(la - max(la)))
    U <- runif(1)
    if (abs(hull$hp[j]) < 1e-10) {
        y <- hull$z[j] + U * (hull$z[j + 1] - hull$z[j]); logg <- hull$hx[j]
    } else {
        m <- max(v[j, ])
        logg <- m + log((1 - U) * exp(v[j, 1] - m) + U * exp(v[j, 2] - m))
        y <- hull$x[j] + (logg - hull$hx[j]) / hull$hp[j]
    }
    c(y = unname(y), logg = unname(logg))
}

### ARS for Y = log X, X ~ Gamma(alpha, 1) truncated to (l, u):
### h(y) = alpha*y - exp(y) is concave for every alpha > 0
ars_loggamma <- function(n, alpha, l = 0, u = Inf) {
    a <- log(l); b <- log(u); m <- log(alpha)       # m = mode of h
    h <- function(y) alpha * y - exp(y)
    hprime <- function(y) alpha - exp(y)
    ### starting points: an infinite end needs a tangent sloping downhill
    ### towards it, i.e. a point left of the mode if a = -Inf, right if b = Inf
    lo <- if (is.finite(a)) a else min(b, m) - 1
    hi <- if (is.finite(b)) b else max(a, m) + 1
    hull <- build_hull(lo + (hi - lo) * c(0.1, 0.5, 0.9), h, hprime, a, b)
    y <- numeric(n); got <- 0; proposed <- 0
    while (got < n) {
        d <- sample_hull(hull); proposed <- proposed + 1
        if (log(runif(1)) <= h(d["y"]) - d["logg"]) {
            got <- got + 1; y[got] <- d["y"]
```

```
        } else {                                    # rejected: add a tangent there
            hull <- build_hull(c(hull$x, d["y"]), h, hprime, a, b)
        }
    }
    list(x = exp(y), y = y, accept = n / proposed, tangents = length(hull$x))
}
```

Table 10.5 checks the sampler on shapes well below and above 1, with and without truncation. The true truncated mean comes from the identity $x\,\text{dgamma}(x;\alpha) = \alpha\,\text{dgamma}(x;\alpha+1)$, which gives

$$E(X \mid l < X < u) = \alpha\,\frac{P_{\alpha+1}(l,u)}{P_{\alpha}(l,u)}, \qquad P_{\alpha}(l,u) = P(l < X < u),\ X \sim \text{Gamma}(\alpha,1). \tag{10.30}$$

```
### P(l < X < u) for X ~ Gamma(alpha, 1)
p_trunc <- function(alpha, l, u) pgamma(u, alpha) - pgamma(l, alpha)
lg_cases <- list(c(0.05, 0, Inf), c(0.3, 0, Inf), c(0.3, 1, Inf), c(0.5, 0, 2),
                 c(2, 0.5, 3), c(5, 0, 1), c(1, 15, Inf))
set.seed(11)
lg_tab <- t(sapply(lg_cases, function(cc) {
    alpha <- cc[1]; l <- cc[2]; u <- cc[3]
    r <- ars_loggamma(5000, alpha, l, u)
    F_tr <- function(q) (pgamma(pmin(pmax(q, l), u), alpha) - pgamma(l, alpha)) /
        p_trunc(alpha, l, u)
    c(alpha = alpha, l = l, u = u, accept = r$accept, tangents = r$tangents,
      true_mean = alpha * p_trunc(alpha + 1, l, u) / p_trunc(alpha, l, u),
      sample_mean = mean(r$x), ks_p = ks.test(r$x, F_tr)$p.value)
}))
knitr::kable(lg_tab, digits = c(2, 2, 2, 4, 0, 4, 4, 3),
             col.names = c("alpha", "l", "u", "acceptance", "tangents",
                           "true E(X)", "sample mean", "KS p-value"))
```

Table 10.5: Verifies the ARS sampler on log X for truncated Gamma distributions, including shapes below 1 where the Gamma density itself is not log-concave. Each row uses 5,000 draws. The acceptance rate stays above 99% after only a dozen or two tangents, the sample means match the exact truncated means, and the Kolmogorov-Smirnov p-values show no departure from the target.

alpha	l	u	acceptance	tangents	true E(X)	sample mean	KS p-value
0.05	0.0	Inf	0.9966	20	0.0500	0.0509	0.867
0.30	0.0	Inf	0.9956	25	0.3000	0.2979	0.465
0.30	1.0	Inf	0.9966	20	1.7583	1.7602	0.952
0.50	0.0	2	0.9958	24	0.3869	0.3889	0.543
2.00	0.5	3	0.9972	17	1.5828	1.5839	0.745
5.00	0.0	1	0.9986	10	0.8118	0.8102	0.666
1.00	15.0	Inf	0.9982	12	16.0000	16.0023	0.989

Figure 10.5 shows the draws for three of these cases on both scales. On the y scale every target is a smooth

log-concave curve, even for $\alpha = 0.3$, where the density of X (bottom left) has a pole at 0. Truncation at $l = 1$ (middle) or to $(0.5, 3)$ (right) only cuts the y-density at $\log l$ and $\log u$.

```
fig_cases <- list(c(0.3, 0, Inf), c(0.3, 1, Inf), c(2, 0.5, 3))
set.seed(12)
draws <- lapply(fig_cases, function(cc) ars_loggamma(5000, cc[1], cc[2], cc[3]))
lab <- function(cc) sprintf("alpha = %g, (l, u) = (%g, %g)", cc[1], cc[2], cc[3])
par(mfrow = c(2, 3), mar = c(4, 4, 2.2, 1))
for (i in 1:3) {
    cc <- fig_cases[[i]]; P <- p_trunc(cc[1], cc[2], cc[3])
    hist(draws[[i]]$y, breaks = 30, freq = FALSE, col = "steelblue", border = "white",
         main = lab(cc), xlab = "y = log x", cex.main = 0.95)
    curve(exp(cc[1] * x - exp(x) - lgamma(cc[1])) / P, add = TRUE, lwd = 3,
          col = "firebrick", n = 400)
}
for (i in 1:3) {
    cc <- fig_cases[[i]]; P <- p_trunc(cc[1], cc[2], cc[3]); x <- draws[[i]]$x
    xmax <- quantile(x, 0.99)          # crop the long right tail for display
    hist(x, breaks = c(seq(cc[2], xmax, length.out = 30), max(x)), freq = FALSE,
         xlim = c(cc[2], xmax), col = "grey60", border = "white",
         main = lab(cc), xlab = "x = exp(y)", cex.main = 0.95)
    curve(dgamma(x, cc[1]) / P, from = max(cc[2], 1e-3), to = xmax, add = TRUE,
          lwd = 3, col = "firebrick", n = 400)
}
```

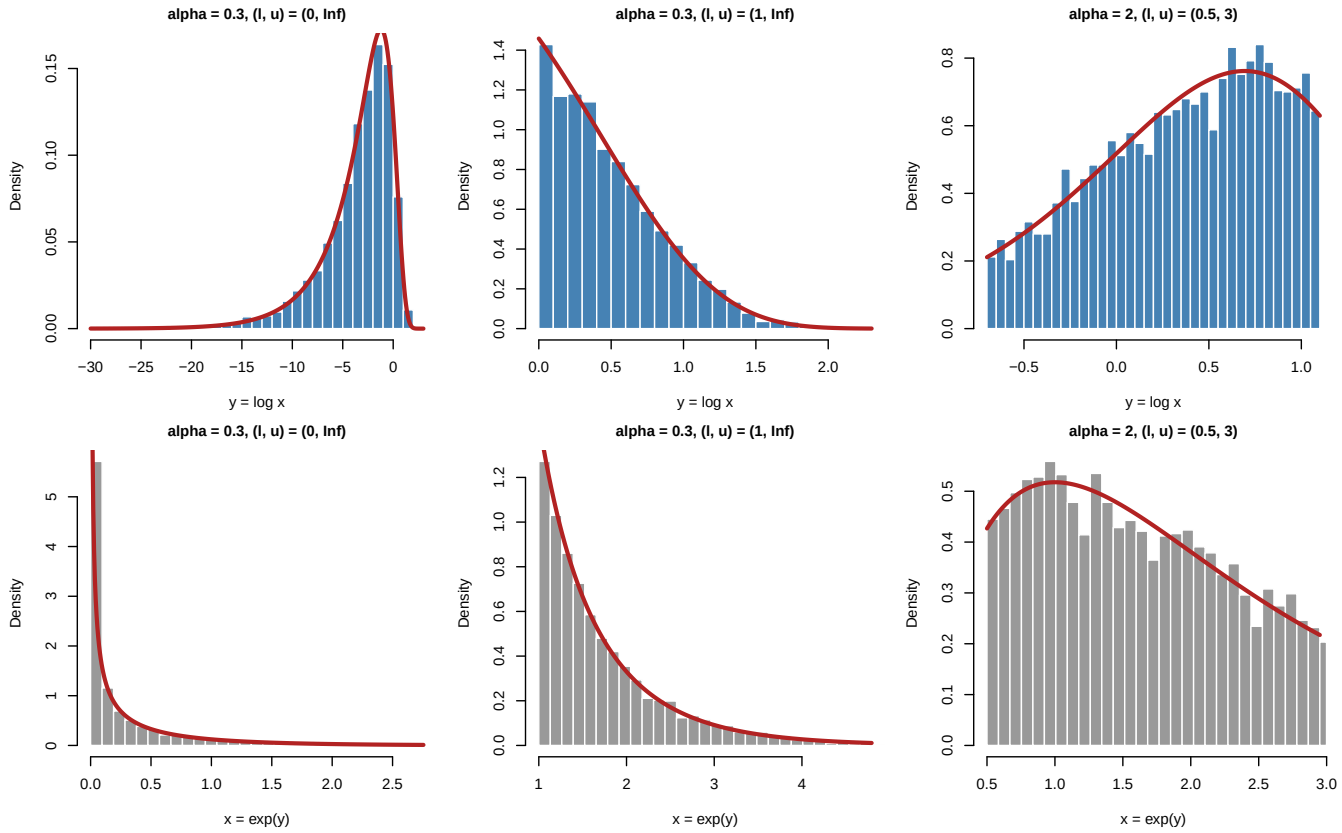


Figure 10.5: Verifies that sampling Y = log X by adaptive rejection sampling and returning X = exp(Y) gives exact draws from the truncated Gamma distribution, including for shapes below 1. Top: histograms of the ARS draws of Y = log X against the true log-concave density of Y (red). Bottom: the same draws transformed to X = exp(Y), against the truncated Gamma density. The bottom-left density has a pole at 0 and is not log-concave, but the density of Y above it is.

10.6.1 Shinylive App for Sampling a Gamma Distribution by ARS on $\log X$

There is a shinylive app that runs this sampler one proposal at a time. You choose $\alpha \in (0, 5]$ and a truncation interval (l, u). The app shows the tangent hull adapting to $h(y)$, and the histograms of Y and $X = e^Y$ filling in. A sidebar table compares the running estimates of $E(\log X)$ and $E(X)$ with their exact values.

10.7 Summary

- Rejection sampling turns draws from an envelope $g \geq f$ into *exact* iid draws from f, using only the ratio f/g; no normalizing constants are needed. The geometric view — uniform points under g, keep those under f — explains why this works and why efficiency is exactly the ratio of the two areas.
- A good envelope is tight (high acceptance) and easy to sample; finding one by hand, as in the Cauchy envelope for the Gamma distribution above, requires matching the target's tails and curvature and can be delicate.

- Adaptive rejection sampling automates envelope construction for any log-concave target via tangent lines to $\log f$, tightening the envelope on every rejection.
- A target that is not log-concave can often be made so by a change of variable: the Gamma density with $\alpha < 1$ is not log-concave, but the density of $\log X$ is for every α, so ARS on $\log X$ followed by $X = e^Y$ gives exact Gamma draws.
- Limitations: an envelope with a bounded ratio g/f may not exist when f has heavier tails than any convenient g, and in high dimensions the acceptance rate of a product envelope typically decays exponentially with dimension, for the same curse-of-dimensionality reason quadrature grids fail (Unit 8). These limitations motivate **importance sampling**, which reweights instead of rejecting, and **Markov chain Monte Carlo**, which builds a chain that adapts to the target as it runs — the subjects of the next two chapters.

11 Importance Sampling

```
library(ars)
library(latex2exp)
```

11.1 Goal and Idea

11.1.1 The Goal

Let $f(\theta)$ be an **unnormalized** function of interest. In Bayesian inference it is typically the product of a prior and a likelihood, $f(\theta) = \pi(\theta)L(\theta)$. Two integrals with respect to $f(\theta)$ come up again and again:

1. **The normalizing constant** (in Bayesian inference, the marginal likelihood),

$$C_f = \int f(\theta)\, d\theta. \tag{11.1}$$

2. **An expectation** (in Bayesian inference, a posterior expectation),

$$A = \frac{\int a(\theta)\, f(\theta)\, d\theta}{\int f(\theta)\, d\theta} = E_f[a(\theta)], \qquad \theta \sim \frac{f(\theta)}{C_f}. \tag{11.2}$$

Plain Monte Carlo would average $a(\theta)$ over draws from $f(\theta)/C_f$, which in most problems we cannot produce. Rejection sampling turns draws from an envelope $g(\theta) \geq f(\theta)$ into exact draws from $f(\theta)/C_f$, but every rejected proposal is wasted, and in the tails of a poor envelope almost every proposal is rejected. An envelope may also be hard to find at all.

11.1.2 The Idea: Reweighting

Importance sampling keeps every proposal instead of discarding most of them. Draw from an easier function $g(\theta)$ and attach a *weight* to each draw that corrects for sampling from the wrong distribution:

- Draw $\theta \sim g(\theta)/C_g$, where $g(\theta)$ is another, possibly unnormalized, function we can sample from, with $C_g = \int g(\theta)\, d\theta$.
- Attach the **importance weight**

$$w(\theta) = \frac{f(\theta)}{g(\theta)}. \tag{11.3}$$

- Hope that $w(\theta)$ is roughly constant, that is, that $g(\theta)$ has roughly the shape of $f(\theta)$.

Nothing is ever thrown away, and no envelope condition $g(\theta) \geq f(\theta)$ is required. The price is that the sample is no longer an exact iid sample from $f(\theta)/C_f$, only a *weighted* one.

The one requirement on $g(\theta)$ is that it be positive wherever $f(\theta)$ is:

$$\{\theta : g(\theta) > 0\} \supseteq \{\theta : f(\theta) > 0\}. \tag{11.4}$$

Otherwise part of $f(\theta)$ is never visited and the estimate is biased. Even when the requirement holds, a proposal can put so little mass where $f(\theta)$ lives that the region is never visited in practice; Example 3 below shows how badly that can go.

11.1.3 Picture: Areas and Weights

Figure 11.1 shows the idea for two univariate functions. The area under $f(\theta)$ is C_f and the area under $g(\theta)$ is C_g. The draws θ_i come from $g(\theta)/C_g$, so they are dense where $g(\theta)$ is high. Each carries the weight $w(\theta_i) = f(\theta_i)/g(\theta_i)$: where $f(\theta)$ rises above $g(\theta)$ the weights are large, and where $g(\theta)$ dominates they are small. The average weight estimates the area ratio C_f/C_g, as the next section shows.

```
f  <- function(x) 2 * dnorm(x, 0.8, 0.8)          # unnormalized target, C_f = 2
g  <- function(x) 1.5 * dnorm(x, 0, 1.4)          # proposal, C_g = 1.5
xs <- seq(-4.5, 4.5, length.out = 600)
th <- c(-3.0, -1.9, -1.0, -0.3, 0.4, 1.0, 1.6, 2.5, 3.4)
w  <- f(th) / g(th)
sc <- 0.5 * max(f(xs)) * 1.08 / max(w)            # tallest weight reaches half the y-axis

par(mar = c(4.2, 4.5, 1, 4.5), cex = 1.3)
plot(xs, f(xs), type = "n", ylim = c(0, max(f(xs)) * 1.08), yaxs = "i",
     xlab = expression(theta), ylab = "function value", las = 1)
polygon(c(xs, rev(xs)), c(g(xs), rep(0, length(xs))),
        col = adjustcolor("royalblue", 0.30), border = NA)
polygon(c(xs, rev(xs)), c(f(xs), rep(0, length(xs))),
        col = adjustcolor("firebrick", 0.30), border = NA)
lines(xs, g(xs), col = "royalblue", lwd = 3)
lines(xs, f(xs), col = "firebrick", lwd = 3)
segments(th, 0, th, w * sc, col = "darkgreen", lwd = 4)
points(th, w * sc, pch = 19, col = "darkgreen", cex = 1.3)
points(th, rep(0, length(th)), pch = 17, col = "black", cex = 1.6, xpd = NA)
wt <- pretty(c(0, max(w)))
axis(4, at = wt * sc, labels = wt, las = 1, col.axis = "darkgreen", col = "darkgreen")
mtext(expression(w(theta[i]) == f(theta[i]) / g(theta[i])), side = 4, line = 3.2,
      col = "darkgreen", cex = 1.3)
legend("topleft", bty = "n",
       legend = c(expression("area under " * f == C[f]), expression("area under " * g == C[g]),
                  expression(theta[i] %~% g / C[g]), "weight (right axis)"),
       fill = c(adjustcolor("firebrick", 0.3), adjustcolor("royalblue", 0.3), NA, NA),
       border = c("firebrick", "royalblue", NA, NA),
       pch = c(NA, NA, 17, NA), lty = c(NA, NA, NA, 1), lwd = c(NA, NA, NA, 4),
       col = c(NA, NA, "black", "darkgreen"))
```

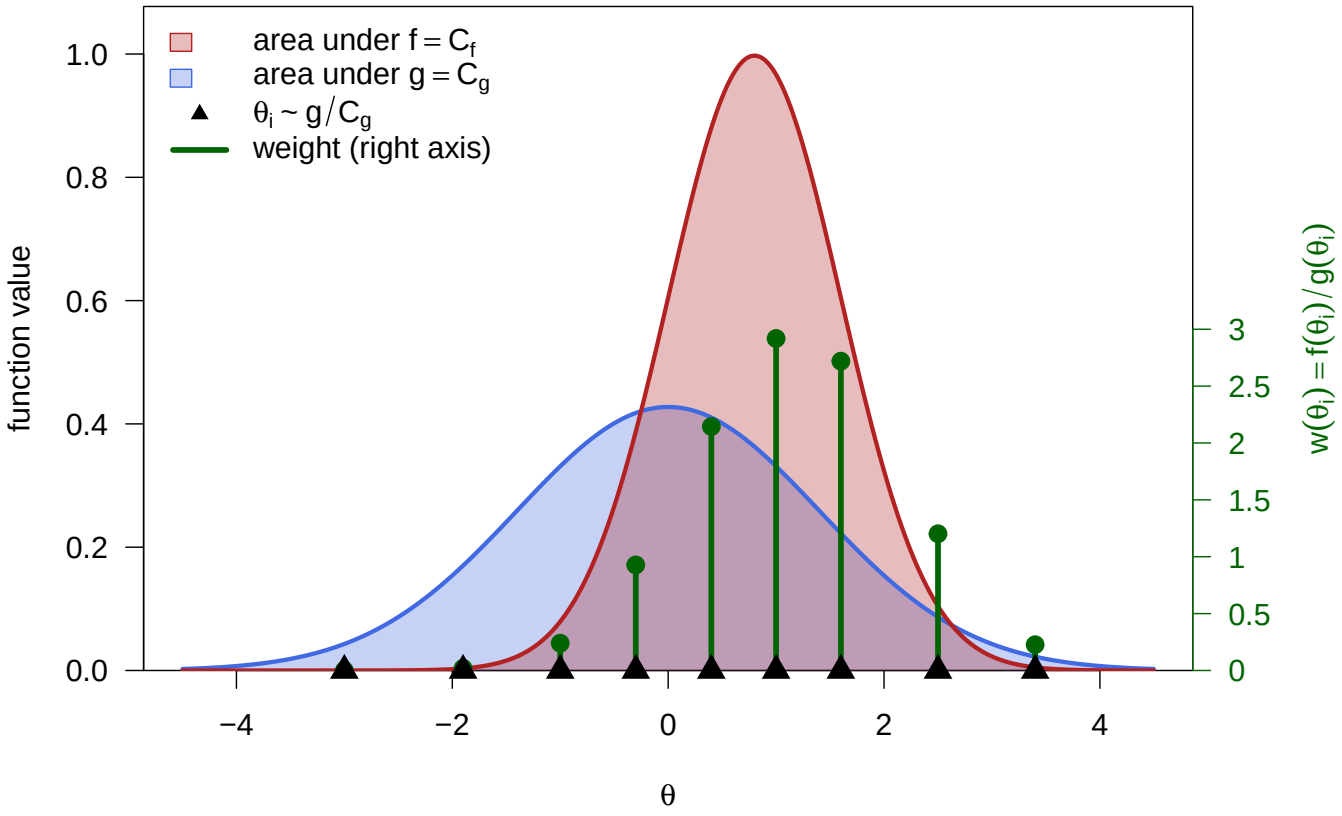


Figure 11.1: Shows importance weights as the correction from the proposal to the target. The unnormalized target f (red, area C_f) and proposal g (blue, area C_g); black triangles are draws from g/C_g, and each green segment is the weight w = f/g at that draw (right axis), scaled so that the largest weight reaches half the height of the plot.

11.2 The Importance Reweighting Formula

11.2.1 Derivation

Let $C_f = \int f(\theta)\, d\theta$, $C_g = \int g(\theta)\, d\theta$, and $w(\theta) = f(\theta)/g(\theta)$.

The ratio of normalizing constants. Multiply and divide by $g(\theta)$ inside the integral for C_f:

$$\frac{C_f}{C_g} = \frac{\int f(\theta)\, d\theta}{\int g(\theta)\, d\theta} = \int \frac{f(\theta)}{g(\theta)} \cdot \frac{g(\theta)}{C_g}\, d\theta = E_g[w(\theta)], \tag{11.5}$$

an expectation under the normalized proposal $g(\theta)/C_g$.

The expectation of $a(\theta)$. The same trick in the numerator, then the result above in the denominator:

$$\begin{aligned} E_f[a(\theta)] &= \frac{\int a(\theta)\, f(\theta)\, d\theta}{C_f} = \frac{\displaystyle\int a(\theta)\, \frac{f(\theta)}{g(\theta)} \cdot \frac{g(\theta)}{C_g}\, d\theta}{C_f/C_g} \\ &= \frac{E_g[a(\theta)\, w(\theta)]}{C_f/C_g} = \frac{E_g[a(\theta)\, w(\theta)]}{E_g[w(\theta)]}. \end{aligned} \tag{11.6}$$

Dividing by $g(\theta)$ is legitimate only on $\{\theta : g(\theta) > 0\}$, which is where the support requirement comes from.

11.2.2 The Estimators

Draw $\theta_1, \dots, \theta_n \overset{iid}{\sim} g(\theta)/C_g$ and compute $w_i = w(\theta_i) = f(\theta_i)/g(\theta_i)$. Replacing each E_g above by a sample average gives two estimators.

Ratio of normalizing constants, and C_f itself when C_g is known.

$$\frac{\widehat{C_f}}{C_g} = \frac{1}{n}\sum_{i=1}^{n} w_i, \qquad \hat{C}_f = C_g \cdot \frac{\widehat{C_f}}{C_g} = \frac{C_g}{n}\sum_{i=1}^{n} \frac{f(\theta_i)}{g(\theta_i)}. \tag{11.7}$$

Both are unbiased. With a normalized proposal ($C_g = 1$), $\hat{C}_f$ is just the average weight; this is how importance sampling estimates a marginal likelihood.

Self-normalized expectation.

$$\hat{A} = \frac{\frac{1}{n}\sum_i a(\theta_i)\, w_i}{\frac{1}{n}\sum_i w_i} = \frac{1}{n}\sum_{i=1}^{n} a(\theta_i)\, \hat{w}_i^\star, \qquad \hat{w}_i^\star = \frac{w_i}{\frac{1}{n}\sum_{j=1}^{n} w_j}, \qquad \frac{1}{n}\sum_{i=1}^{n} \hat{w}_i^\star = 1. \tag{11.8}$$

The **normalized weights** $\hat{w}_i^\star$ have mean 1. They estimate the normalized weight function

$$w^\star(\theta) = \frac{f(\theta)/C_f}{g(\theta)/C_g}, \qquad E_g[w^\star(\theta)] = 1, \tag{11.9}$$

the ratio of the two normalized densities, evaluated at θ_i. They turn the proposal sample into a **weighted sample from** $f(\theta)/C_f$, and because the unknown constants in $f(\theta)$ and $g(\theta)$ cancel in the ratio, $\hat{A}$ needs both functions only up to proportionality. $\hat{A}$ is a ratio of two averages, so it is biased, with bias of order $O(1/n)$, but it is consistent.

11.2.3 Effective Sample Size

How many equally informative draws is a weighted sample worth? The **effective sample size** (ESS) is

$$n_{\text{eff}} = \frac{n}{\frac{1}{n}\sum_{i=1}^{n} (\hat{w}_i^\star)^2} = \frac{\left(\sum_i w_i\right)^2}{\sum_i w_i^2}. \tag{11.10}$$

If the weights are all equal (the proposal has the shape of the target), $\hat{w}_i^\star = 1$ and $n_{\text{eff}} = n$: full efficiency. If one weight dominates, $n_{\text{eff}} \approx 1$: only one draw is doing any work.

11.2.4 Derivation of ESS

The normalizing constants cancel in $\hat{A}$: multiplying every w_i by the same constant leaves $\hat{A}$ unchanged. So, without loss of generality, assume $f(\theta)$ and $g(\theta)$ are both normalized densities. Then $w(\theta) = w^\star(\theta)$, $E_g[w(\theta)] = 1$, and for any function $h(\theta)$,

$$E_g[w(\theta)\, h(\theta)] = \int h(\theta) \frac{f(\theta)}{g(\theta)}\, g(\theta)\, d\theta = E_f[h(\theta)]. \tag{11.11}$$

The derivation then takes five short steps.

1. **Write the error as a ratio.** Since A is a constant,

$$\hat{A} - A = \frac{\frac{1}{n}\sum_i w_i\, a(\theta_i)}{\frac{1}{n}\sum_i w_i} - A = \frac{\frac{1}{n}\sum_i w_i\, (a(\theta_i) - A)}{\frac{1}{n}\sum_i w_i}. \tag{11.12}$$

2. **Delta method.** The denominator converges to $E_g[w(\theta)] = 1$. Replacing it by its limit is the first-order (delta-method) approximation, and the only place the delta method enters:

$$\hat{A} - A \approx \frac{1}{n}\sum_{i=1}^{n} Z_i, \qquad Z_i = w_i\, (a(\theta_i) - A). \tag{11.13}$$

   The Z_i are iid with mean $E_g[Z] = E_f[a(\theta) - A] = 0$.

3. **Variance of an iid mean.**

$$\mathrm{Var}(\hat{A}) \approx \frac{1}{n} E_g[Z^2] = \frac{1}{n} E_g\Big[w(\theta)^2 (a(\theta) - A)^2\Big] = \frac{1}{n} E_f\Big[w(\theta)(a(\theta) - A)^2\Big]. \tag{11.14}$$

4. **Assume $w(\theta)$ and $(a(\theta) - A)^2$ are uncorrelated under f.** The expectation then factors, and $E_f[w(\theta)] = E_g[w(\theta)^2]$ by the identity above:

$$E_f\Big[w(\theta)(a(\theta) - A)^2\Big] \approx E_f[w(\theta)]\, \mathrm{Var}_f\,(a(\theta)) = E_g[w(\theta)^2]\, \mathrm{Var}_f\,(a(\theta)). \tag{11.15}$$

5. **Match an iid sample from f.** An average of m iid draws from f has variance $\mathrm{Var}_f(a(\theta))/m$. Matching,

$$\mathrm{Var}(\hat{A}) \approx \frac{\mathrm{Var}_f\,(a(\theta))}{n / E_g[w(\theta)^2]} \quad \Longrightarrow \quad n^\star_{\text{eff}} = \frac{n}{E_g[w^\star(\theta)^2]} = \frac{n}{1 + \mathrm{Var}_g\,(w^\star(\theta))} \le n. \tag{11.16}$$

The population ESS $n^\star_{\text{eff}}$ is a property of $f(\theta)$ and $g(\theta)$ alone, independent of $a(\theta)$. Replacing $E_g[w^\star(\theta)^2]$ by the sample average of $(\hat{w}^\star_i)^2$ (which restores mean 1 in the sample) gives exactly the n_{eff} above, also known as Kish's formula. So n_{eff} is a plug-in estimate of $n^\star_{\text{eff}}$, not an ad hoc heuristic. Step 4 is the only real approximation; the next section shows an example where it fails. The derivation also assumes the support condition: if $g(\theta)$ misses part of $f(\theta)$, then $E_g[w^\star(\theta)] < 1$ and step 2 already fails.

```r
### effective sample size from raw (possibly unnormalized) weights
ess <- function(w) sum(w)^2 / sum(w^2)
### numerically stable log(sum(exp(lx)))
log_sum_exp <- function(lx) { m <- max(lx); m + log(sum(exp(lx - m))) }
### effective sample size computed from LOG weights (avoids under/overflow)
log_ess <- function(lw) exp(2 * log_sum_exp(lw) - log_sum_exp(2 * lw))
```

11.2.5 Limitations of the ESS Diagnostic

The variance approximation is not uniformly precise. Take $f(\theta)$ the $N(0,1)$ density, $g(\theta)$ the $N(0,2^2)$ density, $a(\theta)=\theta^2$, and $n=4000$. The empirical $\mathrm{Var}(\widehat{A})$ is about 3.15×10^{-4}, which matches the exact delta-method calculation. But $\mathrm{Var}_f(a(\theta))/n_{\text{eff}}^{\star}$ is about 7.56×10^{-4}, more than twice as large, because $w^{\star}(\theta)$ and $(a(\theta)-A)^2$ are negatively correlated here and the factorization above does not hold.

More critically, n_{eff} is computed **only from the weights that were actually observed**. If the proposal never visits the region where $f(\theta)$ and $g(\theta)$ disagree most, n_{eff} can be close to n while the estimate is badly wrong. Example 3 shows exactly this.

11.3 Example 1: The Volume of a Set

11.3.1 Indicator Functions as f and g

The estimators above never assume that $f(\theta)$ is a likelihood or a density. The simplest possible choice makes every quantity in the theory explicit. Take two indicator functions, with $A\subseteq B\subset\mathbb{R}^d$:

$$f(\theta)=I_A(\theta),\qquad g(\theta)=I_B(\theta),\qquad C_f=\int I_A(\theta)\,d\theta=|A|,\qquad C_g=|B|. \tag{11.17}$$

The target $f(\theta)/C_f$ is the uniform distribution on A, and the proposal $g(\theta)/C_g$ is the uniform distribution on B, which is easy to sample when B is a box. **The problem:** estimate the volume $C_f=|A|$, with $|B|$ known.

Draw $\theta_1,\dots,\theta_n\overset{iid}{\sim}\mathrm{Unif}(B)$. The weight is an indicator variable,

$$w_i=\frac{f(\theta_i)}{g(\theta_i)}=I_A(\theta_i)=\begin{cases}1, & \theta_i\in A,\\ 0, & \theta_i\in B\setminus A,\end{cases}\qquad w_i\sim\text{Bernoulli}(p),\quad p=\frac{|A|}{|B|}, \tag{11.18}$$

so the importance sampling estimator is the familiar "hit-or-miss" Monte Carlo estimator,

$$\widehat{C_f}=|B|\cdot\frac{1}{n}\sum_{i=1}^{n}I_A(\theta_i)=|B|\,\frac{N_A}{n},\qquad N_A=\#\{i:\theta_i\in A\}\sim\text{Binomial}(n,p). \tag{11.19}$$

The darts used earlier in the course to estimate π as the area of a circle are this estimator with B a square.

```
local({
set.seed(3)
rA <- function(t) 0.30 + 0.07 * sin(3 * t) + 0.05 * cos(2 * t)   # boundary of A (polar, about
 ↪  the centre)
ctr <- c(0.5, 0.5); n <- 300
x <- runif(n); y <- runif(n)                                      # theta_i ~ Unif(B), B =
 ↪  unit square
t <- atan2(y - ctr[2], x - ctr[1])
hit <- sqrt((x - ctr[1])^2 + (y - ctr[2])^2) <= rA(t)             # w_i = I_A(theta_i)
tt <- seq(0, 2 * pi, length.out = 400)
par(mar = c(0.5, 0.5, 0.5, 0.5))
```

```
plot(NA, xlim = c(0, 1.9), ylim = c(0, 1), asp = 1, axes = FALSE, xlab = "", ylab = "")
rect(0, 0, 1, 1, col = "grey95", border = "black", lwd = 2)
polygon(ctr[1] + rA(tt) * cos(tt), ctr[2] + rA(tt) * sin(tt), col = "#cfe2f3", border =
 ↪  "steelblue4", lwd = 2)
points(x[!hit], y[!hit], pch = 1, col = "grey45", cex = 0.8)
points(x[hit], y[hit], pch = 19, col = "steelblue4", cex = 0.8)
text(0.5, 0.5, "A", cex = 2.2, font = 3, col = "steelblue4")
text(0.06, 0.94, "B", cex = 2.2, font = 3)
legend(1.05, 0.95, bty = "n", cex = 1.25, pch = c(19, 1), col = c("steelblue4", "grey45"),
       legend = c(sprintf("hit:  w = 1   (N_A = %d)", sum(hit)),
                  sprintf("miss: w = 0   (%d)", sum(!hit))))
text(1.08, 0.45, adj = 0, cex = 1.25,
     sprintf("n = %d draws, |B| = 1\nestimate of |A| = N_A / n = %.3f", n, mean(hit)))
text(1.08, 0.22, adj = 0, cex = 1.25, sprintf("true |A| = %.3f", integrate(function(t) rA(t)^2
 ↪  / 2, 0, 2 * pi)$value))
})
```

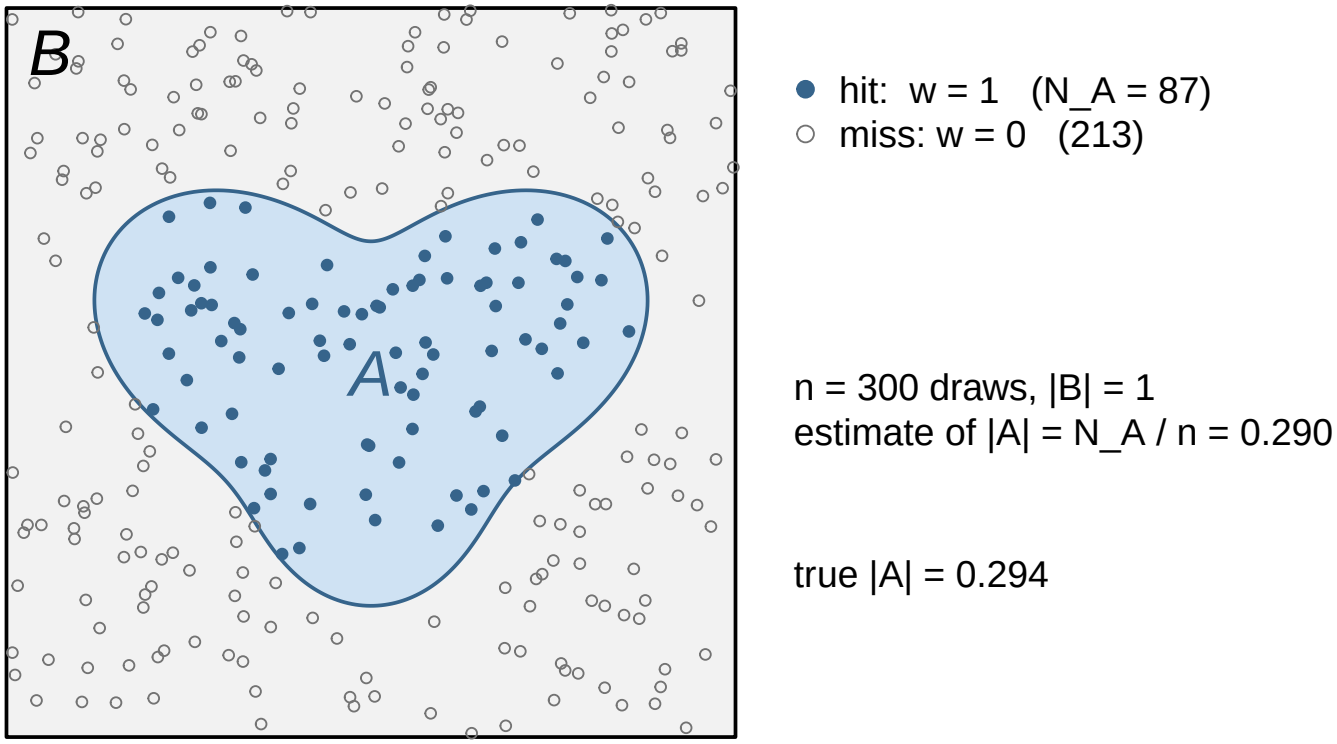


Figure 11.2: Shows the volume example as importance sampling with indicator functions. 300 draws uniform on the unit square B; filled points land in A (weight 1), open points miss (weight 0). The fraction of hits estimates p = |A|/|B|.

11.3.2 Variance

$\widehat{C_f} = |B|\, N_A/n$ is unbiased. With $N_A \sim \text{Binomial}(n, p)$ and $|B| = C_f/p$,

$$\begin{aligned} \text{Var}(\widehat{C_f}) &= \frac{|B|^2}{n^2}\,\text{Var}(N_A) = \frac{|B|^2}{n^2}\, n\, p(1-p) = |B|^2\,\frac{p(1-p)}{n} \\ &= \frac{C_f^2}{p^2}\cdot\frac{p(1-p)}{n} = C_f^2 \times \frac{1}{n}\cdot\frac{1-p}{p}. \end{aligned} \tag{11.20}$$

For a fixed target (C_f fixed), $(1-p)/p$ decreases in p. The larger $p = |A|/|B|$ is, that is, **the more tightly the proposal B fits A, the smaller the variance.** The relative standard deviation is $\sqrt{(1-p)/(np)}$; a 1% relative error needs about $n \approx 10^4(1-p)/p$ draws, which is 10^4 draws for $p = 1/2$ but about 10^6 for $p = 0.01$.

11.3.3 Effective Sample Size

The normalized weight function and its moments are

$$w^\star(\theta) = \frac{f(\theta)/C_f}{g(\theta)/C_g} = \frac{I_A(\theta)}{p}, \qquad E_g[w^\star(\theta)] = 1, \qquad E_g[w^\star(\theta)^2] = \frac{1}{p}, \tag{11.21}$$

so the population and sample effective sample sizes are

$$n^\star_{\text{eff}} = \frac{n}{E_g[w^\star(\theta)^2]} = n\,p = n\,\frac{|A|}{|B|}, \qquad \widehat{n_{\text{eff}}} = \frac{\left(\sum_i w_i\right)^2}{\sum_i w_i^2} = \frac{N_A^2}{N_A} = N_A. \tag{11.22}$$

The ESS is exactly the number of hits in A, and $E(N_A) = np = n^\star_{\text{eff}}$.

11.3.4 What n_{eff} Means

n_{eff} compares the weighted sample with an iid sample **from** $f(\theta)/C_f$. Such a sample can estimate expectations $E_f[a(\theta)]$, but it cannot estimate C_f at all (draws from the uniform distribution on A say nothing about how big A is). So to see what n_{eff} means here, use a simple $a(\theta)$: take a subset $D \subseteq A$ and $a(\theta) = I_D(\theta)$, so that

$$E_f[a(\theta)] = q = \frac{|D|}{|A|}, \quad \text{the fraction of } A \text{ lying in } D. \tag{11.23}$$

Compare two estimators of q.

- $\hat{q}_A$**, from an iid sample of size** m **from** $\text{Unif}(A)$ (direct sampling from $f(\theta)/C_f$):

$$\hat{q}_A = \frac{\#\{\theta_i \in D\}}{m}, \qquad \text{Var}(\hat{q}_A) = \frac{q(1-q)}{m}. \tag{11.24}$$

- $\hat{q}_B$**, from an importance sample of size** n **from** $\text{Unif}(B)$**.** The normalized weights and the estimate are

$$\hat{w}_i^\star = \frac{w_i}{\frac{1}{n}\sum_{j=1}^n w_j} = \begin{cases} n/N_A, & \theta_i \in A, \\ 0, & \theta_i \notin A, \end{cases} \qquad \hat{q}_B = \frac{1}{n}\sum_{i=1}^n I_D(\theta_i)\,\hat{w}_i^\star = \frac{N_D}{N_A}, \qquad N_D = \#\{\theta_i \in D\}. \tag{11.25}$$

The variance of $\hat{q}_B$. Given N_A, the hits are iid Unif(A), so $N_D \mid N_A \sim \text{Binomial}(N_A, q)$ and

$$E(\hat{q}_B \mid N_A) = q, \qquad \text{Var}(\hat{q}_B \mid N_A) = \frac{q(1-q)}{N_A}. \tag{11.26}$$

By the law of total variance, with $N_A \sim \text{Binomial}(n, p)$ (conditioning on $N_A \geq 1$),

$$\begin{aligned} \text{Var}(\hat{q}_B) &= E\big[\,\text{Var}(\hat{q}_B \mid N_A)\big] + \text{Var}\left[E(\hat{q}_B \mid N_A)\right] = q(1-q)\,E\Big[\frac{1}{N_A}\Big] + \underbrace{\text{Var}(q)}_{=0} \\ &= q(1-q)\,E\Big[\frac{1}{N_A}\Big] \;\approx\; \frac{q(1-q)}{np}, \end{aligned} \tag{11.27}$$

since $E[1/N_A] \approx 1/(np)$ for large np. Compared with $\text{Var}(\hat{q}_A) = q(1-q)/m$:

- **Given** N_A, $\hat{q}_B$ behaves exactly like $\hat{q}_A$ with $m = N_A = \widehat{n_{\text{eff}}}$: the misses contribute nothing.
- **On average**, it behaves like $\hat{q}_A$ with $m = np = n^{\star}_{\text{eff}} = n/E_g[w^{\star}(\theta)^2]$.

So in this example the effective sample size is literally the size of the iid sample from $f(\theta)/C_f$ that the weighted sample is worth.

11.3.5 Numerical Example: Area of a Crescent

Consider a region built from two disks, a crescent $A = D_1 \setminus D_2$, where D_1 is the unit disk centred at the origin and D_2 is a disk of radius $r_2 = 0.9$ centred at $(0.3, 0)$, overlapping most of D_1. This is a **constructive solid geometry (CSG)** shape: testing whether a point lies in A is a one-line computation, but the area of a Boolean combination of shapes has no elementary formula in general, which is the situation CAD and engineering software face after a few Boolean operations on solids. Two circles are simple enough that a closed form still exists, from the circle-intersection formula; it is used here only to check the estimates.

Two proposals $B \supseteq A$ are compared: a generous box $[-2, 2]^2$, with $p = |A|/16 \approx 0.056$, and the disk D_1 itself, with $p = |A|/\pi \approx 0.286$. In both cases the weight of a draw is $|B|$ if it lands in A and 0 otherwise.

```
r1 <- 1; r2 <- 0.9; d <- 0.3
in_A <- function(x, y) (x^2 + y^2 <= r1^2) & ((x - d)^2 + y^2 > r2^2)

### exact area of A, from the circle-circle intersection formula; used only
### to check the Monte Carlo estimates below
circle_intersection_area <- function(r1, r2, d) {
    r1^2 * acos((d^2 + r1^2 - r2^2) / (2 * d * r1)) +
    r2^2 * acos((d^2 + r2^2 - r1^2) / (2 * d * r2)) -
    0.5 * sqrt((-d + r1 + r2) * (d + r1 - r2) * (d - r1 + r2) * (d + r1 + r2))
}
area_true <- pi * r1^2 - circle_intersection_area(r1, r2, d)

### bounding box for the naive proposal: generous, with no attempt to hug A
xlim <- c(-2, 2); ylim <- c(-2, 2)

### importance-sampling estimate of Area(A) under one of two proposals:
```

```r
### "box" samples uniformly from the bounding box; "disk" samples uniformly
### from D1, which is known to contain A
crescent_is <- function(proposal, n, seed) {
    set.seed(seed)
    if (proposal == "box") {
        x <- runif(n, xlim[1], xlim[2]); y <- runif(n, ylim[1], ylim[2])
        Cg <- (xlim[2] - xlim[1]) * (ylim[2] - ylim[1])        # C_g = |B|
    } else {
        rr <- r1 * sqrt(runif(n)); th <- runif(n, 0, 2 * pi)
        x <- rr * cos(th); y <- rr * sin(th)
        Cg <- pi * r1^2                                        # C_g = |D_1|
    }
    w <- in_A(x, y)                                            # w = f/g = I_A(theta_i)
    list(x = x, y = y, hit = w,
         estimate = Cg * mean(w), me = 1.96 * Cg * sd(w) / sqrt(n), ess = ess(w))
}
```

```r
res_box_vis  <- crescent_is("box",  1500, seed = 1)
res_disk_vis <- crescent_is("disk", 1500, seed = 1)
theta_seq <- seq(0, 2 * pi, length.out = 200)

par(mfrow = c(1, 2))
plot(res_box_vis$x, res_box_vis$y, asp = 1, pch = 20,
     col = ifelse(res_box_vis$hit, "steelblue", "tomato"),
     xlab = "x", ylab = "y",
     main = paste(sum(res_box_vis$hit), "of 1500 hit A (box)"))
rect(xlim[1], ylim[1], xlim[2], ylim[2], border = "black", lwd = 2)
lines(r1 * cos(theta_seq), r1 * sin(theta_seq), lwd = 2)
lines(d + r2 * cos(theta_seq), r2 * sin(theta_seq), lwd = 2)

plot(res_disk_vis$x, res_disk_vis$y, asp = 1, pch = 20,
     col = ifelse(res_disk_vis$hit, "steelblue", "tomato"),
     xlab = "x", ylab = "y", xlim = xlim, ylim = ylim,
     main = paste(sum(res_disk_vis$hit), "of 1500 hit A (disk)"))
lines(r1 * cos(theta_seq), r1 * sin(theta_seq), lwd = 2)
lines(d + r2 * cos(theta_seq), r2 * sin(theta_seq), lwd = 2)
par(mfrow = c(1, 1))
```

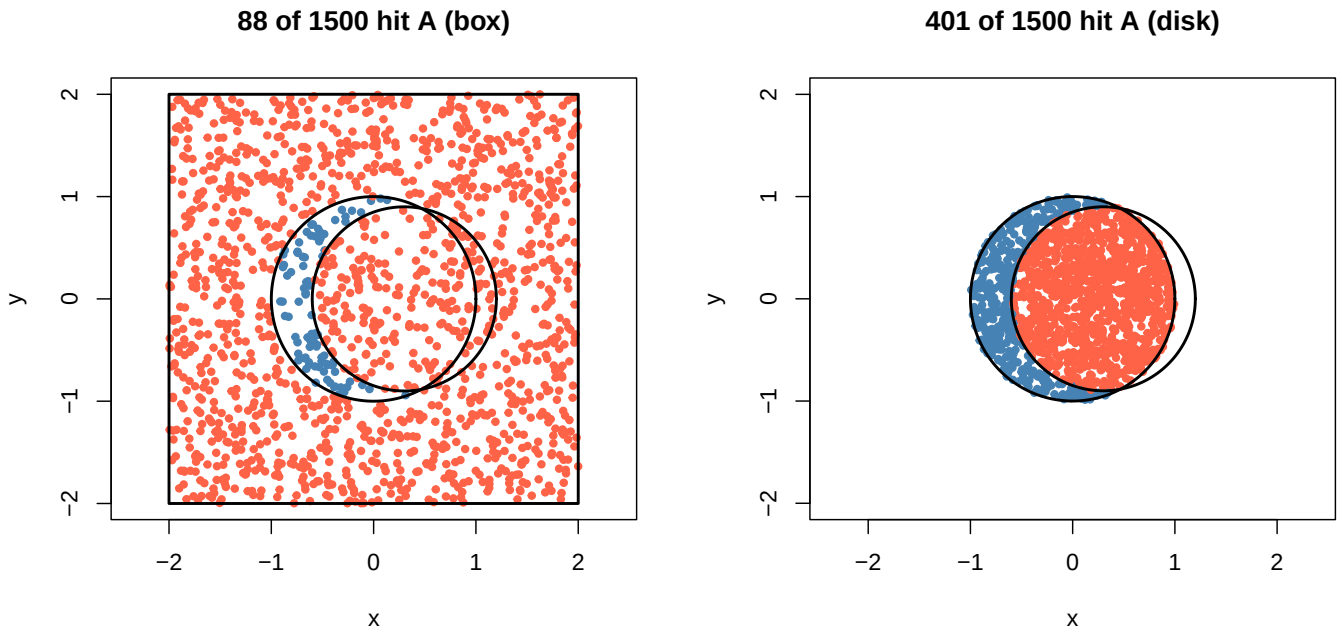


Figure 11.3: Compares a naive bounding-box proposal with a proposal matched to the target's support for estimating the area of an irregular region. 1,500 darts thrown at the crescent $A = D_1 \setminus D_2$ (outlined in black) from a generous bounding box (left) and from D_1 itself (right); blue points land in A (weight $|B|$), red points miss (weight 0).

```
n <- 5000
res_box  <- crescent_is("box",  n, seed = 1)
res_disk <- crescent_is("disk", n, seed = 1)

knitr::kable(data.frame(
    proposal = c("Bounding box (naive)", "Disk $D_1$ (importance)"),
    truth    = rep(area_true, 2),
    estimate = c(res_box$estimate, res_disk$estimate),
    ME       = c(res_box$me, res_disk$me),
    ess_pct  = 100 * c(res_box$ess, res_disk$ess) / n
), digits = 4, align = "lrrrr",
col.names = c("Proposal $g$", "True area", "Estimate", "M.E.", "ESS %"))
```

Table 11.1: Importance-sampling estimates of Area(A) with n = 5000 draws, box versus disk proposal. Both are unbiased; the ESS percentage is the hit rate, close to p = |A|/|B|.

Proposal g	True area	Estimate	M.E.	ESS %
Bounding box (naive)	0.8984	0.9056	0.1025	5.66
Disk D_1 (importance)	0.8984	0.8891	0.0392	28.30

Table 11.1 confirms the theory of this section:

- Both proposals are unbiased: importance sampling never needed A's shape, only a B that covers it.
- The ESS percentage is the hit rate, close to $p = |A|/|B|$ as $n_{\text{eff}}^{\star} = np$ predicts: about 5.6% for the box and 28.6% for D_1.
- The relative variance $(1-p)/(np)$ is about 6.7 times smaller for the disk, so the margin of error drops by about $\sqrt{6.7} \approx 2.6$ (0.103 against 0.039). A box that is too generous costs precision, not correctness. The lesson carries over to every example below: match the proposal to where the target actually lives.

11.4 Example 2: Tail Probability of a Student-t Distribution

11.4.1 The Problem and the Proposal

The target is a standard Student-t_ν distribution. The degrees of freedom ν is a **tuning parameter of the target's tail weight**: $\nu = 1$ is the standard Cauchy, and $\nu \to \infty$ recovers the standard normal. The goal is $P(c_1 < X < c_2)$ for a finite interval (c_1, c_2), estimated from draws of a truncated-normal proposal generated by adaptive rejection sampling (ARS).

As in the ARS section of the previous chapter, $\log \varphi$ is concave, and restricting it to (c_1, c_2) preserves concavity, so the `ars` package applies directly to the *proposal*: `rtnorm_ars()` below just hands it the log density and its derivative, plus the truncation bounds.

```
## ---------------------------------------------------------------------
##  rtnorm_ars(): draw from N(mu, sigma^2) truncated to (lo, hi) via the
##  ars package. log-phi is concave, so ars() applies directly; x0 gives
##  it three starting abscissae inside (lo, hi), with the rightmost past
##  mu when hi = Inf so the upper piece has negative slope.
## ---------------------------------------------------------------------
rtnorm_ars <- function(n, mu = 0, sigma = 1, lo = 0, hi = Inf) {
    logf   <- function(x) dnorm(x, mu, sigma, log = TRUE)
    fprima <- function(x) -(x - mu) / sigma^2
    if (is.finite(hi)) {
        x0 <- lo + (hi - lo) * c(0.1, 0.5, 0.9)
    } else {
        right <- max(mu, lo) + 3 * sigma
        x0 <- sort(unique(c(lo + 0.1 * sigma, max(lo + 0.2 * sigma, mu + 1e-6), right)))
    }
    ars(n, f = logf, fprima = fprima, x = x0,
        lb = TRUE, xlb = lo, ub = is.finite(hi), xub = if (is.finite(hi)) hi else 0)
}
```

The target itself is a different matter: $\log f_{t_\nu}$ is **not concave** at any finite ν. It is convex out in the tails, and only in the limit $\nu \to \infty$ (the normal) is it concave everywhere. So ARS cannot sample the target directly, and the truncated-normal proposal is a substitute for it, which is exactly why the sample needs importance weights.

11.4.2 The Target, the Proposal and the Estimator

Both functions are taken **unnormalized**: the t_ν and $N(\mu,\sigma^2)$ densities, cut off outside (c_1,c_2),

$$f(x)=\begin{cases} f_{t_\nu}(x), & c_1<x<c_2,\\ 0, & \text{otherwise,}\end{cases}\qquad g(x)=\begin{cases}\dfrac{1}{\sigma\sqrt{2\pi}}\,e^{-(x-\mu)^2/(2\sigma^2)}, & c_1<x<c_2,\\ 0, & \text{otherwise.}\end{cases}\tag{11.28}$$

Their normalizing constants are the probability we want and a known normal probability:

$$C_f=\int f(x)\,dx=P(c_1<X<c_2),\qquad C_g=\int g(x)\,dx=\Phi\!\left(\tfrac{c_2-\mu}{\sigma}\right)-\Phi\!\left(\tfrac{c_1-\mu}{\sigma}\right).\tag{11.29}$$

Draw $X_1,\dots,X_n\sim g(x)/C_g$ (the truncated normal), weight each by $w(x)=f(x)/g(x)$, and scale the average weight by the known C_g:

$$\widehat{C_f}=C_g\cdot\frac{1}{n}\sum_{i=1}^{n}w(X_i).\tag{11.30}$$

This is unbiased for C_f for *any* $g(x)$ that is positive wherever $f(x)$ is, whatever c_1,c_2,μ,σ,ν are. Because $f(x)$ and $g(x)$ share the same bounded interval, the ratio $f(x)/g(x)$ is bounded on it, so the weights also have finite variance and the usual standard error applies.

Figure 11.4 draws $f(x)$ and $g(x)$ for $\nu=1$ (Cauchy), $\mu=0$, $\sigma=1$ and $(c_1,c_2)=(1,3)$, together with 10 draws from $g(x)/C_g$ and their weights. The draws are dense near c_1, where $g(x)$ is high, and get small weights there; the rare draws on the right, where the heavy t tail lies above the normal, carry large weights. The frequency of the draws and the size of their weights balance each other.

```
nu <- 1; mu <- 0; sigma <- 1; c1 <- 1
draw_fg <- function(c2, xmax, main) {
    hi <- min(c2, xmax)
    xs <- seq(-1, xmax, length.out = 1500)
    xin <- seq(c1, hi, length.out = 800)
    ft <- dt(xs, nu); gn <- dnorm(xs, mu, sigma)
    plot(xs, ft, type = "n", ylim = c(0, max(ft, gn) * 1.05),
         xlab = "x", ylab = "unnormalized density", main = main)
    lines(xs, ft, col = adjustcolor("firebrick", 0.35), lty = 3, lwd = 2)
    lines(xs, gn, col = adjustcolor("royalblue", 0.35), lty = 3, lwd = 2)
    polygon(c(c1, xin, hi), c(0, dt(xin, nu), 0), col = adjustcolor("firebrick", 0.15), border
     ↪  = NA)
    polygon(c(c1, xin, hi), c(0, dnorm(xin, mu, sigma), 0), col = adjustcolor("royalblue",
     ↪  0.15), border = NA)
    lines(xin, dt(xin, nu), col = "firebrick", lwd = 4)
    lines(xin, dnorm(xin, mu, sigma), col = "royalblue", lwd = 4)
    abline(v = c(c1, if (is.finite(c2)) c2), lty = 2, col = "grey40")
    ## 10 draws from g(x)/C_g (truncated normal, inverse CDF) and their weights
    set.seed(48)
    xp <- qnorm(runif(10, pnorm(c1, mu, sigma), pnorm(hi, mu, sigma)), mu, sigma)
    w  <- dt(xp, nu) / dnorm(xp, mu, sigma)
    ytop <- max(ft, gn) * 1.05; sc <- 0.5 * ytop / max(w)
    segments(xp, 0, xp, w * sc, col = "darkgreen", lwd = 2)
```

```
    points(xp, w * sc, pch = 19, col = "darkgreen")
    points(xp, rep(0, 10), pch = 17, cex = 1.2, xpd = NA)
    wt <- pretty(c(0, max(w)))
    axis(4, at = wt * sc, labels = wt, las = 1, col.axis = "darkgreen", col = "darkgreen")
    mtext(TeX(r"($w(x) = f(x)/g(x)$)"), side = 4, line = 2.8, col = "darkgreen")
    legend("topright", bg = "white", box.col = NA, lwd = c(4, 4, 2, 3), lty = c(1, 1, 3, 1),
           col = c("firebrick", "royalblue", "grey60", "darkgreen"),
           legend = TeX(c(r"($f(x)$: $t_\nu$ on $(c_1,c_2)$)",
                          r"($g(x)$: $N(\mu,\sigma^2)$ on $(c_1,c_2)$)",
                          r"(dotted: cut-off parts)",
                          r"(weight $w(x)$ (right axis))")))
}
par(mar = c(4.5, 4.5, 3, 4.5))
draw_fg(3, 5, TeX(r"($(c_1,c_2)=(1,3)$)"))
```

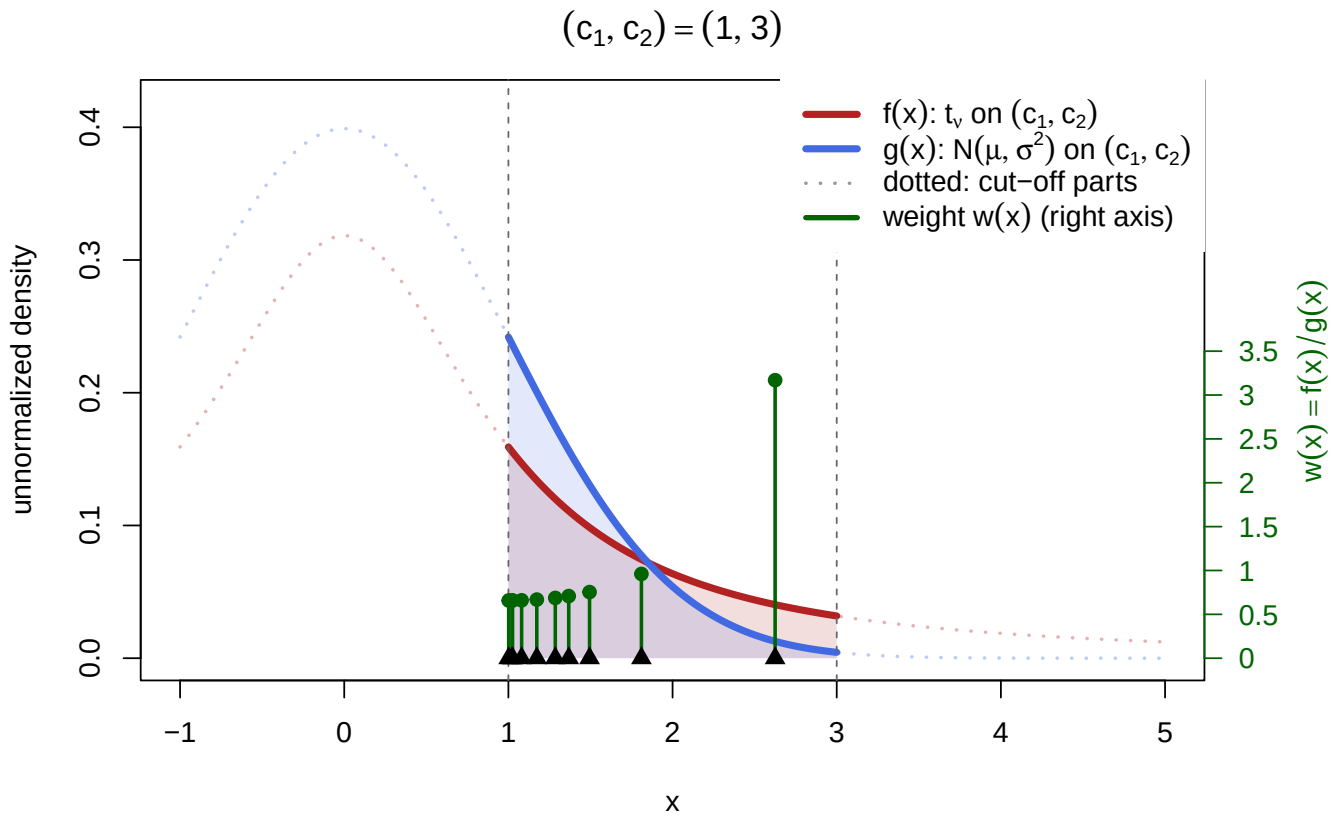


Figure 11.4: Shows the unnormalized target and proposal for a t interval probability, with draws and their weights. $f(x)$: the Cauchy density on $(1, 3)$ (red, area C_f); $g(x)$: the $N(0, 1)$ density on $(1, 3)$ (blue, area C_g); dotted: the parts cut off outside the interval. Black triangles: 10 draws from $g(x)/C_g$; green segments: their weights $w(x) = f(x)/g(x)$ (right axis).

11.4.3 Implementation and Results

`is_t_interval()` turns the estimator into code: draw X from `rtnorm_ars()`, form the log weight $\log f_{t_\nu}(X) - \log g(X)$ with the unnormalized g, and report $\widehat{C_f}$ (scaled by C_g), its standard error, and n_{eff}.

```
## ------------------------------------------------------------------
##  is_t_interval(): importance-sampling estimate of P(c1 < X < c2) for a
##  standard t_df target, using X ~ N(mu, sigma^2) truncated to (c1, c2)
##  as the proposal.
## ------------------------------------------------------------------
is_t_interval <- function(n, df, c1, c2, mu, sigma, seed = NULL) {
    if (!is.null(seed)) set.seed(seed)
    X <- rtnorm_ars(n, mu, sigma, c1, c2)

    Cg   <- pnorm(c2, mu, sigma) - pnorm(c1, mu, sigma)            # known C_g
    logw <- dt(X, df, log = TRUE) - dnorm(X, mu, sigma, log = TRUE) # f/g, unnormalized g

    m    <- max(logw); v <- exp(logw - m)          # rescaled weights
    Chat <- Cg * mean(v) * exp(m)                  # unbiased estimate of P(c1 < X < c2)
    se   <- Cg * sd(v) / sqrt(n) * exp(m)          # standard error
    wn   <- v / mean(v)                            # normalized weights hat w*_i, mean 1

    list(X = X, wn = wn, Chat = Chat, se = se, ess = n / mean(wn^2),
         truth = pt(c2, df) - pt(c1, df))
}
```

Table 11.2 runs it for two intervals, one in the tail and one in the centre, at a heavy ($\nu = 1$) and a lighter ($\nu = 10$) tail.

```
settings <- data.frame(df = c(1, 10, 1, 10), c1 = c(1, 1, -1, -1), c2 = c(3, 3, 2, 2))
res <- lapply(seq_len(nrow(settings)), function(i) with(settings[i, ],
    is_t_interval(n = 5000, df = df, c1 = c1, c2 = c2, mu = 0, sigma = 1, seed = 1)))

knitr::kable(data.frame(
    df       = settings$df,
    c1       = settings$c1,
    c2       = settings$c2,
    truth    = sapply(res, `[[`, "truth"),
    estimate = sapply(res, `[[`, "Chat"),
    se       = sapply(res, `[[`, "se"),
    ess_pct  = 100 * sapply(res, `[[`, "ess") / 5000
), digits = 4, col.names = c("df", "c1", "c2", "truth", "estimate", "s.e.", "ESS %"))
```

Table 11.2: Importance-sampling estimates of $P(c_1 < X < c_2)$ for a standard t target, from n = 5000 draws of the $N(0,1)$ proposal truncated to (c_1, c_2).

df	c1	c2	truth	estimate	s.e.	ESS %
1	1	3	0.1476	0.1490	0.0014	68.7656
10	1	3	0.1638	0.1642	0.0004	96.8918
1	-1	2	0.6024	0.6025	0.0008	99.1570
10	-1	2	0.7929	0.7930	0.0002	99.9562

Every estimate is within a couple of standard errors of the truth, for heavy and light tails alike. The effective sample size is lowest for the Cauchy target on $(1, 3)$, where the weights grow across the interval as in Figure 11.4, and close to n for the central interval, where the normal and t shapes nearly agree.

11.4.4 Shinylive App for Estimating a t Probability with a Normal Sample

There is a shinylive app that shows the target, the proposal and the importance weights for any choice of (c_1, c_2), ν, μ and σ. Things to try:

- **Default settings**: raising n improves the estimate at the usual $1/\sqrt{n}$ rate, and the 95% interval covers about as often as claimed.
- **Small σ, or μ far outside** (c_1, c_2): the draws pile up wherever the proposal has mass, the weights are nearly equal there and n_{eff} stays high, yet the estimate can still miss badly.
- **Large σ**: a single draw's weight can dominate and n_{eff} collapses toward 1, which is visibly unreliable, and so better than being invisibly unreliable.

11.5 Example 3: Same Support, but a Mode Is Missed

11.5.1 The Setup

$$f(\theta) = \tfrac{1}{2}\varphi(\theta + d) + \tfrac{1}{2}\varphi(\theta - d), \qquad g_\sigma(\theta) = \tfrac{1}{\sigma}\varphi\big(\tfrac{\theta+d}{\sigma}\big), \qquad d = 5. \tag{11.31}$$

Both are normalized, so $C_f/C_g = 1$, and $E_f(\theta) = 0$. Both are positive on all of $\mathbb{R}$, so the support requirement holds. But for σ of order one the proposal concentrates around $-d$ and never produces draws near the second mode at $+d$. For θ near $-d$ and $\sigma = 1$,

$$w(\theta) = \frac{f(\theta)}{g_1(\theta)} = \tfrac{1}{2} + \tfrac{1}{2}\,\frac{\varphi(\theta - d)}{\varphi(\theta + d)} = \tfrac{1}{2} + \tfrac{1}{2}e^{2d\theta} \;\approx\; \tfrac{1}{2}, \tag{11.32}$$

so every observed weight is nearly equal, $\hat{w}_i^\star \approx 1$ and $n_{\text{eff}} \approx n$, while $\widehat{C_f/C_g} \approx \frac{1}{2}$ against a truth of 1 and $\widehat{E}_f(\theta) \approx -d$ against a truth of 0. The draws that would expose the error carry $w(\theta) \approx \frac{1}{2}e^{2d\theta} \gg 1$ near $\theta = +d$, and they have probability $\Phi(-2d) \approx 10^{-23}$ under g_1: they never appear.

11.5.2 The Figure

Figure 11.5 draws $f(\theta)$ and $g(\theta)$ on the density scale with 10 draws from $g(\theta)$ and their weights, on a weight axis on which the true value $C_f/C_g = 1$ sits at half the height of the plot. Every draw has weight close to $\frac{1}{2}$, the weights look perfectly balanced, and the red mode at $+5$ receives no draws at all. The text block summarizes a run with $n = 2000$ draws.

```
d <- 5; sg <- 1; n <- 2000
set.seed(11)

## log of the mixture density, computed stably
log_f <- function(th, d) {
    a <- dnorm(th, -d, log = TRUE); b <- dnorm(th, d, log = TRUE)
    m <- pmax(a, b)
    m + log(exp(a - m) + exp(b - m)) - log(2)
}

## exact log E_g(w^2) for this pair; infinite when 2 - 1/sigma^2 <= 0,
## i.e. when sigma <= 1/sqrt(2). Since E_g(w) = 1 exactly, the exact
## effective fraction is 1 / E_g(w^2).
log_Ew2 <- function(d, sg) {
    A <- 2 - 1 / sg^2
    if (A <= 0) return(Inf)
    lterm <- function(a, b) {
        m <- (a + b) / 2
        B <- 2 * m + d / sg^2
        C <- 2 * m^2 - d^2 / sg^2
        dnorm(a, b, sqrt(2), log = TRUE) + log(sg) + 0.5 * log(2) +
            0.5 * log(2 * pi / A) - 0.5 * (C - B^2 / A)
    }
    lv <- c(lterm(-d, -d), log(2) + lterm(-d, d), lterm(d, d))
    m  <- max(lv)
    m + log(sum(exp(lv - m))) - log(4)
}

## ---- sample and weight (n = 2000) ------------------------------------
th   <- rnorm(n, -d, sg)
logw <- log_f(th, d) - dnorm(th, -d, sg, log = TRUE)
mx   <- max(logw); v <- exp(logw - mx)
Chat <- mean(v) * exp(mx); se <- sd(v) / sqrt(n) * exp(mx)   # C_f / C_g, truth 1
wn   <- v / mean(v); ess_obs <- n / mean(wn^2)               # hat w*_i, mean 1
Ehat <- mean(wn * th)                                        # E_f(theta), truth 0
ess_x <- 100 * exp(-log_Ew2(d, sg))                          # exact effective fraction (%)

## ---- picture: densities, 10 draws and their weights ------------------
xs <- seq(-9, 9, length.out = 1000)
fx <- exp(log_f(xs, d)); gx <- dnorm(xs, -d, sg)
ytop <- max(fx, gx) * 1.05

set.seed(48); tp <- rnorm(10, -d, sg)                # 10 draws from g to display
wp <- exp(log_f(tp, d) - dnorm(tp, -d, sg, log = TRUE))
```

```r
sc <- 0.5 * ytop / max(1, wp)                          # w = 1 reaches half the y-axis

par(mar = c(4.5, 4.5, 3, 4.5), cex = 1.15)
plot(xs, fx, type = "n", ylim = c(0, ytop), yaxs = "i", xlab = expression(theta),
     ylab = "density", main = "bimodal target and a proposal centred on one mode")
polygon(c(xs, rev(xs)), c(gx, 0 * xs), col = adjustcolor("royalblue", 0.20), border = NA)
polygon(c(xs, rev(xs)), c(fx, 0 * xs), col = adjustcolor("firebrick", 0.20), border = NA)
lines(xs, gx, col = "royalblue", lwd = 4); lines(xs, fx, col = "firebrick", lwd = 4)
segments(tp, 0, tp, wp * sc, col = "darkgreen", lwd = 2)
points(tp, wp * sc, pch = 19, col = "darkgreen")
points(tp, rep(0, 10), pch = 17, cex = 1.2, xpd = NA)
wt <- pretty(c(0, max(1, wp)))
axis(4, at = wt * sc, labels = wt, las = 1, col.axis = "darkgreen", col = "darkgreen")
mtext(TeX(r"($w(\theta) = f(\theta)/g(\theta)$)"), side = 4, line = 2.8, col = "darkgreen")
legend("topright", bty = "n", lwd = c(4, 4, 2), col = c("firebrick", "royalblue", "darkgreen"),
       legend = TeX(c(r"(target $f(\theta)$)", sprintf(r"(proposal $g(\theta)$, sd $= %.0f$)",
        ↪  sg),
                      r"(10 draws and weights $w(\theta_i)$)")))
legend(-3.7, ytop * 0.8, bty = "n", text.col = "navy", cex = 0.9,
       legend = TeX(c(sprintf(r"(with $n = %d$ draws:)", n),
           sprintf(r"($C_f/C_g$: true 1, est. %.3f, 95%% CI $(%.3f, %.3f)$)", Chat,
                   Chat - 1.96 * se, Chat + 1.96 * se),
           sprintf(r"($E_f(\theta)$: true 0, est. %.2f)", Ehat),
           sprintf(r"(observed $n_{eff}$ = %.1f%% of n; exact $n_{eff}$ = %.3g%% of n)",
                   100 * ess_obs / n, ess_x))))
```

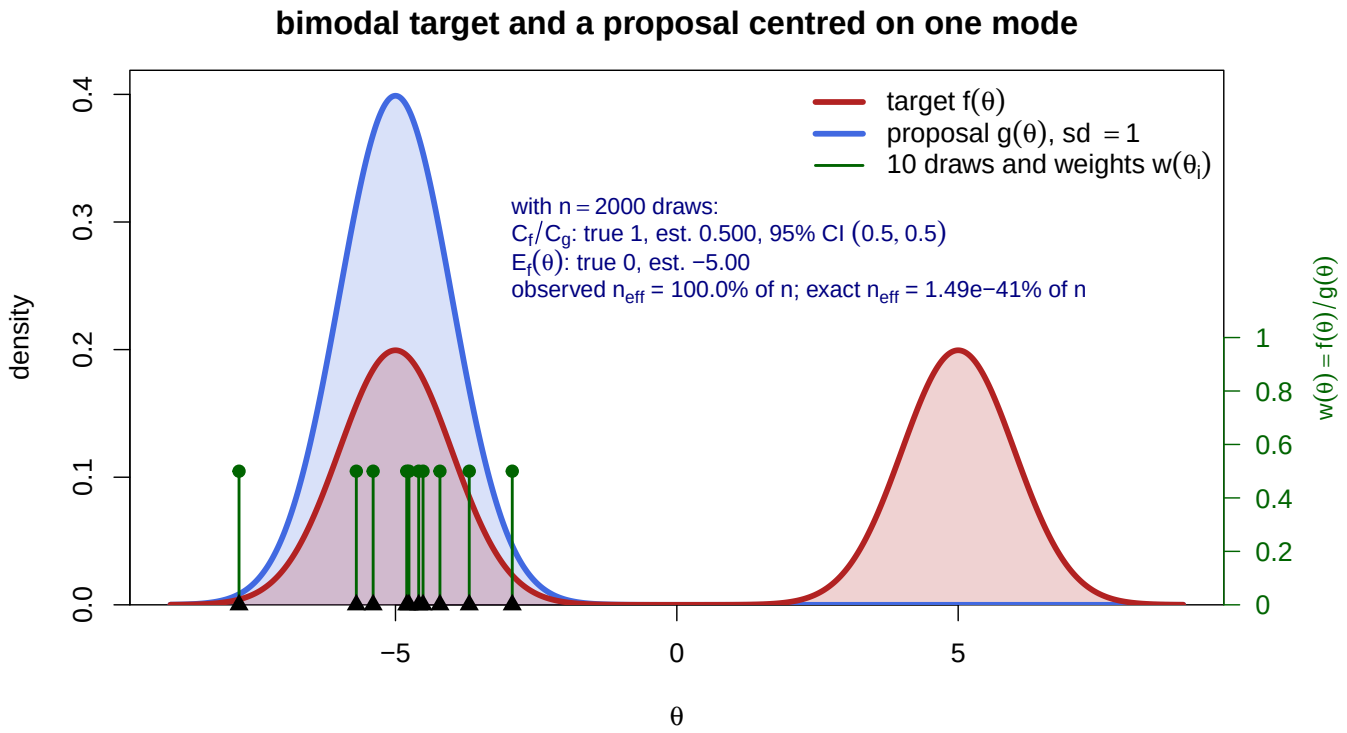


Figure 11.5: Demonstrates a silent failure of importance sampling: when the proposal misses a mode of the target, the weights look well behaved while the estimate is wrong. The bimodal target f (red) and the proposal g (blue), centred on one mode; black triangles are 10 draws from g, and green segments their weights w = f/g (right axis), all close to 1/2. The text gives the estimates and effective sample sizes from n = 2000 draws.

The last line of the text block is the whole lesson. The observed effective sample size is a statistic of the weights that actually appeared, and it reports near-perfect efficiency. The exact effective fraction $n^{\star}_{\text{eff}}/n = 1/E_g[w^{\star}(\theta)^2]$, available here in closed form because both densities are normal mixtures, is smaller by some forty orders of magnitude; it is a property of $f(\theta)$ and $g(\theta)$ alone, invisible to any single sample. At $\sigma = 1$ the variance is finite, so the central limit theorem does apply and the estimator is consistent; it would simply need n of order 10^{43} before its sampling distribution looked normal. Finiteness of the variance is itself fragile: the exact second moment diverges once $2 - \sigma^{-2} \leq 0$, that is, for every $\sigma \leq 1/\sqrt{2}$.

11.5.3 Shinylive App for the Failure of Importance Sampling

There is a shinylive app that varies σ with everything else held fixed. Near $\sigma = 1$ the failure is silent: the observed effective sample size is close to n while the estimate of C_f/C_g sits at one half. Past $\sigma \approx 4$ the proposal reaches the second mode, the observed effective sample size *drops*, which looks like deterioration but is the diagnostic finally working, and the estimate moves onto the truth.

11.6 Example 4: Marginal Likelihood of Normal Data via Importance Sampling

11.6.1 The Target and the Proposal

The quadrature and Laplace-approximation chapters compute the marginal likelihood of the same conjugate-free Gaussian model by a product grid and by a mode-and-Hessian approximation. Importance sampling offers a third route, and one that comes with a built-in accuracy diagnostic, n_{eff}, that neither of the other two methods provides.

The data are $y_1, \dots, y_m \sim N(\mu, \sigma^2)$, with parameter $\theta = (\mu, w)$, $w = \log \sigma^2$, and independent normal priors $\mu \sim N(\mu_0, \sigma_\mu^2)$ and $w \sim N(w_0, \sigma_w^2)$. The **target** is prior $\times$ likelihood, whose normalizing constant is the marginal likelihood, $C_f = P(y)$:

$$f(\mu, w) = N(\mu \mid \mu_0, \sigma_\mu^2)\; N(w \mid w_0, \sigma_w^2) \prod_{k=1}^{m} N(y_k \mid \mu,\, e^w). \tag{11.33}$$

The **proposal** is the Gaussian (Laplace) approximation to the posterior. Locate the mode $(\hat{\mu}, \hat{w})$ of f by optimization, let $H = -\nabla^2 \log f(\hat{\mu}, \hat{w})$ be the Hessian there, and take

$$g(\mu, w) = N_2\big((\mu, w) \,\big|\, (\hat{\mu}, \hat{w}),\; H^{-1}\big). \tag{11.34}$$

Since g is a normalized density, $C_g = 1$ and

$$\widehat{P(y)} = \frac{1}{n} \sum_{i=1}^{n} \frac{f(\theta_i)}{g(\theta_i)}, \qquad \theta_i \sim g(\theta). \tag{11.35}$$

Because g is shaped like the posterior near its mode, it is a good proposal by construction. The contrast is sampling directly from the (typically much wider) **prior**, which is importance sampling with $g(\theta) = \pi(\theta)$: its weights are just the likelihood values $L(\theta_i)$.

11.6.2 The Contours of f and g

Figure 11.6 shows why the weights are still needed. With only $m = 5$ observations the posterior is skewed in $w = \log \sigma^2$, with a long upper tail, and it fans out in μ as σ^2 grows, so its contours are not the ellipses of the quadratic approximation g. The weights f/g correct for that difference. As m grows, f becomes more nearly Gaussian and g fits better.

```
local({
set.seed(2); m <- 5; y <- rnorm(m)
mu0 <- 0; smu <- 5; w0 <- 0; sw <- 5
logf <- function(mu, w) dnorm(mu, mu0, smu, log = TRUE) + dnorm(w, w0, sw, log = TRUE) +
    sapply(seq_along(mu), function(i) sum(dnorm(y, mu[i], exp(w[i] / 2), log = TRUE)))
fit <- optim(c(mean(y), log(var(y))), function(t) -logf(t[1], t[2]), hessian = TRUE)
md <- fit$par; S <- solve(fit$hessian)
mus <- seq(md[1] - 3.2, md[1] + 3.2, length.out = 260)
ws  <- seq(md[2] - 2.8, md[2] + 5.0, length.out = 260)
G   <- expand.grid(mu = mus, w = ws)
```

```
lf  <- matrix(logf(G$mu, G$w), length(mus)) ; lf <- lf - max(lf)
Si  <- solve(S); D <- cbind(G$mu - md[1], G$w - md[2])
lg  <- matrix(-0.5 * rowSums((D %*% Si) * D), length(mus))
lev <- -qchisq(c(0.5, 0.9, 0.99), 2) / 2       # same density levels for both
par(mar = c(4.5, 4.5, 1, 1), cex = 1.2)
plot(NA, xlim = range(mus), ylim = range(ws), xlab = expression(mu),
     ylab = expression(w == log ~ sigma^2))
contour(mus, ws, lg, levels = lev, drawlabels = FALSE, col = "royalblue", lwd = 2.5, add =
 ↪  TRUE)
contour(mus, ws, lf, levels = lev, drawlabels = FALSE, col = "firebrick", lwd = 2.5, add =
 ↪  TRUE)
points(md[1], md[2], pch = 3, cex = 1.6, lwd = 2)
legend("topright", bty = "n", lwd = 2.5, col = c("firebrick", "royalblue"),
       legend = c("target f (posterior)", "proposal g (Laplace)"))
})
```

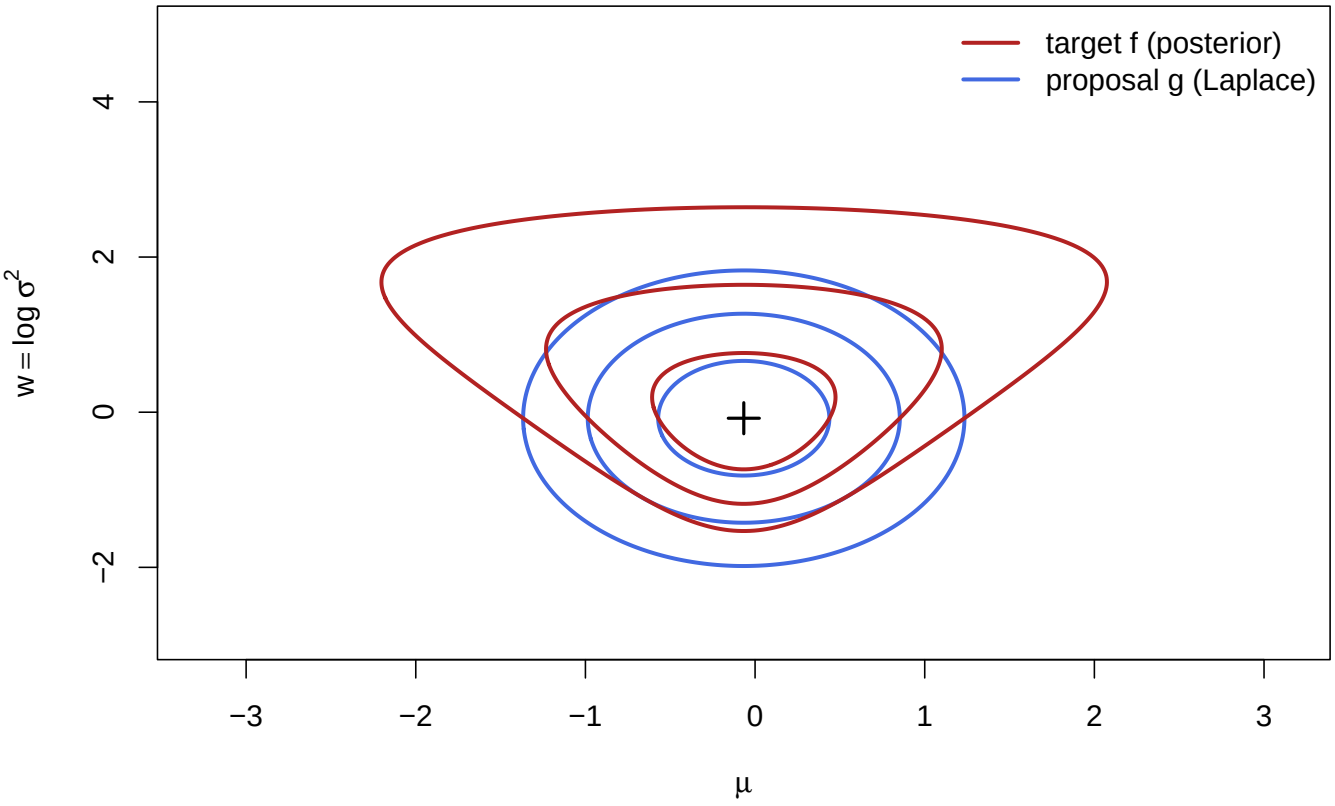


Figure 11.6: Shows that the posterior is not quadratic on the log scale while its Laplace approximation is. Contours of the target f (posterior, red) and of the Gaussian proposal g (blue) at the same three density levels, for m = 5 simulated observations with $N(0, 5^2)$ priors on μ and w; the cross marks the shared mode.

11.6.3 Implementation

```
log_lik   <- function(x, mu, w) sum(dnorm(x, mu, exp(w / 2), log = TRUE))
log_prior <- function(mu, w, mu_0, sigma_mu, w_0, sigma_w) {
    dnorm(mu, mu_0, sigma_mu, log = TRUE) + dnorm(w, w_0, sigma_w, log = TRUE)
}
neg_log_post <- function(theta, x, mu_0, sigma_mu, w_0, sigma_w) {
    -log_lik(x, theta[1], theta[2]) -
        log_prior(theta[1], theta[2], mu_0, sigma_mu, w_0, sigma_w)
}

### log density of N(mu, A^{-1}) at every column of a 2 x n matrix of points at once,
### replacing a per-point apply() call with one vectorized quadratic form
log_dmvnorm_batch <- function(Theta, mu, A) {
    Tc <- Theta - mu
    quad <- colSums((A %*% Tc) * Tc)
    0.5 * (-nrow(Theta) * log(2 * pi) + sum(log(svd(A)$d)) - quad)
}

### log marginal likelihood by importance sampling from the Laplace Gaussian.
### Besides n_eff, attaches "se": the standard error of the *log* estimate,
### sd(v) / (sqrt(n) * mean(v)) for v = exp(lw - max(lw)) -- the delta-method
### transfer of the raw estimator's SE onto the log scale.
log_marlik_imps <- function(x, mu_0, sigma_mu, w_0, sigma_w, n) {
    fit <- nlm(neg_log_post, c(mean(x), log(stats::var(x))), hessian = TRUE,
               x = x, mu_0 = mu_0, sigma_mu = sigma_mu, w_0 = w_0, sigma_w = sigma_w)
    mu_hat <- fit$estimate; H <- fit$hessian
    L <- t(chol(solve(H)))                          # Sigma = H^{-1} = L L'
    Theta <- L %*% matrix(rnorm(2 * n), 2, n) + mu_hat

    log_g <- log_dmvnorm_batch(Theta, mu_hat, H)
    log_f <- -apply(Theta, 2, neg_log_post, x = x, mu_0 = mu_0, sigma_mu = sigma_mu,
                    w_0 = w_0, sigma_w = sigma_w)
    lw <- log_f - log_g
    m  <- max(lw); v <- exp(lw - m)
    structure(log_sum_exp(lw) - log(n),
              ess = log_ess(lw), se = sd(v) / (sqrt(n) * mean(v)))
}

### log marginal likelihood by sampling from the prior (g = pi): a second
### importance-sampling estimator, with an unfavourable g
log_marlik_prior <- function(x, mu_0, sigma_mu, w_0, sigma_w, n) {
    mus <- rnorm(n, mu_0, sigma_mu); ws <- rnorm(n, w_0, sigma_w)
    lw <- vapply(seq_len(n), function(i) log_lik(x, mus[i], ws[i]), numeric(1))
    m  <- max(lw); v <- exp(lw - m)
    structure(log_sum_exp(lw) - log(n),
              ess = log_ess(lw), se = sd(v) / (sqrt(n) * mean(v)))
}

### "truth": log marginal likelihood by a midpoint grid over (mu, w), centred
```

```
### and scaled by the Laplace fit. The likelihood term is vectorized over the
### whole mu-grid at once via the sufficient statistic sum((x - mu)^2).
log_marlik_grid <- function(x, mu_0, sigma_mu, w_0, sigma_w, n_grid = 400, width = 8) {
    m   <- length(x)
    fit <- nlm(neg_log_post, c(mean(x), log(stats::var(x))), hessian = TRUE,
               x = x, mu_0 = mu_0, sigma_mu = sigma_mu, w_0 = w_0, sigma_w = sigma_w)
    se  <- sqrt(diag(solve(fit$hessian)))
    mu_grid <- fit$estimate[1] + seq(-width, width, length.out = n_grid) * se[1]
    w_grid  <- fit$estimate[2] + seq(-width, width, length.out = n_grid) * se[2]

    SS <- sum(x^2) - 2 * mu_grid * sum(x) + m * mu_grid^2   # sum((x - mu)^2)
    log_lik_grid   <- outer(SS, w_grid,
                            function(ss, w) -0.5 * m * log(2 * pi) - m * w / 2 - 0.5 * exp(-w)
                             ↪  * ss)
    log_prior_grid <- outer(dnorm(mu_grid, mu_0, sigma_mu, log = TRUE),
                            dnorm(w_grid, w_0, sigma_w, log = TRUE), "+")

    h_mu <- diff(mu_grid[1:2]); h_w <- diff(w_grid[1:2])
    log_sum_exp(as.vector(log_lik_grid + log_prior_grid)) + log(h_mu) + log(h_w)
}
```

`log_marlik_grid()` supplies the ground truth by quadrature: a midpoint grid over (μ, w), centred and scaled by the same Laplace fit used for the proposal, and wide enough (8 posterior standard deviations each way, on a 400×400 grid) that refining it further does not move the answer.

11.6.4 Results Across Sample Sizes

Both proposals target the same $f = L\pi$, and the prior is a legitimate proposal for it: every posterior draw has positive prior density, so the support condition holds. The two differ only in how well their shape matches the target's, and Table 11.3 varies the amount of data m to make that difference visible.

```
set.seed(4)
ms <- c(5, 20, 100, 400)
n  <- 2000

## one proposal, one data set: estimate, interval and n_eff as a % of n
summarize_is <- function(fn, x, n) {
    est <- fn(x, 0, 5, 0, 5, n)
    se  <- attr(est, "se")
    c(estimate = sprintf("%.2f", as.numeric(est)),
      ci       = sprintf("(%.2f, %.2f)",
                         as.numeric(est) - 1.96 * se,
                         as.numeric(est) + 1.96 * se),
      ess      = sprintf("%.1f", 100 * attr(est, "ess") / n))
}

merged_tbl <- do.call(rbind, lapply(ms, function(m) {
    x   <- rnorm(m)
```

Table 11.3: Compares importance sampling proposals for estimating a marginal likelihood. Log marginal likelihood for a freshly simulated data set at each data size m: truth by grid quadrature, against single runs of importance sampling (n = 2000 draws) from the Laplace Gaussian and from the prior.

		Laplace proposal			prior sampling		
m	truth	estimate	95% CI	ESS %	estimate	95% CI	ESS %
5	-10.09	-10.16	(-10.21, -10.11)	43.3	-10.00	(-10.31, -9.69)	1.9
20	-37.29	-37.31	(-37.33, -37.29)	84.9	-36.86	(-37.32, -36.40)	0.9
100	-153.02	-153.01	(-153.02, -153.01)	96.3	-153.94	(-155.12, -152.76)	0.1
400	-531.18	-531.18	(-531.18, -531.18)	99.2	-534.74	(-536.43, -533.05)	0.1

```
    lap <- summarize_is(log_marlik_imps,  x, n)
    pri <- summarize_is(log_marlik_prior, x, n)
    data.frame(m     = m,
               truth = sprintf("%.2f", log_marlik_grid(x, 0, 5, 0, 5)),
               lap_estimate = lap[["estimate"]],
               lap_ci       = lap[["ci"]],
               lap_ess      = lap[["ess"]],
               pri_estimate = pri[["estimate"]],
               pri_ci       = pri[["ci"]],
               pri_ess      = pri[["ess"]],
               row.names    = NULL)
}))

## repeated labels are intentional: the proposal is carried by the spanner
gt::gt(merged_tbl) |>
    gt::tab_spanner("Laplace proposal", c(lap_estimate, lap_ci, lap_ess)) |>
    gt::tab_spanner("prior sampling", c(pri_estimate, pri_ci, pri_ess)) |>
    gt::cols_label(lap_estimate = "estimate", lap_ci = "95% CI", lap_ess = "ESS %",
                   pri_estimate = "estimate", pri_ci = "95% CI", pri_ess = "ESS %") |>
    gt::cols_align("right")
```

The Laplace proposal re-centres and re-scales itself around the posterior mode for every data set, and it improves as m grows: n_{eff} rises from about 43% of n at $m = 5$ to 99% at $m = 400$, and the estimate agrees with the truth to two decimals from $m = 20$ on. At $m = 5$ the posterior is visibly non-Gaussian (Figure 11.6), the weights vary more, and the single run is off by 0.07 on the log scale, with an interval that just misses the truth. Prior sampling degrades monotonically. The posterior concentrates at rate $m^{-1/2}$ while the prior does not move, so the fraction of prior mass lying where the likelihood is appreciable shrinks with m, the weights become ever more concentrated on a handful of draws, and n_{eff} falls to a small fraction of one percent. Its interval may still cover the truth, but it is only as trustworthy as those few draws.

Note the contrast in how the two failures in this chapter announce themselves. Prior sampling fails *noisily*: n_{eff} reports the collapse directly, which is exactly what the diagnostic is built to catch. The missed mode of Example 3 fails *silently*, with n_{eff} near n while the estimate is badly wrong, because there the weights are uniform among the draws actually observed and the trouble lies entirely outside the region the proposal ever visits.

11.7 Example 5: Choosing the Number of Predictors in Logistic Regression

The marginal likelihood can compare models of different dimension, because integrating over the coefficients penalizes parameters the data do not need. Section 9.2.5 computed it for a sequence of nested logistic regressions with a Laplace-adapted product grid, whose k^p points force k down to 2 at $p = 20$. Importance sampling with the Laplace approximation as the proposal, as in Example 4, has no such limit: its cost is n draws whatever p is, and n_{eff} reports how well the proposal fits in each dimension.

Data and models. There are $m = 200$ observations with 20 candidate predictors $x_{k1}, \dots, x_{k,20} \overset{iid}{\sim} N(0,1)$ and

$$y_k \mid x_k \sim \text{Bernoulli}(\,\text{logit}^{-1}(x_{k1} - x_{k2})), \tag{11.36}$$

so only the first two predictors carry signal; there is no intercept. Model M_p uses the first p predictors, $p = 1, 2, 3, 4, 5, 10, 20$, so M_2 is the true model. With $\eta_k = \sum_{j=1}^{p} x_{kj}\beta_j$,

$$f_p(\beta) = \pi(\beta)\, L_p(\beta) = \prod_{j=1}^{p} N(\beta_j \mid 0,\, \sigma_\beta^2) \prod_{k=1}^{m} \frac{e^{y_k \eta_k}}{1 + e^{\eta_k}}, \qquad P(y \mid M_p) = \int_{\mathbb{R}^p} f_p(\beta)\, d\beta, \tag{11.37}$$

for two prior scales, $\sigma_\beta = 1$ and $\sigma_\beta = 10$.

Laplace proposal. For each model, find the posterior mode $\hat\beta$ and the Hessian H of $-\log f_p$ there, and take $g = N_p(\hat\beta, H^{-1})$. With the Cholesky factorization $H = R^\top R$, a draw is $\beta_i = \hat\beta + R^{-1} z_i$ with $z_i \sim N_p(0, I)$, and its log density is

$$\log g(\beta_i) = \sum_{j=1}^{p} \log \phi(z_{ij}) + \sum_{j=1}^{p} \log R_{jj}. \tag{11.38}$$

The estimate and its log-scale standard error are those of Example 4,

$$\widehat{P(y \mid M_p)} = \frac{1}{n} \sum_{i=1}^{n} w_i, \qquad w_i = \frac{f_p(\beta_i)}{g(\beta_i)}, \tag{11.39}$$

computed from the log weights. The Laplace approximation itself is $\log f_p(\hat\beta) + \frac{p}{2}\log(2\pi) - \sum_j \log R_{jj}$, which is the value $\widehat{P(y \mid M_p)}$ would take if every weight equalled the one at $z = 0$; importance sampling corrects it by averaging the weights over the whole proposal.

```
### data: 20 candidate predictors, only x1 and x2 carry signal
set.seed(2024)
m_lr <- 200; p_max <- 20
X_lr <- matrix(rnorm(m_lr * p_max), m_lr, p_max)
y_lr <- rbinom(m_lr, 1, plogis(drop(X_lr %*% c(1, -1, rep(0, p_max - 2)))))

### log P(y | M_p) by importance sampling from the Laplace normal N_p(beta_hat, H^{-1});
### also returns the Laplace approximation, the log-scale SE and n_eff as % of n
log_ml_logit_is <- function(p, sigma_beta, n) {
  Xp <- X_lr[, seq_len(p), drop = FALSE]
  log_f_rows <- function(B) {                    # log f_p at every row of B (n x p)
    E <- Xp %*% t(B)                             # m x n linear predictors
    colSums(y_lr * E - log1p(exp(E))) + rowSums(dnorm(B, 0, sigma_beta, log = TRUE))
```

```
  }
  fit <- optim(rep(0, p), function(b) -log_f_rows(matrix(b, 1)),
               method = "BFGS", hessian = TRUE)
  R <- chol(fit$hessian)                          # H = R'R, so H^{-1} = R^{-1} R^{-T}
  Z <- matrix(rnorm(n * p), n, p)
  B <- sweep(t(backsolve(R, t(Z))), 2, fit$par, "+")   # beta_i = beta_hat + R^{-1} z_i
  log_g <- rowSums(dnorm(Z, log = TRUE)) + sum(log(diag(R)))
  lw <- log_f_rows(B) - log_g
  v  <- exp(lw - max(lw))
  c(laplace = -fit$value + p / 2 * log(2 * pi) - sum(log(diag(R))),
    is      = log_sum_exp(lw) - log(n),
    se      = sd(v) / (sqrt(n) * mean(v)),
    ess     = 100 * log_ess(lw) / n)
}

p_vals  <- c(1, 2, 3, 4, 5, 10, 20)
sb_vals <- c(1, 10)
n_lr    <- 5000
set.seed(11)
is_res <- lapply(sb_vals, function(sb) t(sapply(p_vals, log_ml_logit_is, sigma_beta = sb, n =
↪  n_lr)))
names(is_res) <- sb_vals

### BIC = -2 log L(beta_mle) + p log m and AIC = -2 log L(beta_mle) + 2 p, on the same
### scale as -2 log P(y | M_p). Neither depends on the prior.
loglik_mle <- sapply(p_vals, function(p)
  as.numeric(logLik(glm(y_lr ~ X_lr[, seq_len(p), drop = FALSE] - 1, family = binomial))))
ic <- cbind(bic = -2 * loglik_mle + p_vals * log(m_lr),
            aic = -2 * loglik_mle + 2 * p_vals,
            dev = -2 * loglik_mle)                 # training loss, no penalty
```

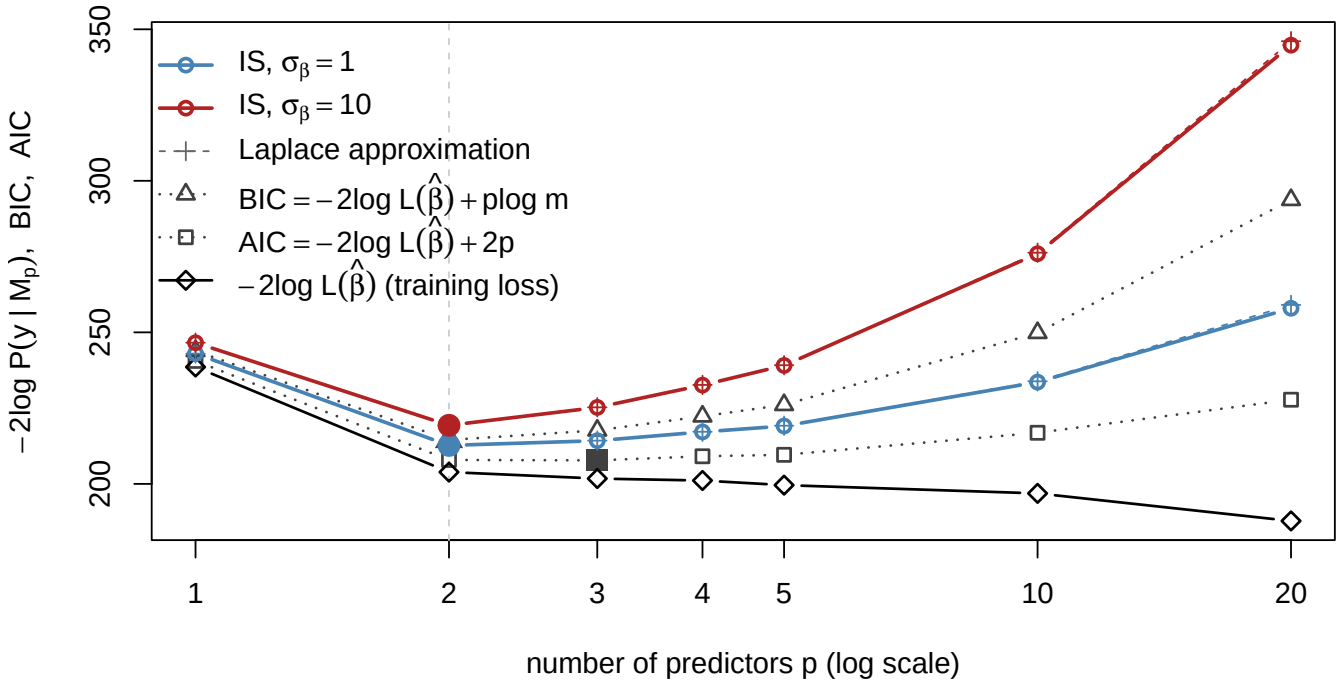


Figure 11.7: Shows that the marginal likelihood, estimated by importance sampling from the Laplace approximation, picks the true number of predictors in a logistic regression, for two prior scales. $-2 \log P(y \mid M_p)$ for models using the first p of 20 predictors ($m = 200$ observations; only x_1 and x_2 carry signal), with prior $\beta_j \sim N(0, \sigma_\beta^2)$ for $\sigma_\beta = 1$ and 10. Solid coloured lines with circles: importance sampling with $n = 5000$ draws from $N_p(\hat{\beta}, H^{-1})$; their 95% intervals (in the table below) are narrower than the plotted points, so they are not drawn. Dashed coloured lines with plus signs: the Laplace approximation alone; it differs from importance sampling by at most about 1.3 (at $p = 20$), so each plus sign sits on or next to its circle. Dotted gray lines: BIC $= -2 \log L(\hat{\beta}) + p \log m$ (triangles) and AIC $= -2 \log L(\hat{\beta}) + 2p$ (squares), computed from the maximum likelihood fit and the same for both priors. Black line with diamonds: the training loss $-2 \log L(\hat{\beta})$ with no penalty, which keeps decreasing as predictors are added and so always favours the largest model. On every other curve the filled symbol marks the minimum. Both marginal-likelihood curves and BIC are smallest at $p = 2$, and the marginal likelihood rises faster for the more diffuse prior. AIC, with its lighter penalty of 2 per parameter, is smallest at $p = 3$, but only by 0.14 (207.76 against 207.90 at $p = 2$).

```
fmt_block <- function(r) data.frame(
  laplace = sprintf("%.3f", r[, "laplace"]),
  is      = sprintf("%.3f", r[, "is"]),
  ci      = sprintf("(%.3f, %.3f)", r[, "is"] - 1.96 * r[, "se"], r[, "is"] + 1.96 * r[,
   ↪  "se"]),
  ess     = sprintf("%.1f", r[, "ess"]))
tb <- cbind(p = p_vals, setNames(fmt_block(is_res[[1]]), paste0(c("laplace", "is", "ci",
 ↪  "ess"), "_1")),
                        setNames(fmt_block(is_res[[2]]), paste0(c("laplace", "is", "ci",
                         ↪  "ess"), "_10")))
gt::gt(tb) |>
```

Table 11.4: log P(y | M_p) for the logistic regression models in the figure above: the Laplace approximation, the importance-sampling estimate (n = 5000 draws) with its 95% interval, and n_eff as a percentage of n, for the two prior scales.

	σβ = 1				σβ = 10			
p	Laplace	IS	95% CI	ESS %	Laplace	IS	95% CI	ESS %
1	-121.496	-121.493	(-121.495, -121.490)	99.2	-123.295	-123.289	(-123.292, -123.286)	98.7
2	-106.356	-106.347	(-106.352, -106.342)	96.8	-109.668	-109.663	(-109.669, -109.658)	96.2
3	-107.139	-107.124	(-107.130, -107.119)	95.9	-112.630	-112.617	(-112.625, -112.610)	93.1
4	-108.603	-108.574	(-108.581, -108.567)	93.8	-116.324	-116.293	(-116.302, -116.285)	91.0
5	-109.594	-109.548	(-109.557, -109.539)	91.1	-119.579	-119.540	(-119.549, -119.530)	90.1
10	-116.924	-116.791	(-116.807, -116.776)	76.5	-138.141	-137.974	(-137.997, -137.951)	59.9
20	-129.527	-128.973	(-129.011, -128.936)	35.4	-173.029	-172.382	(-172.432, -172.333)	23.9

```
gt::tab_spanner(gt::md(" <sub> </sub> = 1"),  c(laplace_1, is_1, ci_1, ess_1)) |>
gt::tab_spanner(gt::md(" <sub> </sub> = 10"), c(laplace_10, is_10, ci_10, ess_10)) |>
gt::cols_label(laplace_1 = "Laplace", is_1 = "IS", ci_1 = "95% CI", ess_1 = "ESS %",
               laplace_10 = "Laplace", is_10 = "IS", ci_10 = "95% CI", ess_10 = "ESS %") |>
gt::cols_align("right")
```

```
Warning: HTML tags found, and they will be removed.
* Set `options(gt.html_tag_check = FALSE)` to disable this check.
HTML tags found, and they will be removed.
* Set `options(gt.html_tag_check = FALSE)` to disable this check.
```

On the $-2 \log P(y \mid M_p)$ scale of the figure, both curves drop sharply from $p = 1$ to $p = 2$, because the model with only x_1 misses a real effect, reach their minimum at the true $p = 2$, and then rise as useless predictors are added: each extra coefficient improves the fit only slightly, while its prior spreads probability over values the data do not support. The rise is steeper for $\sigma_\beta = 10$, so the prior scale, which hardly matters for estimating β, matters a great deal for comparing models.

The figure plots $-2 \log P(y \mid M_p)$, so smaller is better, together with the two information criteria on the same scale,

$$\text{BIC}_p = -2 \log L_p(\hat{\beta}) + p \log m, \qquad \text{AIC}_p = -2 \log L_p(\hat{\beta}) + 2p, \tag{11.40}$$

with $\hat{\beta}$ the maximum likelihood estimate. Both start from the training loss $-2 \log L_p(\hat{\beta})$, also drawn in the figure, which can only decrease as predictors are added because each model contains the one before it; on its own it always picks $p = 20$. The criteria add a penalty that grows with p, and they need no prior and no integration. BIC is the large-m simplification of the Laplace approximation (Section 9.3.1): it keeps -2 times the maximized log likelihood and the $p \log m$ term and drops everything of order one, including the prior, so its curve has the same shape as $-2 \log P(y \mid M_p)$ and lies between the two prior scales. It also picks $p = 2$. AIC charges only 2 per parameter instead of $\log m \approx 5.3$, so its curve rises slowly; here it narrowly prefers $p = 3$, the usual tendency of AIC to admit a spurious predictor when the extra fit is small. AIC targets prediction rather than the probability of the model, so the two criteria answer different questions.

The table also shows how the Laplace proposal degrades with dimension. For small p the posterior is close to normal, n_{eff} is above 90% of n, and the Laplace approximation and the importance-sampling estimate agree to two or three decimals. As p grows, the small departures of each coordinate from normality compound: n_{eff} falls

to between a quarter and a third of n at $p = 20$, and the Laplace approximation is too low by about 0.6 on the log scale. That error is far smaller than the differences between models, so here it does not change the choice of p, but importance sampling both corrects it and measures it, which the Laplace approximation alone cannot do. Unlike the prior-sampling failure of Example 4, this decline is gradual and announced by n_{eff}; a heavier-tailed proposal such as a multivariate t centred at $\widehat{\beta}$ with scale H^{-1} would slow it further.

11.8 Advanced Applications of Importance Sampling

11.8.1 Free Energy and the Marginal Likelihood

Statistical physics supplies a second reading of the same estimator. A system with energy $U(x)$ at inverse temperature β has normalizing constant (the partition function) and free energy

$$Z(\beta) = \int e^{-\beta U(x)}\, dx, \qquad F(\beta) = -\frac{1}{\beta}\log Z(\beta). \tag{11.41}$$

$e^{-\beta U(x)}/Z(\beta)$ is the Boltzmann distribution at temperature $1/\beta$: large β (low temperature) concentrates mass where U is small, and small β (high temperature) spreads it out. Importance sampling with

$$g(x) = e^{-\beta_0 U(x)}, \qquad f(x) = e^{-\beta_1 U(x)}, \qquad w(x) = \frac{f(x)}{g(x)} = e^{-(\beta_1-\beta_0)U(x)} \tag{11.42}$$

gives the ratio of normalizing constants as an expectation under the β_0-tempered distribution,

$$\frac{Z(\beta_1)}{Z(\beta_0)} = E_{\beta_0}\Big[e^{-(\beta_1-\beta_0)U(x)}\Big]. \tag{11.43}$$

The Bayesian analogue. Set $U(\theta) = -\log L(\theta)$ and include the prior: $g(\theta) = \pi(\theta)$ at $\beta_0 = 0$ and $f(\theta) = L(\theta)\pi(\theta)$ at $\beta_1 = 1$. The ratio of normalizing constants is the marginal likelihood,

$$P(y) = \frac{C_f}{C_g} = E_\pi\big[L(\theta)\big] \approx \frac{1}{n}\sum_{i=1}^{n} L(\theta_i), \qquad \theta_i \sim \pi(\theta), \tag{11.44}$$

and $-\log P(y)$ is, in this language, the free energy of the posterior. This is not a new estimator: it is the prior sampling of Example 4. The physics vocabulary names its failure precisely: one jump from $\beta = 0$ to $\beta = 1$ asks a single importance-sampling step to bridge the *entire* temperature range, and almost all prior draws get negligible weight.

11.8.2 Stepping Stones: Many Small Jumps Instead of One Big One

The fix follows from the algebra of the ratio itself. Chain temperatures $0 = \beta_0 < \beta_1 < \cdots < \beta_K = 1$ and estimate each factor by importance sampling from the tempered distribution at β_{k-1} to the one at β_k:

$$\frac{Z(1)}{Z(0)} = \prod_{k=1}^{K} \frac{Z(\beta_k)}{Z(\beta_{k-1})}. \tag{11.45}$$

If the temperatures are close together, neighbouring tempered distributions nearly coincide, and every step is a well-matched proposal with near-constant weights. This is the **stepping-stones** estimator, a discretization of thermodynamic integration and the same schedule idea that underlies simulated annealing and parallel tempering.

Figure 11.8 shows the idea with the volumes of Example 1. Instead of jumping from B straight to the small set A, step through nested sets $B = A_0 \supset A_1 \supset A_2 \supset A_3 = A$: each ratio $p_k = |A_k|/|A_{k-1}|$ is large and easy to estimate by drawing uniformly on A_{k-1} and counting hits in A_k, and their product gives $p = |A|/|B|$. Tempering does the same with densities, with A_k replaced by $L(\theta)^{\beta_k}\pi(\theta)$.

```
local({
base <- function(t) 0.30 + 0.07 * sin(3 * t) + 0.05 * cos(2 * t)  # same blob shape as Example
 ↪ 1
sc   <- c(1, 0.55, 0.30)                                          # A_1, A_2, A_3 = A
ctr  <- list(c(0.50, 0.50), c(0.56, 0.54), c(0.60, 0.56))
area <- sapply(sc, function(k) integrate(function(t) (k * base(t))^2 / 2, 0, 2 * pi)$value)
ar   <- c(1, area)                                                # |A_0| = |B| = 1
cols <- c("grey92", "#d9e8f5", "#9cc3e6", "#4a86c5")
tt <- seq(0, 2 * pi, length.out = 400)
par(mar = c(0.5, 0.5, 0.5, 0.5))
plot(NA, xlim = c(0, 2.05), ylim = c(0, 1), asp = 1, axes = FALSE, xlab = "", ylab = "")
rect(0, 0, 1, 1, col = cols[1], border = "black", lwd = 2)
for (k in 1:3) {
    r <- sc[k] * base(tt)
    polygon(ctr[[k]][1] + r * cos(tt), ctr[[k]][2] + r * sin(tt), col = cols[k + 1],
            border = "steelblue4", lwd = 2)
}
text(0.07, 0.93, expression(A[0] == B), cex = 1.6, adj = 0)
text(0.27, 0.30, expression(A[1]), cex = 1.6)
text(0.47, 0.44, expression(A[2]), cex = 1.6)
text(ctr[[3]][1], ctr[[3]][2], expression(A[3] == A), cex = 1.3, col = "white")
y0 <- 0.86
text(1.08, y0, adj = 0, cex = 1.3, "one big jump:")
text(1.12, y0 - 0.09, adj = 0, cex = 1.3,
     TeX(sprintf(r"($p = |A|/|B| = %.3f$)", ar[4])))
text(1.08, y0 - 0.25, adj = 0, cex = 1.3, "three small jumps:")
for (k in 1:3)
    text(1.12, y0 - 0.25 - 0.09 * k, adj = 0, cex = 1.3,
         TeX(sprintf(r"($p_%d = |A_%d|/|A_%d| = %.2f$)", k, k, k - 1, ar[k + 1] / ar[k])))
text(1.08, y0 - 0.70, adj = 0, cex = 1.3, TeX(r"($p = p_1 \, p_2 \, p_3$)"))
})
```

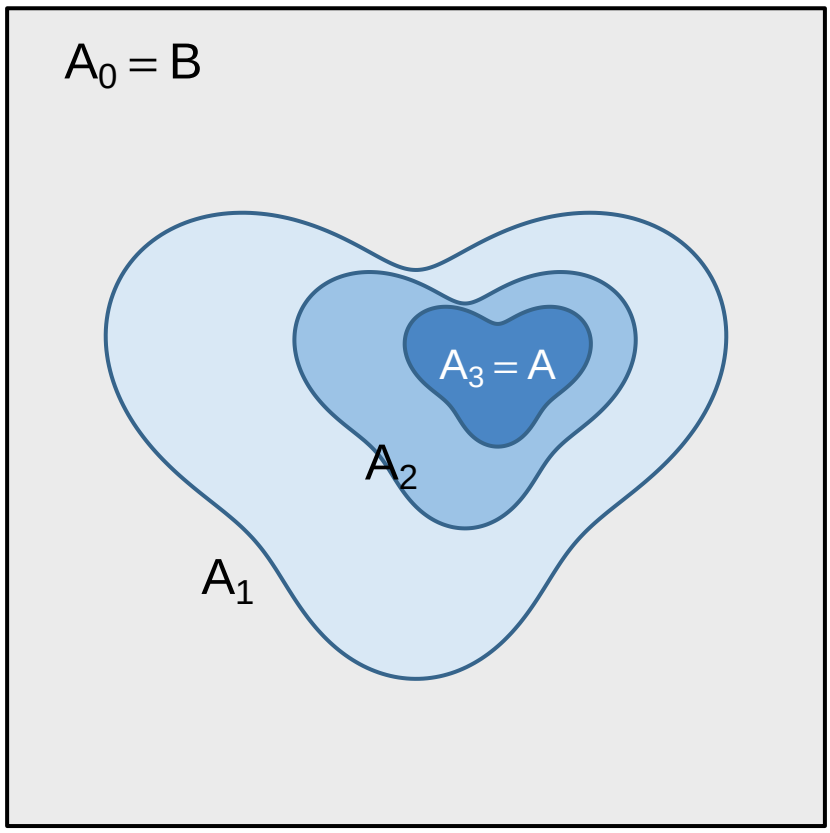


Figure 11.8: Illustrates stepping stones with nested sets. One big jump estimates the small ratio p = |A|/|B| directly; three small jumps estimate the large ratios p_k = |A_k|/|A_{k-1}| and multiply them.

11.8.3 Bayesian Leave-One-Out Cross-Validation

Cross-validation asks how well a model predicts observations it was not fitted to. The Bayesian version of a single leave-one-out fold is the **leave-one-out predictive density**

$$p(y_j \mid y_{-j}) = \int p(y_j \mid \theta)\, p(\theta \mid y_{-j})\, d\theta, \tag{11.46}$$

the posterior predictive density of y_j under the posterior that never saw y_j. Summed in log over $j = 1, \dots, m$, these give the expected log pointwise predictive density, a standard measure of a model's out-of-sample fit. Computed directly, it requires refitting the model once per left-out observation, which is exactly the cost importance sampling avoids.

With prior $\pi(\theta)$ and independent observations $y_1, \dots, y_m$, both posteriors involved are prior $\times$ likelihood (unnormalized):

- **Target**, the posterior without y_j:

$$f(\theta) = \pi(\theta) \prod_{k \neq j} p(y_k \mid \theta). \tag{11.47}$$

- **Proposal**, the full posterior, whose draws $\theta_1, \dots, \theta_n$ we already have from a single fit:

$$g(\theta) = \pi(\theta) \prod_{k=1}^{m} p(y_k \mid \theta). \tag{11.48}$$

- **Weight**: the prior and every likelihood factor except the one for y_j cancel,

$$w_j(\theta) = \frac{f(\theta)}{g(\theta)} = \frac{\pi(\theta) \prod_{k \neq j} p(y_k \mid \theta)}{\pi(\theta) \prod_{k=1}^{m} p(y_k \mid \theta)} = \frac{1}{p(y_j \mid \theta)}. \tag{11.49}$$

The self-normalized estimator with $a(\theta) = p(y_j \mid \theta)$ then collapses to a **harmonic mean** of the likelihood of y_j over the full-posterior draws (Gelfand, Dey & Chang, 1992):

$$\hat{p}(y_j \mid y_{-j}) = \frac{\sum_i p(y_j \mid \theta_i)\, w_j(\theta_i)}{\sum_i w_j(\theta_i)} = \frac{n}{\sum_{i=1}^{n} 1/p(y_j \mid \theta_i)}, \tag{11.50}$$

one weighted average per left-out observation, all reusing the same n posterior draws.

11.9 Summary

- Importance sampling reweights draws from a proposal $g(\theta)$ by $w(\theta) = f(\theta)/g(\theta)$ instead of accepting or rejecting them, using

$$\widehat{C_f/C_g} = \frac{1}{n} \sum_i w_i, \qquad \hat{A} = \frac{1}{n} \sum_i a(\theta_i)\, \hat{w}_i^{\star}, \quad \hat{w}_i^{\star} = \frac{w_i}{\frac{1}{n} \sum_j w_j}; \tag{11.51}$$

  only the support condition $\{\theta : g(\theta) > 0\} \supseteq \{\theta : f(\theta) > 0\}$ is required, not an envelope $g(\theta) \geq f(\theta)$.
- The effective sample size $n_{\text{eff}} = (\sum_i w_i)^2 / \sum_i w_i^2$ estimates $n_{\text{eff}}^{\star} = n / E_g[w^{\star}(\theta)^2]$. In the volume example (Example 1) it is exactly the number of hits, and the weighted sample is worth an iid sample of that size from $f(\theta)/C_f$.
- A proposal shaped like the target on its support gives near-constant weights and $n_{\text{eff}} \approx n$ (Examples 2 and 4), and a single weighted sample can answer several questions about the same target.
- A proposal that misses a mode or part of the support fails **silently** (Example 3): the observed weights look healthy while the estimate is badly wrong, and no amount of extra sampling fixes it. n_{eff} diagnoses only the *noisy* failure, never the silent one.
- For the marginal likelihood, a proposal built from a Laplace approximation at the posterior mode stays efficient as the data accumulate and the posterior sharpens, while a fixed proposal such as the prior does not (Examples 4 and 5).
- Advanced uses: ratios of normalizing constants (free energy, marginal likelihood) via tempering and stepping stones, and leave-one-out cross-validation from full-posterior draws with $w_j(\theta) = 1/p(y_j \mid \theta)$.

12 Markov Chain Monte Carlo

```
library(truncnorm)
library(rstan)
```

```
Warning: package 'StanHeaders' was built under R version 4.6.1
```

```
rstan_options(auto_write = TRUE)
```

The last two chapters drew from a **fixed** proposal g: an envelope for rejection sampling, an importance distribution for reweighting. Both work well when g can be made to resemble the target f over its whole support, but in more than a handful of dimensions no fixed g does — the target occupies an ever-thinner sliver of the space g covers, exactly the curse of dimensionality met already for quadrature grids. **Markov chain Monte Carlo (MCMC)** answers this differently: instead of one global proposal, build a **random process** $\theta^{(0)}, \theta^{(1)}, \theta^{(2)}, \ldots$ that moves *locally* — each step depends only on where the chain currently is — and is engineered so that, after enough steps, $\theta^{(t)}$ is (approximately) distributed as the target f. The price is that consecutive draws are dependent and the guarantee is only asymptotic; the payoff is that the same local-move idea scales to any dimension.

This chapter develops the Markov chain theory that makes this work, then builds three increasingly automatic ways to construct such a chain — Gibbs sampling, Metropolis–Hastings, and Hamiltonian Monte Carlo — and ends with Stan, which automates the last of these for essentially arbitrary models. Each method is presented first as theory, followed by one or more examples, each named after the problem it solves. A final section applies Stan to a practical problem, a sparse logistic regression with as many predictors as cases, and compares it with `glm` and the LASSO.

12.1 Markov Chains

12.1.1 Definition and the convergence question

A sequence $\theta^{(0)}, \theta^{(1)}, \theta^{(2)}, \ldots$ is a **Markov chain** if the future depends on the past only through the present,

$$P\big(\theta^{(i)} \mid \theta^{(0)}, \ldots, \theta^{(i-1)}\big) = T\big(\theta^{(i)} \mid \theta^{(i-1)}\big), \tag{12.1}$$

where T, the **transition distribution**, does not depend on i. A chain is completely specified by a starting value (or distribution) and T; simulating it just means repeatedly drawing $\theta^{(i)} \sim T(\cdot \mid \theta^{(i-1)})$.

A distribution π is the **equilibrium** (or limiting) distribution of the chain if, for every starting value $\theta^{(0)}$,

$$\lim_{n\to\infty} P\big(\theta^{(n)} = \theta \mid \theta^{(0)}\big) = \pi(\theta)\,: \tag{12.2}$$

after many steps the chain forgets where it started. The usual Markov-chain question is *given T, find π*. MCMC reverses it: given a target π known only up to a constant, can we *design* a T such that, no matter where the chain starts, $\theta^{(i)}$ converges in distribution to π as $i \to \infty$? If so, running the chain for a long time and averaging $a(\theta^{(i)})$ over the later iterations estimates $E_\pi[a(\theta)]$ — this is what "MCMC" means in practice.

12.1.2 Invariance, detailed balance, and convergence

π is an **invariant** (or stationary) distribution of transition T if

$$\int \pi(\theta)\, T(\theta \to \theta')\, d\theta = \pi(\theta') \qquad \text{for all } \theta'. \tag{12.3}$$

In words: if $\theta \sim \pi$ and one step of the chain is applied, the result is again distributed as π — π is a fixed point of T. Invariance alone does not guarantee the chain *converges* to π from an arbitrary start (a chain that never moves leaves every distribution invariant), so two more conditions are needed: **aperiodicity** (the chain does not cycle deterministically through a fixed sequence of states) and **irreducibility** (every state can eventually be reached from every other). Under all three, $\theta^{(i)} \to \pi$ in distribution from any starting point, and ergodic averages converge, $\frac{1}{n}\sum_i a(\theta^{(i)}) \to E_\pi[a(\theta)]$.

Invariance is a *global* condition (it involves integrating over all θ) and is awkward to check directly for a proposed T. The standard shortcut is a *local*, pairwise condition. T satisfies **detailed balance** with respect to π if

$$\pi(\theta)\, T(\theta \to \theta') = \pi(\theta')\, T(\theta' \to \theta) \qquad \text{for all } \theta, \theta', \tag{12.4}$$

i.e. the flow of probability from θ to θ' under π exactly balances the reverse flow (the transition T itself need not be symmetric — only the π-weighted flow is). Detailed balance implies invariance: integrating both sides over θ,

$$\begin{aligned} \int \pi(\theta)\, T(\theta \to \theta')\, d\theta &= \int \pi(\theta')\, T(\theta' \to \theta)\, d\theta \\ &= \pi(\theta') \underbrace{\int T(\theta' \to \theta)\, d\theta}_{=1} = \pi(\theta'). \end{aligned} \tag{12.5}$$

Every construction in this chapter — Gibbs, Metropolis–Hastings, Hamiltonian Monte Carlo — is a recipe for building a T that satisfies detailed balance (or, for Gibbs, invariance directly) with respect to a *given* target π, so that simulating the chain long enough and discarding an initial **burn-in** period produces (dependent) draws from π.

12.1.3 Example 1: Two Random Walks on a Cycle

The machinery above can be run in the direction opposite to MCMC: take a transition T and *find* its equilibrium by solving the detailed-balance equations. Consider two chains on the eleven states $\{0, 1, \dots, 10\}$ arranged on a circle, with indices taken mod 11, so that 10 and 0 are neighbours and 5 sits in the middle. At each step a chain moves to one of its two neighbours or stays put, and the two chains differ only in the probabilities:

- the **cyclic walk** moves forward (clockwise) with probability 0.45, backward with probability 0.45, and stays with probability 0.1;
- the **walk toward 5** moves to the neighbour *closer to 5* with probability 0.4, to any other neighbour with probability 0.2, and stays with the remaining probability.

Writing $d(i) = |i - 5|$ for the distance from i to 5 (which is also the distance around the circle), the second chain has, for a neighbour $j = i \pm 1$,

$$T_5(i \to j) = \begin{cases} 0.4, & d(j) < d(i) \quad \text{(toward 5)}, \\ 0.2, & \text{otherwise}, \end{cases} \qquad T_5(i \to i) = 1 - T_5(i \to i+1) - T_5(i \to i-1), \tag{12.6}$$

so the probability of staying is 0.4 at every state except 5, where both moves lead away from 5 and it is 0.6. The neighbours 10 and 0 are both at distance 5, so a move between them is "other" in both directions and has probability 0.2.

```
par(mfrow = c(1, 2), mar = c(0.5, 0.5, 2, 0.5), cex = 1.1)
ang <- -pi / 2 - 2 * pi * ((0:10) - 5) / 11; rr <- 0.13; dl <- 0.11   # clockwise, 5 at the
 ↪ bottom
draw_cycle <- function(main, col_fwd, col_bwd, lwd_fwd, lwd_bwd, node_bg = rep("white", 11)) {
    plot(NA, xlim = c(-1.45, 1.45), ylim = c(-1.35, 1.35), asp = 1, axes = FALSE, xlab = "",
     ↪ ylab = "", main = main)
    for (i in 0:10) {
        a1 <- ang[i + 1]; a2 <- ang[(i + 1) %% 11 + 1]
        arrows(1.07 * cos(a1 - dl), 1.07 * sin(a1 - dl), 1.07 * cos(a2 + dl), 1.07 * sin(a2 +
         ↪ dl),
               length = 0.07, col = col_fwd[i + 1], lwd = lwd_fwd[i + 1])        # i -> i + 1
                ↪ (clockwise)
        arrows(0.93 * cos(a2 + dl), 0.93 * sin(a2 + dl), 0.93 * cos(a1 - dl), 0.93 * sin(a1 -
         ↪ dl),
               length = 0.07, col = col_bwd[i + 1], lwd = lwd_bwd[i + 1])        # i + 1 -> i
    }
    symbols(cos(ang), sin(ang), circles = rep(rr, 11), inches = FALSE, add = TRUE, bg =
     ↪ node_bg, lwd = 2)
    text(cos(ang), sin(ang), 0:10)
}
draw_cycle("cyclic walk", rep("gray40", 11), rep("gray40", 11), rep(1.8, 11), rep(1.8, 11))
text(0, 0.1, "each move 0.45", col = "gray30"); text(0, -0.1, "stay 0.1", col = "gray30")
## clockwise i -> i + 1 is toward 5 for i = 0, ..., 4; i + 1 -> i is toward 5 for i = 5, ...,
 ↪ 9;
## the pair (10, 0) is at equal distance from 5, so 0.2 both ways
toward_fwd <- (0:10) <= 4; toward_bwd <- (0:10) >= 5 & (0:10) <= 9
draw_cycle("walk toward 5",
           ifelse(toward_fwd, "steelblue", "firebrick"), ifelse(toward_bwd, "steelblue",
            ↪ "firebrick"),
           ifelse(toward_fwd, 3, 1.5), ifelse(toward_bwd, 3, 1.5), node_bg = ifelse(0:10 == 5,
            ↪ "gold", "white"))
text(0, 0.2, "toward 5: 0.4", col = "steelblue"); text(0, 0, "other: 0.2", col = "firebrick")
text(0, -0.2, "stay: the rest", col = "gray30")
```

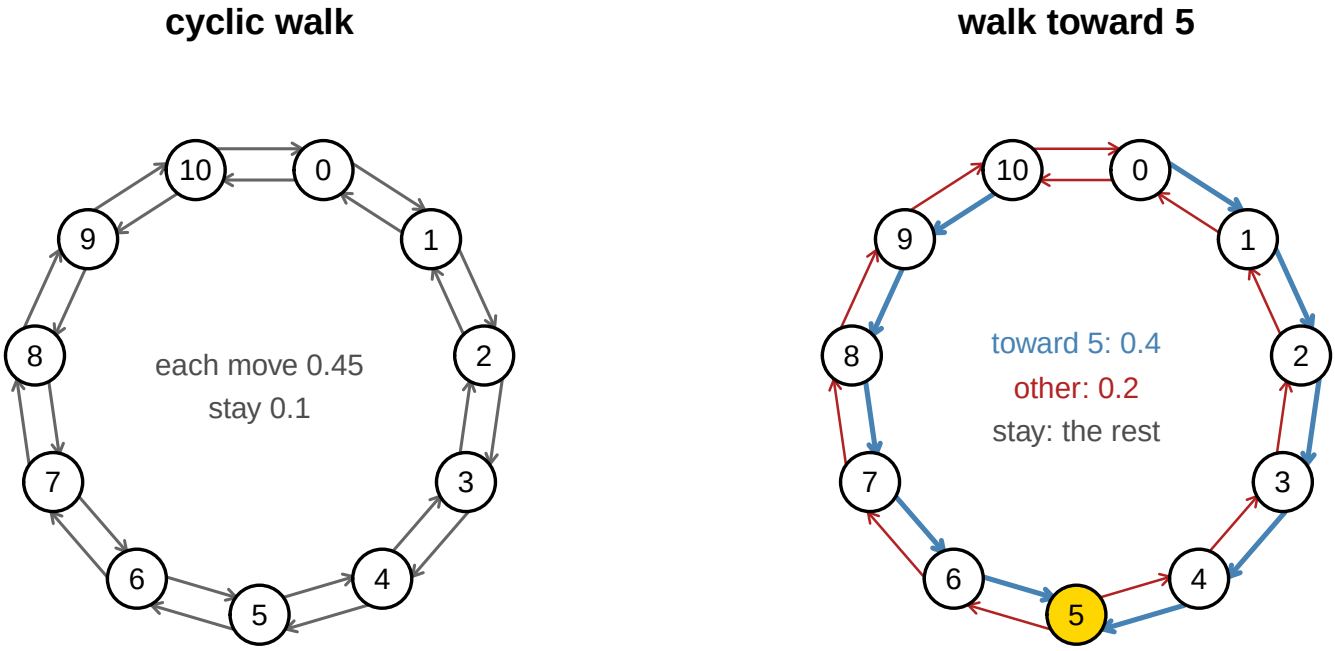


Figure 12.1: Shows the transition structure of two random walks on eleven states arranged on a circle. Left: the cyclic walk, which moves to either neighbour with probability 0.45. Right: the walk toward 5, which moves toward state 5 (thick blue arrows) with probability 0.4 and in any other direction (thin red arrows) with probability 0.2.

Detailed balance only involves neighbours. In both chains $T(i \to j) = 0$ unless j is a neighbour of i, so for non-neighbours both sides of the detailed-balance equation are 0, and for $i = j$ the equation is trivial. Only the eleven neighbouring pairs $(i, i+1)$ need to be checked, and the stay probabilities never enter.

The cyclic walk. For every pair,

$$\pi(i)\,T(i \to i+1) = \pi(i+1)\,T(i+1 \to i) \quad\Longleftrightarrow\quad 0.45\,\pi(i) = 0.45\,\pi(i+1) \quad\Longleftrightarrow\quad \pi(i+1) = \pi(i), \tag{12.7}$$

so going around the circle $\pi(0) = \pi(1) = \cdots = \pi(10)$ and, normalizing, $\pi_{\text{cyc}}(i) = 1/11$. By the theorem above this π is invariant; the chain is irreducible (every state is reached by moving around the circle) and aperiodic (it can stay put), so the uniform distribution is its equilibrium. Every state has the same local picture, so the equilibrium treats them all alike.

The walk toward 5. In every neighbouring pair except $\{10, 0\}$, one state j is one step closer to 5 than the other state i, and detailed balance reads

$$\pi(i)\,\underbrace{T(i \to j)}_{0.4\ \text{(toward)}} = \pi(j)\,\underbrace{T(j \to i)}_{0.2\ \text{(away)}} \quad\Longleftrightarrow\quad \pi(j) = 2\,\pi(i), \tag{12.8}$$

while the pair $\{10, 0\}$, both at distance 5, gives $0.2\,\pi(10) = 0.2\,\pi(0)$. Each step closer to 5 doubles π, so $\pi(i) \propto 2^{-|i-5|}$, and normalizing with $\sum_i 2^{5-|i-5|} = 32 + 2(16+8+4+2+1) = 94$,

$$\pi_5(i) = \frac{2^{\,5-|i-5|}}{94}, \qquad i = 0, 1, \ldots, 10. \tag{12.9}$$

The equations are consistent around the circle (π doubles five times from 0 up to 5, halves five times down to 10, and $\pi(10) = \pi(0)$), and the chain is irreducible and aperiodic, so this is its equilibrium: symmetric, with a peak of $32/94 \approx 0.34$ at 5, falling by half for each step away.

Empirical check. Each chain is simulated by a one-line step function that, given the current state and a uniform draw u, returns the next state:

```
cyc_step <- function(s, u)
    ifelse(u < 0.45, (s + 1) %% 11, ifelse(u < 0.9, (s - 1) %% 11, s))
toward5_step <- function(s, u) {
    fwd <- (s + 1) %% 11; bwd <- (s - 1) %% 11
    p_fwd <- ifelse(abs(fwd - 5) < abs(s - 5), 0.4, 0.2)     # toward 5: 0.4, other: 0.2
    p_bwd <- ifelse(abs(bwd - 5) < abs(s - 5), 0.4, 0.2)
    ifelse(u < p_fwd, fwd, ifelse(u < p_fwd + p_bwd, bwd, s))
}
sim_chain <- function(step, ini, iters) {
    mc <- integer(iters + 1); mc[1] <- ini
    for (i in 2:(iters + 1)) mc[i] <- step(mc[i - 1], runif(1))
    mc
}
set.seed(1)
cyc <- sim_chain(cyc_step, 2, 20000)
tw5 <- sim_chain(toward5_step, 2, 20000)
pi_cyc <- rep(1 / 11, 11); pi_5 <- 2^(5 - abs(0:10 - 5)) / 94       # from detailed balance
freq <- function(mc) as.vector(table(factor(mc[-(1:200)], levels = 0:10))) /
 ↪  length(mc[-(1:200)])
```

```
par(mfrow = c(2, 2), mar = c(4, 4, 2, 1))
plot(cyc[1:300], type = "s", col = "steelblue", xlab = "step", ylab = "state", ylim = c(0, 10),
     main = "cyclic walk: first 300 steps", yaxt = "n"); axis(2, at = 0:10, las = 1)
plot(tw5[1:300], type = "s", col = "firebrick", xlab = "step", ylab = "state", ylim = c(0, 10),
     main = "walk toward 5: first 300 steps", yaxt = "n"); axis(2, at = 0:10, las = 1)
bp <- barplot(freq(cyc), names.arg = 0:10, col = "lightsteelblue", ylim = c(0, 0.4),
              xlab = "state", ylab = "frequency", main = "cyclic walk: uniform, 1/11")
points(bp, pi_cyc, pch = 19, col = "steelblue4", cex = 1.3)
bp <- barplot(freq(tw5), names.arg = 0:10, col = "mistyrose", ylim = c(0, 0.5),
              xlab = "state", ylab = "frequency", main = "walk toward 5")
points(bp, pi_5, pch = 19, col = "firebrick4", cex = 1.3)
legend("topleft", bty = "n", pch = c(15, 19), col = c("mistyrose", "firebrick4"), pt.cex = c(2,
 ↪  1.3),
       legend = c("observed frequency", "equilibrium from detailed balance"))
```

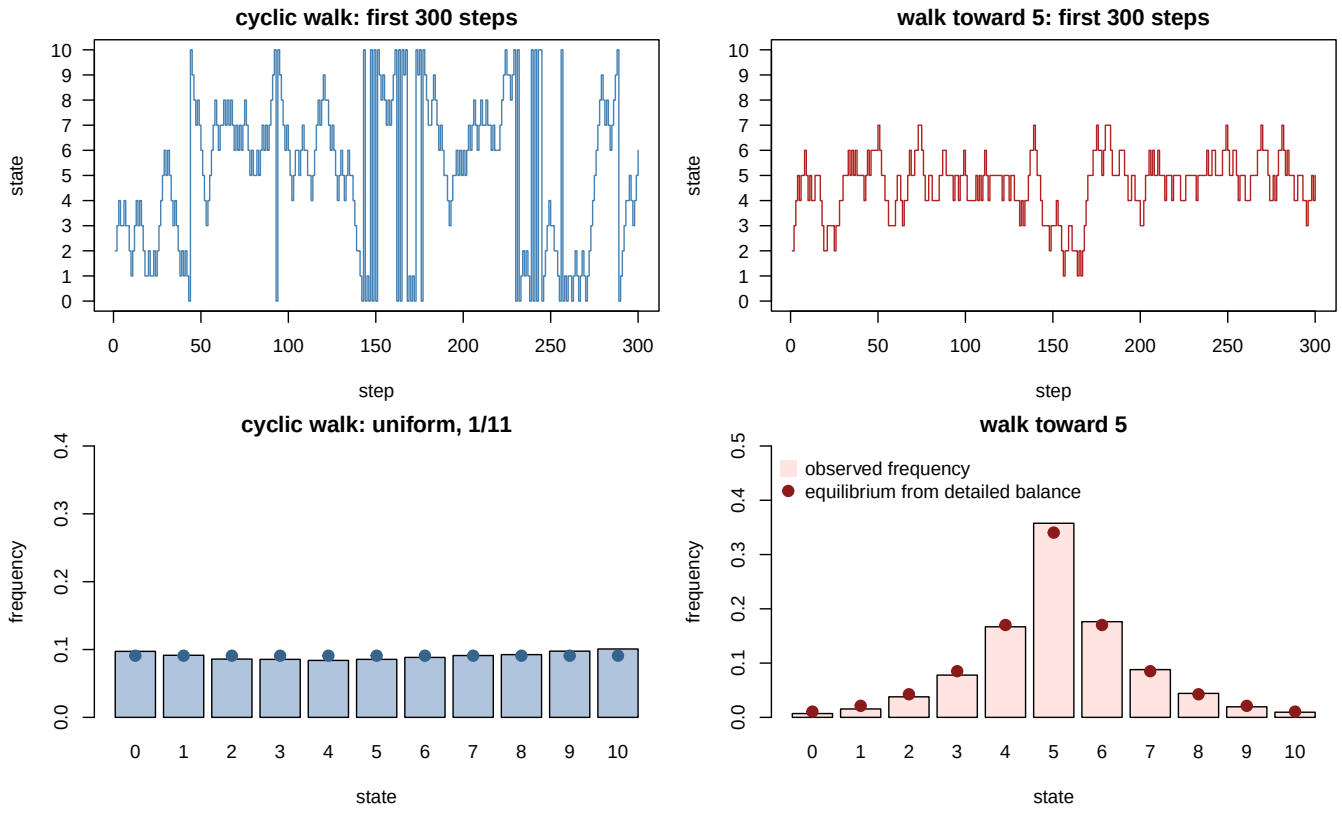


Figure 12.2: Confirms the equilibrium distributions found from detailed balance by simulating the two random walks. One chain of each kind, started at state 2 and run for 20,000 steps. Top: the first 300 steps. Bottom: long-run frequencies of the states after discarding 200 steps (bars), against the equilibrium from detailed balance (points): uniform for the cyclic walk, and 2^(5 - |i - 5|)/94 for the walk toward 5.

```
tab <- rbind("cyclic walk, frequency" = freq(cyc), "cyclic walk, equilibrium" = pi_cyc,
             "walk toward 5, frequency" = freq(tw5), "walk toward 5, equilibrium" = pi_5)
colnames(tab) <- 0:10
knitr::kable(tab, digits = 3)
```

Table 12.1: Compares the long-run state frequencies of the two simulated random walks with their equilibrium distributions from detailed balance. Frequencies over 20,000 steps after discarding the first 200.

	0	1	2	3	4	5	6	7	8	9	10
cyclic walk, frequency	0.097	0.092	0.086	0.086	0.084	0.086	0.088	0.091	0.092	0.098	0.101
cyclic walk, equilibrium	0.091	0.091	0.091	0.091	0.091	0.091	0.091	0.091	0.091	0.091	0.091
walk toward 5, frequency	0.007	0.015	0.038	0.078	0.167	0.358	0.176	0.088	0.044	0.019	0.009
walk toward 5, equilibrium	0.011	0.021	0.043	0.085	0.170	0.340	0.170	0.085	0.043	0.021	0.011

The observed frequencies agree with the equilibria found from detailed balance up to Monte Carlo error. The two chains differ only in their transition probabilities, yet one spreads its time evenly over the circle and the other concentrates around state 5: the equilibrium is determined entirely by T, and the pairwise detailed-balance equations are the easiest way to read it off. MCMC turns this around: it starts from the target π and chooses a T whose detailed-balance equations π satisfies.

12.2 Gibbs Sampling

12.2.1 The idea and why it works

Suppose $\theta = (\theta_1, \theta_2)$ and both full conditionals $\pi(\theta_1 \mid \theta_2)$ and $\pi(\theta_2 \mid \theta_1)$ can be sampled directly, even though the joint $\pi(\theta_1, \theta_2)$ has no closed form or cannot be sampled directly. The **Gibbs sampler** alternates

$$\theta_1^{(t)} \sim \pi(\theta_1 \mid \theta_2^{(t-1)}), \qquad \theta_2^{(t)} \sim \pi(\theta_2 \mid \theta_1^{(t)}). \tag{12.10}$$

Each half-step leaves π invariant: if $(\theta_1, \theta_2) \sim \pi$, then $\theta_2 \sim \pi(\theta_2)$ (its marginal), and replacing θ_1 by a fresh draw from $\pi(\theta_1 \mid \theta_2)$ produces a pair with joint density $\pi(\theta_2)\,\pi(\theta_1 \mid \theta_2) = \pi(\theta_1, \theta_2)$ — exactly π again. Cycling through both coordinates therefore also leaves π invariant, and as long as the full conditionals are everywhere positive the chain is irreducible and aperiodic. No proposal, no acceptance step, no tuning: Gibbs sampling only needs full conditionals that can be sampled.

12.2.2 Data augmentation

Gibbs sampling extends immediately to models with **latent variables**. If the target $\pi(\theta)$ is awkward but can be written as the marginal of a joint density $\pi(\theta, z)$ over θ and some extra unknowns z,

$$\pi(\theta) = \int \pi(\theta, z)\, dz, \tag{12.11}$$

then a Gibbs sampler on (θ, z) that alternates $z \sim \pi(z \mid \theta)$ and $\theta \sim \pi(\theta \mid z)$ has $\pi(\theta, z)$ as its invariant distribution, so its θ-coordinates are draws from $\pi(\theta)$: the z's are simply discarded. This **data augmentation** pays off when the conditionals given z are standard even though $\pi(\theta)$ itself is not. The z may be real missing data (Example 3, where exact values are hidden by censoring) or a purely mathematical device (Example 4, where a latent variance per observation turns a Cauchy likelihood into a normal one).

12.2.3 Example 2: Normal Data with Unknown Mean and Variance

12.2.3.1 The full conditionals

The running example for this half of the book is $x_1, \dots, x_n \mid \mu, \sigma^2 \overset{iid}{\sim} N(\mu, \sigma^2)$, with independent priors $\mu \sim N(\mu_0, \sigma_0^2)$ and $\sigma^2 \sim \text{IG}(a_0, \lambda_0)$ (inverse-gamma: $1/\sigma^2 \sim \text{Gamma}(a_0, \lambda_0)$, so a_0 is a shape and λ_0 a rate/scale). The joint posterior is

$$\begin{aligned} P(\mu, \sigma^2 \mid x) \;\propto\; & e^{-(\mu-\mu_0)^2/(2\sigma_0^2)} \cdot (\sigma^2)^{-(a_0+1)} e^{-\lambda_0/\sigma^2} \\ & \times (\sigma^2)^{-n/2} \exp\Big\{ -\sum_{i=1}^{n} \tfrac{(x_i-\mu)^2}{2\sigma^2} \Big\}, \end{aligned} \tag{12.12}$$

two-dimensional with no closed-form normalizing constant, but each conditional is standard.

The model is easiest to read as a graph (Figure 12.3). Squares are fixed hyperparameters, open circles are unknown random quantities, shaded circles are observed data, and each arrow points from a quantity to one whose distribution depends on it. The box (a *plate*) stands for n copies of what it contains, one for each observation. The full conditional of a node depends only on its parents, its children and its children's other parents, which is why the conditional of μ needs only its prior constants, the data, and σ^2 (the other parent of the data), and likewise for σ^2.

```
### draw a directed acyclic graph of a Bayesian model:
### squares = fixed hyperparameters, open circles = unknown random quantities,
### shaded circles = observed data (type "observed_square" draws observed data as a
### shaded square; type "dots" is a bare label, e.g. for an omitted stretch of a chain); arrows
 ↪  point from parent to child, and a
### rounded box (plate) holds the quantities repeated over i = 1, ..., n
draw_dag <- function(nodes, edges, plates = list(), main = "", r = 0.33,
                     xlim = range(nodes$x) + c(-0.8, 0.8), ylim = range(nodes$y) + c(-0.8,
                      ↪  0.8)) {
    plot.new(); plot.window(xlim = xlim, ylim = ylim, asp = 1)
    title(main, cex.main = 1)
    for (p in plates) {
        rect(p$x[1], p$y[1], p$x[2], p$y[2], border = "gray45", lwd = 1.5)
        text(p$x[2] - 0.08, p$y[1] + 0.08, p$label, adj = c(1, 0), cex = 0.85, col = "gray30")
    }
    for (k in seq_len(nrow(edges))) {
        a <- nodes[nodes$id == edges$from[k], ]; b <- nodes[nodes$id == edges$to[k], ]
        d <- c(b$x - a$x, b$y - a$y); u <- d / sqrt(sum(d^2))
        arrows(a$x + r * u[1], a$y + r * u[2], b$x - (r + 0.03) * u[1], b$y - (r + 0.03) *
         ↪  u[2],
               length = 0.08, lwd = 1.4, col = if (is.null(edges$col)) "gray20" else
                ↪  edges$col[k])
    }
    th <- seq(0, 2 * pi, length.out = 100)
    for (k in seq_len(nrow(nodes))) {
        nd <- nodes[k, ]
        if (nd$type == "dots") {
            # placeholder for the omitted middle of a chain: label only
        } else if (nd$type %in% c("fixed", "observed_square")) {
            rect(nd$x - r * 0.85, nd$y - r * 0.85, nd$x + r * 0.85, nd$y + r * 0.85,
                 col = if (nd$type == "observed_square") "gray80" else "white",
                 border = "gray20", lwd = 1.5)
        } else {
            polygon(nd$x + r * cos(th), nd$y + r * sin(th), border = "gray20", lwd = 1.5,
                    col = if (nd$type == "observed") "gray80" else "white")
        }
        text(nd$x, nd$y, parse(text = nd$label), cex = 1.05)
    }
}
```

```
par(mar = c(0.5, 0.5, 0.5, 0.5))
nodes_norm <- data.frame(
    id    = c("m0", "s0", "a0", "l0", "mu", "s2", "x"),
```

```
    label = c("mu[0]", "sigma[0]^2", "a[0]", "lambda[0]", "mu", "sigma^2", "x[i]"),
    x = c(-1.6, -0.6, 0.6, 1.6, -1.1, 1.1, 0), y = c(2, 2, 2, 2, 1, 1, -0.2),
    type = c(rep("fixed", 4), "latent", "latent", "observed"))
edges_norm <- data.frame(from = c("m0", "s0", "a0", "l0", "mu", "s2"),
                         to   = c("mu", "mu", "s2", "s2", "x", "x"))
draw_dag(nodes_norm, edges_norm,
         plates = list(list(x = c(-0.8, 0.8), y = c(-0.9, 0.5), label = "i = 1, ..., n")))
```

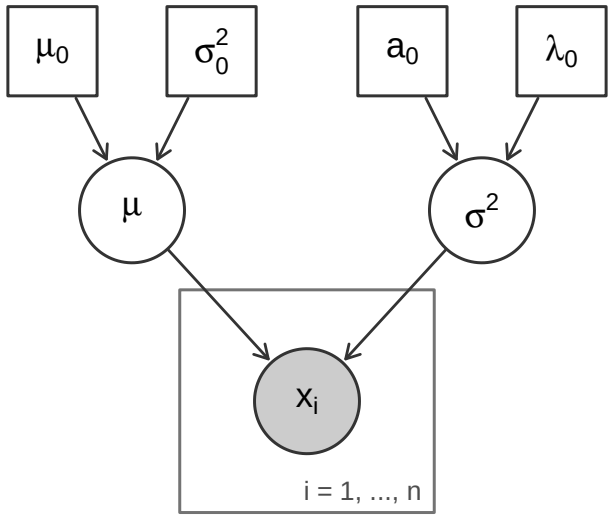


Figure 12.3: Shows the structure of the normal model with unknown mean and variance as a directed graph. Squares: fixed hyperparameters; open circles: unknown parameters; shaded circle: observed data. The plate repeats x_i for i = 1, …, n.

Conditional of μ. Collect the factors that involve μ and expand $\sum_i (x_i-\mu)^2 = n\mu^2 - 2n\mu\bar{x} + \sum_i x_i^2$:

$$P(\mu \mid \sigma^2, x) \;\propto\; \exp\Big\{ -\tfrac{1}{2}\Big[\big(\tfrac{1}{\sigma_0^2} + \tfrac{n}{\sigma^2}\big)\mu^2 - 2\big(\tfrac{\mu_0}{\sigma_0^2} + \tfrac{n\bar{x}}{\sigma^2}\big)\mu \Big]\Big\}, \tag{12.13}$$

which is the kernel of $N(m, v)$ with $1/v = 1/\sigma_0^2 + n/\sigma^2$ and $m/v = \mu_0/\sigma_0^2 + n\bar{x}/\sigma^2$ — a precision-weighted average of the prior mean and the sample mean, with n/σ^2 the weight the data contribute. As $n \to \infty$, $m \to \bar{x}$ and $v \approx \sigma^2/n$: the prior is overwhelmed.

Conditional of σ^2. Collect the factors involving σ^2:

$$\begin{aligned} P(\sigma^2 \mid \mu, x) \;&\propto\; (\sigma^2)^{-(a_0+1)} e^{-\lambda_0/\sigma^2} \cdot (\sigma^2)^{-n/2} e^{-\sum_i (x_i-\mu)^2/(2\sigma^2)} \\ &=\; (\sigma^2)^{-(\frac{n}{2}+a_0+1)} \exp\Big\{ -\frac{\lambda_0 + \frac{1}{2}\sum_i (x_i-\mu)^2}{\sigma^2} \Big\}, \end{aligned} \tag{12.14}$$

again an inverse-gamma kernel, $\sigma^2 \mid \mu, x \sim \mathrm{IG}\big(\tfrac{n}{2} + a_0,\; \tfrac{1}{2}\sum_i (x_i-\mu)^2 + \lambda_0\big)$: the prior contributes a_0 "pseudo-observations" worth of shape and λ_0 worth of sum-of-squares, and the data add n and $\sum_i (x_i-\mu)^2$ to each.

12.2.3.2 Implementation

We use the weak prior $\mu_0 = 0$, $\sigma_0^2 = 100$, $a_0 = \lambda_0 = 1$, stored once as `mu0`, `s02`, `a0` and `lam0` and reused by every sampler of this model in the chapter. Each line of the loop is one of the two full conditionals above; an inverse-gamma draw is the reciprocal of a gamma draw with the same shape and rate.

```
mu0 <- 0; s02 <- 100; a0 <- 1; lam0 <- 1                        # prior
gibbs_norm <- function(iters, x, mu_init = 0, s2_init = 1) {
    n <- length(x)
    out <- matrix(NA_real_, iters, 2, dimnames = list(NULL, c("mu", "s2")))
    mu <- mu_init; s2 <- s2_init
    for (t in seq_len(iters)) {
        v  <- 1 / (1 / s02 + n / s2)                             # mu | sigma^2, x ~ N(m, v)
        mu <- rnorm(1, v * (mu0 / s02 + sum(x) / s2), sqrt(v))
        s2 <- 1 / rgamma(1, shape = n / 2 + a0, rate = sum((x - mu)^2) / 2 + lam0)   # IG
        out[t, ] <- c(mu, s2)
    }
    out
}
```

A preallocated matrix filled by a plain `for` loop replaces the original `replicate()`-with-`<<-` pattern: it does the same work with one direct assignment per iteration instead of a closure mutating variables in its enclosing scope, and it makes the recurrence — each row built from the previous one — visually explicit.

```
set.seed(2)
x <- rnorm(50, 10, 2)
draws_gibbs <- gibbs_norm(10000, x)
burn <- 1:200
mu_g  <- draws_gibbs[-burn, "mu"]
sd_g  <- sqrt(draws_gibbs[-burn, "s2"])

par(mfrow = c(2, 2), mar = c(4, 4, 2, 1))
plot(mu_g, type = "l", col = "steelblue", xlab = "iteration", ylab = expression(mu), main =
 ↪  "trace of mu")
plot(density(mu_g), col = "steelblue", lwd = 2, main = "posterior density of mu", xlab =
 ↪  expression(mu))
plot(sd_g, type = "l", col = "firebrick", xlab = "iteration", ylab = expression(sigma), main =
 ↪  "trace of sigma")
plot(density(sd_g), col = "firebrick", lwd = 2, main = "posterior density of sigma", xlab =
 ↪  expression(sigma))
```

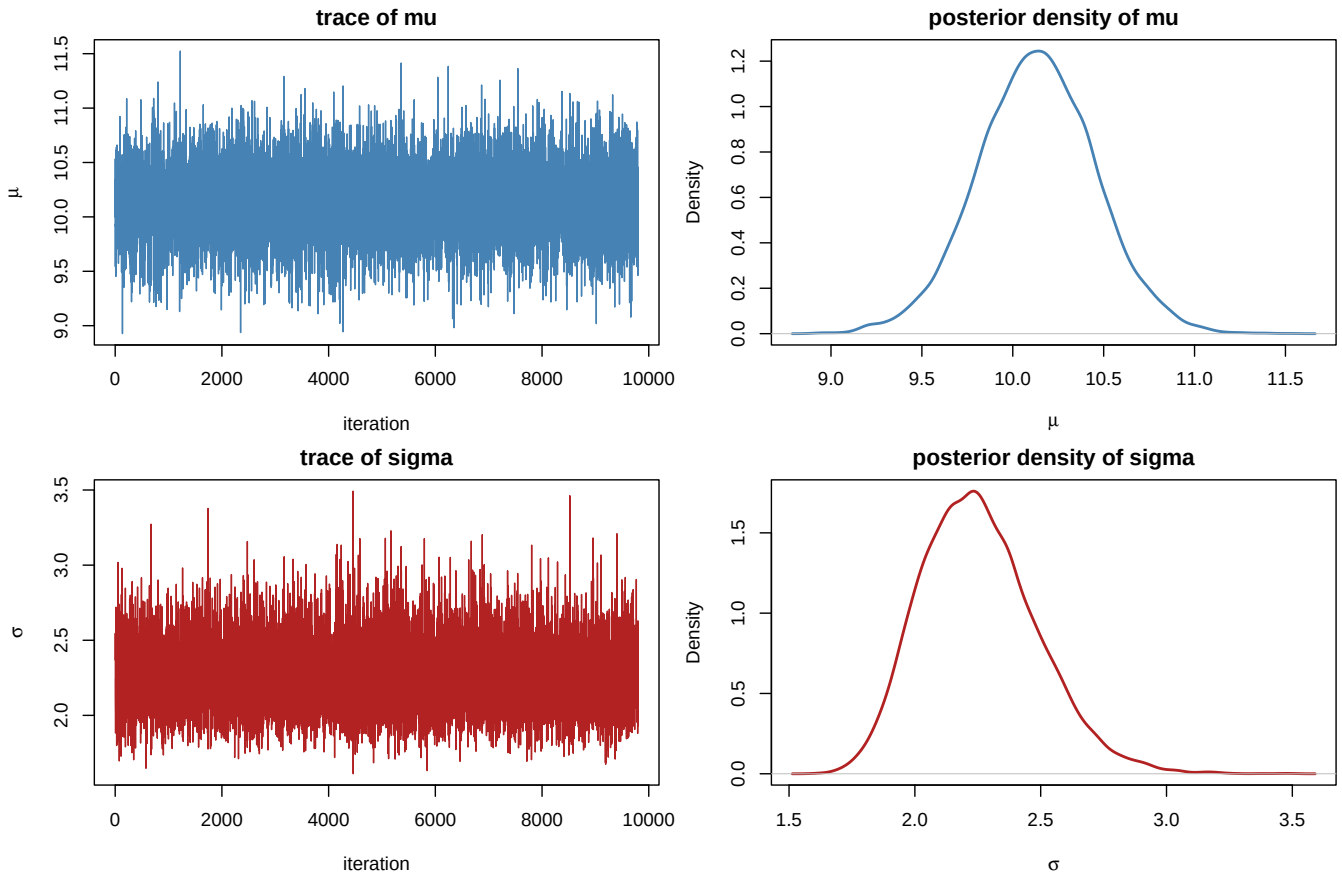


Figure 12.4: Demonstrates the Gibbs sampler on a normal model with unknown mean and standard deviation, using trace plots to check convergence and densities to summarize the posterior. Gibbs sampler on 50 simulated N(10, 2^2) observations: trace plots and posterior densities for mu and sigma, with the weak prior.

```
gibbs_summary <- data.frame(
    parameter = c("mu", "sigma"),
    mean = c(mean(mu_g), mean(sd_g)),
    sd   = c(sd(mu_g), sd(sd_g)),
    q025 = c(quantile(mu_g, 0.025), quantile(sd_g, 0.025)),
    q975 = c(quantile(mu_g, 0.975), quantile(sd_g, 0.975))
)
knitr::kable(gibbs_summary, digits = 3,
             col.names = c("parameter", "mean", "sd", "2.5%", "97.5%"),
             align = c("l", "r", "r", "r", "r"))
```

Table 12.2: Summarizes the posterior distribution obtained from the Gibbs sampler. Posterior summaries from the Gibbs sampler (after discarding a 200-iteration burn-in).

parameter	mean	sd	2.5%	97.5%
mu	10.141	0.330	9.489	10.791
sigma	2.293	0.239	1.885	2.809

Because each update changes one coordinate while holding the other fixed, the path of a Gibbs sampler on the joint posterior is a staircase of horizontal and vertical moves (Figure 12.5). From a poor start the chain reaches the bulk of the posterior within a few iterations.

```
set.seed(21)
d2 <- gibbs_norm(30, x, mu_init = 8, s2_init = 8)
lpost <- function(mu, s2) sum(dnorm(x, mu, sqrt(s2), log = TRUE)) + dnorm(mu, mu0, sqrt(s02),
 ↪  log = TRUE) -
    (a0 + 1) * log(s2) - lam0 / s2
mg <- seq(7.5, 11.5, length.out = 100); sg <- seq(1.5, 9, length.out = 100)
Z <- outer(mg, sg, Vectorize(lpost)); Z <- exp(Z - max(Z))
layout(matrix(c(1, 1, 2, 3), 2)); par(mar = c(4, 4, 2, 1))
contour(mg, sg, Z, nlevels = 8, drawlabels = FALSE, xlab = expression(mu), ylab =
 ↪  expression(sigma^2),
        main = "Gibbs path: first 30 moves from (8, 8)")
pp <- rbind(c(8, 8)); for (t in 1:30) pp <- rbind(pp, c(d2[t, 1], pp[nrow(pp), 2]), d2[t, ])
 ↪  # horizontal then vertical
lines(pp, col = "firebrick", lwd = 1.5); points(d2, pch = 19, col = "firebrick", cex = 0.7)
points(8, 8, pch = 17, cex = 1.5)
plot(draws_gibbs[1:100, "mu"], type = "l", xlab = "t", ylab = expression(mu), main = "trace of
 ↪  mu")
plot(draws_gibbs[1:100, "s2"], type = "l", xlab = "t", ylab = expression(sigma^2), main =
 ↪  "trace of sigma^2")
```

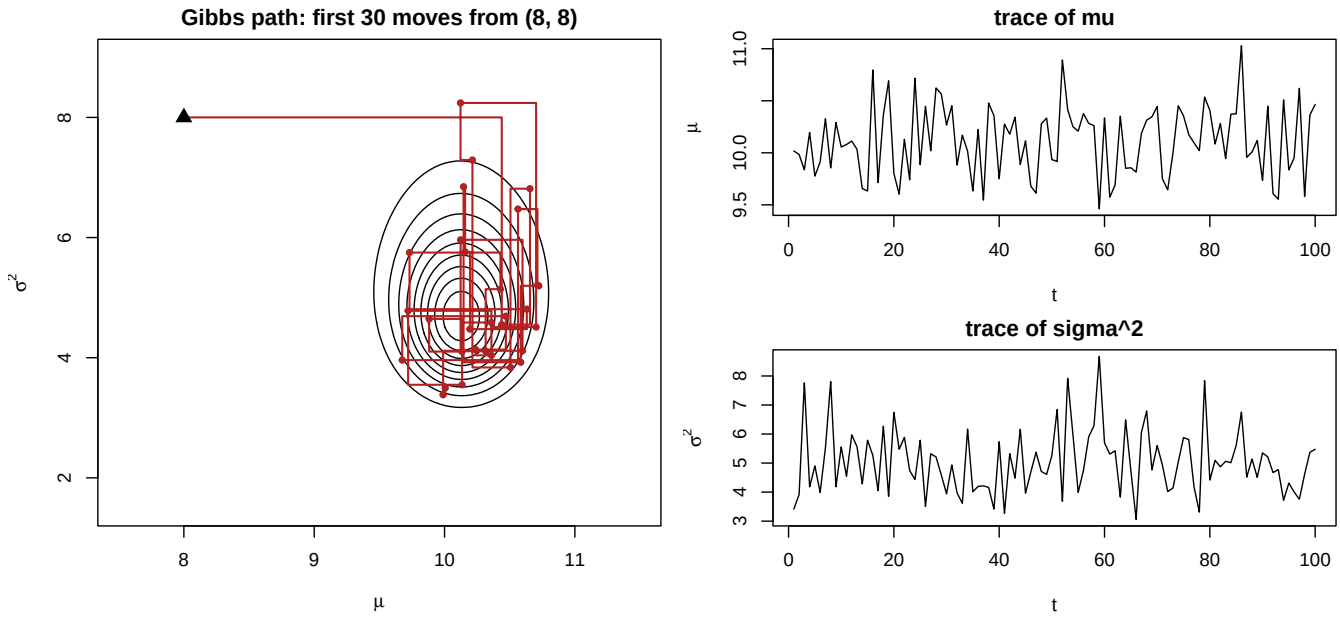


Figure 12.5: Shows how the Gibbs sampler moves on the joint posterior of the normal model, one coordinate at a time. Left: the first 30 moves from the poor start (8, 8) (triangle) over the contours of the exact posterior; each move is horizontal (a mu update) or vertical (a sigma^2 update). Right: trace plots of the first 100 iterations of the main run.

12.2.4 Example 3: Interval-Censored Normal Data

The problem. The quantities of interest are $x_1, \dots, x_n \mid \mu, \sigma^2 \overset{iid}{\sim} N(\mu, \sigma^2)$, with the priors of Example 2, but not every x_i is seen exactly. For each unit we observe a pair (I_i, δ_i):

- $\delta_i = 1$ means x_i is **not censored**: it is observed exactly, and $I_i = \{x_i\}$.
- $\delta_i = 0$ means x_i is **censored**: we only learn an interval $I_i = [a_i, b_i)$ that contains it. The interval may be bounded (a value recorded only to the nearest decade, or between two inspection times) or open-ended, $b_i = \infty$ (a value known only to exceed a detection or follow-up limit, which is right censoring).

The observed data are the (I_i, δ_i), and the censored x_i are unknown. The indicators δ_i are independent of the x_i and of (μ, σ^2), so they carry no information about the parameters, and the likelihood is

$$L(\mu, \sigma^2) = \prod_{i:\,\delta_i = 1} \frac{1}{\sigma}\,\varphi\Big(\frac{x_i - \mu}{\sigma}\Big) \times \prod_{i:\,\delta_i = 0} \Big[\Phi\Big(\frac{b_i - \mu}{\sigma}\Big) - \Phi\Big(\frac{a_i - \mu}{\sigma}\Big)\Big], \tag{12.15}$$

where φ and Φ are the standard normal density and CDF. The products of Φ differences make the posterior non-standard: neither conditional of μ or σ^2 is a normal or inverse-gamma distribution.

Data augmentation. Treat the unobserved x_i as extra unknowns. Given all x_i, the data are complete and the conditionals of μ and σ^2 are exactly those of Example 2. Given (μ, σ^2) and the observed (I_i, δ_i), a censored x_i has

density proportional to the $N(\mu, \sigma^2)$ density times the indicator that $x_i \in I_i$:

$$x_i \mid \mu, \sigma^2, I_i, \delta_i = 0 \;\sim\; N(\mu, \sigma^2) \text{ truncated to } [a_i, b_i), \tag{12.16}$$

while an uncensored x_i is fixed at its observed value. One Gibbs iteration is

$$\begin{aligned} x_i^{(t)} &\sim P\big(x_i \mid \mu^{(t-1)}, \sigma^{2(t-1)}, I_i, \delta_i\big) \quad (\text{censored } i), \\ \mu^{(t)} &\sim P\big(\mu \mid x^{(t)}, \sigma^{2(t-1)}\big), \qquad \sigma^{2(t)} \sim P\big(\sigma^2 \mid x^{(t)}, \mu^{(t)}\big). \end{aligned} \tag{12.17}$$

Integrating the latent x_i out of the augmented model gives back exactly the likelihood above, since $\int_{a_i}^{b_i} \sigma^{-1}\varphi((x-\mu)/\sigma)\,dx$ is the bracketed Φ difference.

In the graph of the model (Figure 12.6), x_i is an unknown circle below the parameters, and the observed interval I_i and indicator δ_i are shaded squares. I_i contains x_i (and equals it when $\delta_i = 1$), so x_i is its parent. The indicator δ_i is independent of x_i and of the parameters (whether a unit is observed exactly does not depend on its value), so it has no arrows: it carries no information about (μ, σ^2).

```
par(mar = c(0.5, 0.5, 0.5, 0.5))
nodes_cens <- rbind(nodes_norm[1:6, ], data.frame(
    id = c("x", "d", "I"), label = c("x[i]", "delta[i]", "I[i]"),
    x = c(0, 1.3, 0), y = c(-0.2, -1.4, -1.4),
    type = c("latent", "observed_square", "observed_square")))
edges_cens <- rbind(edges_norm, data.frame(from = "x", to = "I"))
draw_dag(nodes_cens, edges_cens,
         plates = list(list(x = c(-0.8, 2.0), y = c(-2.1, 0.5), label = "i = 1, ..., n")))
```

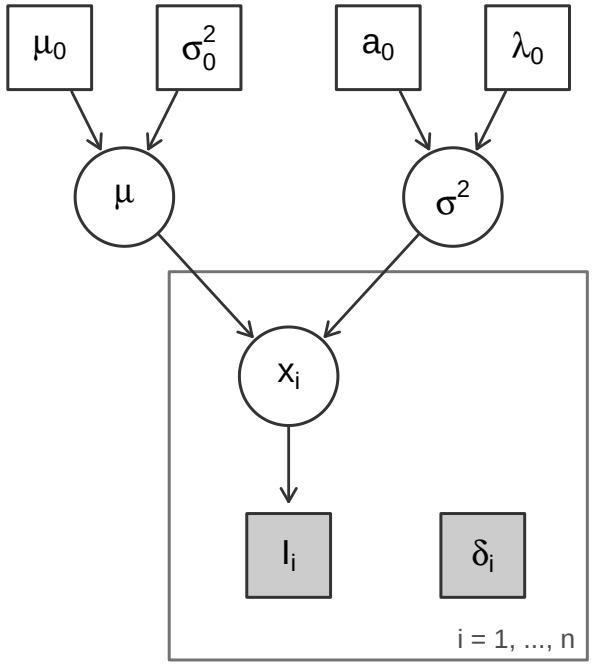


Figure 12.6: Shows the structure of the interval-censored normal model as a directed graph, with the exact values as latent variables. The exact value x_i (open circle) is unknown. The observed data (shaded squares) are the interval I_i that contains x_i and the indicator delta_i, which is 1 when x_i is observed exactly (I_i is the single point x_i) and 0 when it is censored. delta_i is independent of x_i and the parameters, so it has no arrows.

The sampler below stores each interval as its endpoints `a`, `b`, with `a = b = x` for an uncensored unit. The censored x_i are drawn from their truncated normals with `truncnorm::rtruncnorm`, which also handles $b_i = \infty$:

```
gibbs_norm_censored <- function(iters, a, b, delta) {      # prior constants mu0, s02, a0, lam0
 ↪  of Example 2
    n <- length(a); cens <- delta == 0
    x <- ifelse(cens, pmin(a + 1, b), a)     # start: exact values, or a point in the interval
    mu <- mean(x); s2 <- var(x)
    out <- matrix(NA_real_, iters, 2)
    for (t in seq_len(iters)) {
        ### censored units: x_i | mu, sigma^2 ~ N(mu, sigma^2) truncated to [a_i, b_i)
        x[cens] <- rtruncnorm(sum(cens), a = a[cens], b = b[cens], mean = mu, sd = sqrt(s2))
        ### complete-data conditionals of Example 2
        v  <- 1 / (1 / s02 + n / s2)
        mu <- rnorm(1, v * (mu0 / s02 + sum(x) / s2), sqrt(v))
        s2 <- 1 / rgamma(1, shape = n / 2 + a0, rate = sum((x - mu)^2) / 2 + lam0)
        out[t, ] <- c(mu, s2)
    }
    list(mu = out[, 1], sd = sqrt(out[, 2]))
}
```

To test it, simulate $n = 200$ values from $N(100, 40^2)$. Each is observed exactly with probability 0.3. Otherwise it

is censored: values above 150 are only known to be at least 150 (right censoring, $I_i = [150, \infty)$), and the rest are only known to the decade, $I_i = [10k, 10k + 10)$.

```
set.seed(3)
n_c <- 200
x_true <- rnorm(n_c, 100, 40)
delta  <- rbinom(n_c, 1, 0.3)
a_obs  <- ifelse(delta == 1, x_true, ifelse(x_true >= 150, 150, floor(x_true / 10) * 10))
b_obs  <- ifelse(delta == 1, x_true, ifelse(x_true >= 150, Inf, a_obs + 10))
fit_cens <- gibbs_norm_censored(5000, a_obs, b_obs, delta)
burn2 <- 1:200
knitr::kable(data.frame(
    quantity = c("exact / interval / right-censored", "mu", "sigma"),
    truth = c(sprintf("%d / %d / %d", sum(delta == 1), sum(delta == 0 & is.finite(b_obs)),
                      sum(is.infinite(b_obs))), "100", "40"),
    naive = c("", sprintf("%.2f", mean(a_obs)), sprintf("%.2f", sd(a_obs))),
    posterior = c("", sprintf("%.2f (%.2f, %.2f)", mean(fit_cens$mu[-burn2]),
                              quantile(fit_cens$mu[-burn2], 0.025),
                               ↪  quantile(fit_cens$mu[-burn2], 0.975)),
                  sprintf("%.2f (%.2f, %.2f)", mean(fit_cens$sd[-burn2]),
                          quantile(fit_cens$sd[-burn2], 0.025), quantile(fit_cens$sd[-burn2],
                           ↪  0.975)))
), col.names = c("", "truth", "naive", "posterior mean (95% interval)"), align = c("l", "r",
 ↪  "r", "r"))
```

Table 12.3: Shows that Gibbs sampling extends to interval-censored data by treating the exact values as latent variables, and that the parameters are still recovered. n = 200 values from N(100, 40^2): 30% observed exactly, the rest censored to a decade or, above 150, right-censored at 150. The naive estimates treat each censored value as its interval's lower end.

	truth	naive	posterior mean (95% interval)
exact / interval / right-censored	70 / 121 / 9		
mu	100	96.88	93.38 (87.89, 98.69)
sigma	40	38.00	39.65 (35.75, 43.93)

Censoring removes a great deal of information: most units only say which decade x_i fell in, and the largest values are only known to exceed 150. Treating each censored value as the lower end of its interval biases both estimates downward, and the naive estimates come with no honest measure of their uncertainty. The augmented Gibbs sampler still recovers μ and σ, because it imputes each censored x_i from its conditional distribution given the current parameters and so propagates the uncertainty about where in I_i it lies.

12.2.5 Example 4: Cauchy Data with Unknown Location and Scale

The problem. Observations $y_1, \dots, y_n$ are modelled as Cauchy(μ, σ), a heavy-tailed model that is robust to outliers:

$$p(y_i \mid \mu, \sigma) = \frac{1}{\pi\sigma\,(1 + r_i^2)}, \qquad r_i = \frac{y_i - \mu}{\sigma}. \tag{12.18}$$

The priors are $\mu \sim N(\mu_0, \tau^2)$ and $\sigma^2 \sim \text{IG}(a_0, b_0)$, with density $p(\sigma^2) \propto (\sigma^2)^{-(a_0+1)} e^{-b_0/\sigma^2}$. The posterior of (μ, σ^2) is proportional to $e^{-(\mu-\mu_0)^2/(2\tau^2)}\,(\sigma^2)^{-(a_0+1)} e^{-b_0/\sigma^2} \prod_i \sigma^{-1}(1 + r_i^2)^{-1}$. Neither conditional is a standard distribution, because of the factors $(1 + r_i^2)^{-1}$. Data augmentation fixes this.

12.2.5.1 Step 1: the Cauchy as a scale mixture of normals

Introduce a latent variance v_i for each observation, with

$$\begin{aligned} &y_i \mid \mu, \sigma^2, v_i \sim N(\mu, \sigma^2 v_i), \qquad v_i \sim \text{IG}(\tfrac{1}{2}, \tfrac{1}{2}), \\ &p(v_i) = \frac{(1/2)^{1/2}}{\Gamma(1/2)}\, v_i^{-3/2} e^{-1/(2v_i)} = \frac{1}{\sqrt{2\pi}}\, v_i^{-3/2} e^{-1/(2v_i)}. \end{aligned} \tag{12.19}$$

Integrating out v_i gives back the Cauchy density:

$$\begin{aligned} &\int_0^\infty p(y_i \mid \mu, \sigma^2, v_i)\, p(v_i)\, dv_i \\ &\quad = \int_0^\infty \frac{1}{\sqrt{2\pi\sigma^2 v_i}}\, e^{-r_i^2/(2v_i)} \cdot \frac{1}{\sqrt{2\pi}}\, v_i^{-3/2} e^{-1/(2v_i)}\, dv_i \\ &\quad = \frac{1}{2\pi\sigma} \int_0^\infty v_i^{-2} \exp\Big\{ -\frac{1 + r_i^2}{2v_i} \Big\}\, dv_i \\ &\quad = \frac{1}{2\pi\sigma} \cdot \frac{2}{1 + r_i^2} = \frac{1}{\pi\sigma(1 + r_i^2)}, \end{aligned} \tag{12.20}$$

using $\int_0^\infty v^{-2} e^{-c/v}\, dv = 1/c$ (substitute $u = 1/v$). So the augmented model has exactly the original posterior as its (μ, σ^2) marginal, and a Gibbs sampler on (v, μ, σ^2) samples the Cauchy posterior once the v_i are ignored.

Figure 12.7 shows this augmented model. Compared with Figure 12.3, each observation has its own latent variance v_i, a child of the fixed constants $\frac{1}{2}, \frac{1}{2}$ of its IG prior and a parent of y_i. Since v_i has only one child, its full conditional involves only y_i, μ and σ^2, as Step 3 below confirms.

```
par(mar = c(0.5, 0.5, 0.5, 0.5))
nodes_cauchy <- data.frame(
    id    = c("m0", "t2", "a0", "b0", "mu", "s2", "half", "v", "y"),
    label = c("mu[0]", "tau^2", "a[0]", "b[0]", "mu", "sigma^2", "frac(1, 2)", "v[i]", "y[i]"),
    x = c(-1.6, -0.6, 0.6, 1.6, -1.1, 1.1, -2.4, -1.3, 0),
    y = c(2, 2, 2, 2, 1, 1, -0.2, -0.2, -1.2),
    type = c(rep("fixed", 4), "latent", "latent", "fixed", "latent", "observed"))
edges_cauchy <- data.frame(from = c("m0", "t2", "a0", "b0", "half", "mu", "s2", "v"),
                           to   = c("mu", "mu", "s2", "s2", "v", "y", "y", "y"))
```

```
draw_dag(nodes_cauchy, edges_cauchy,
         plates = list(list(x = c(-1.9, 0.6), y = c(-1.9, 0.4), label = "i = 1, ..., n")))
```

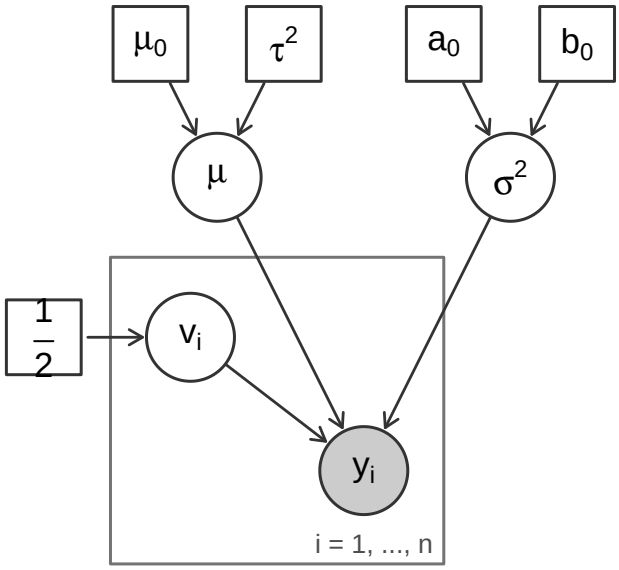


Figure 12.7: Shows the structure of the augmented Cauchy model as a directed graph. Each observation y_i has its own latent variance v_i with a fixed IG(1/2, 1/2) prior; integrating v_i out gives the Cauchy model for y_i.

12.2.5.2 Step 2: the augmented joint posterior

Multiplying priors, the latent-variable densities and the normal likelihood,

$$p(\mu, \sigma^2, v \mid y) \propto e^{-\frac{(\mu-\mu_0)^2}{2\tau^2}} (\sigma^2)^{-(a_0+1)} e^{-b_0/\sigma^2} \times \prod_{i=1}^{n} \Big[(\sigma^2 v_i)^{-1/2} \exp\Big\{ -\frac{(y_i-\mu)^2}{2\sigma^2 v_i} \Big\} \cdot v_i^{-3/2} e^{-1/(2v_i)} \Big]. \tag{12.21}$$

Each full conditional is read off by keeping only the factors that contain the parameter being updated and recognizing the kernel.

12.2.5.3 Step 3: the full conditional of each v_i

The factors containing v_i are those in the i-th term of the product:

$$\begin{aligned} p(v_i \mid \mu, \sigma^2, y) &\propto v_i^{-1/2} \, v_i^{-3/2} \exp\Big\{ -\frac{1}{v_i}\Big(\frac{(y_i-\mu)^2}{2\sigma^2} + \frac{1}{2} \Big) \Big\} \\ &= v_i^{-(1+1)} \exp\Big\{ -\frac{(1 + r_i^2)/2}{v_i} \Big\}. \end{aligned} \tag{12.22}$$

This is the IG(a, b) kernel $v^{-(a+1)}e^{-b/v}$ with $a = 1$ and $b = (1 + r_i^2)/2$:

$$v_i \mid \mu, \sigma^2, y \sim \text{IG}\Big(1,\ \frac{1 + r_i^2}{2}\Big), \qquad i = 1, \dots, n, \tag{12.23}$$

independently across i, since v_i appears in no other factor. The scale $(1 + r_i^2)/2$ grows with the standardized residual r_i, so an outlier gets a large latent variance.

12.2.5.4 Step 4: the full conditional of μ

The factors containing μ are the prior and the exponentials of the likelihood. Writing $w_i = 1/(\sigma^2 v_i)$ for the precision of observation i,

$$\begin{aligned} p(\mu \mid \sigma^2, v, y) &\propto \exp\Big\{ -\frac{(\mu - \mu_0)^2}{2\tau^2} - \sum_i \frac{w_i (y_i - \mu)^2}{2} \Big\} \\ &\propto \exp\Big\{ -\frac{1}{2}\Big[\Big(\frac{1}{\tau^2} + \sum_i w_i \Big)\mu^2 - 2\Big(\frac{\mu_0}{\tau^2} + \sum_i w_i y_i \Big)\mu \Big] \Big\}, \end{aligned} \tag{12.24}$$

after expanding the squares and dropping terms free of μ. Completing the square gives a normal distribution:

$$\begin{aligned} &\mu \mid \sigma^2, v, y \sim N(m, s^2), \\ &\frac{1}{s^2} = \frac{1}{\tau^2} + \sum_{i=1}^{n} w_i, \qquad m = s^2\Big(\frac{\mu_0}{\tau^2} + \sum_{i=1}^{n} w_i y_i \Big). \end{aligned} \tag{12.25}$$

This is a precision-weighted mean of the prior mean and the observations, as in Example 2, except that each observation now has its own weight w_i. An outlier, with a large v_i, has a small weight and barely moves μ; this is where the robustness of the Cauchy model comes from.

12.2.5.5 Step 5: the full conditional of σ^2

The factors containing σ^2 are its prior and the likelihood terms $(\sigma^2)^{-1/2}$ and $\exp\{-(y_i - \mu)^2/(2\sigma^2 v_i)\}$:

$$\begin{aligned} p(\sigma^2 \mid \mu, v, y) &\propto (\sigma^2)^{-(a_0+1)} e^{-b_0/\sigma^2} \cdot (\sigma^2)^{-n/2} \exp\Big\{ -\frac{1}{\sigma^2} \sum_i \frac{(y_i - \mu)^2}{2v_i} \Big\} \\ &= (\sigma^2)^{-(a_0 + \frac{n}{2} + 1)} \exp\Big\{ -\frac{b_0 + \sum_i (y_i - \mu)^2/(2v_i)}{\sigma^2} \Big\}, \end{aligned} \tag{12.26}$$

again an inverse-gamma kernel:

$$\sigma^2 \mid \mu, v, y \sim \text{IG}\Big(a_0 + \frac{n}{2},\ b_0 + \sum_{i=1}^{n} \frac{(y_i - \mu)^2}{2v_i}\Big). \tag{12.27}$$

12.2.5.6 The sampler and a check against the exact posterior

One Gibbs iteration draws all v_i (Step 3), then μ (Step 4), then σ^2 (Step 5), each from its full conditional at the current values of the others. We use $\mu_0 = 0$, $\tau = 10$ and $a_0 = b_0 = 1$, and 30 observations from Cauchy$(2, 1)$, which include several outliers (the largest is above 35). Because the posterior of (μ, σ) is only two-dimensional, it can also be computed exactly on a grid, which gives an independent check of the sampler. On the σ scale the prior density is $p(\sigma^2) \cdot 2\sigma$, by the change of variable $\sigma^2 \to \sigma$.

```
gibbs_cauchy <- function(iters, y, mu0 = 0, tau = 10, a0 = 1, b0 = 1, mu = 0, s2 = 1) {
    n <- length(y)
    out <- matrix(NA_real_, iters, 2, dimnames = list(NULL, c("mu", "sigma")))
    for (t in seq_len(iters)) {
        ### Step 3: v_i | mu, sigma^2 ~ IG(1, (1 + r_i^2) / 2), drawn as 1 / Gamma
        v <- 1 / rgamma(n, shape = 1, rate = (1 + (y - mu)^2 / s2) / 2)
        ### Step 4: mu | sigma^2, v ~ N(m, s^2), with observation weights w_i = 1 / (sigma^2
        ↪  v_i)
        w <- 1 / (s2 * v)
        prec <- 1 / tau^2 + sum(w)
        mu <- rnorm(1, (mu0 / tau^2 + sum(w * y)) / prec, sqrt(1 / prec))
        ### Step 5: sigma^2 | mu, v ~ IG(a0 + n / 2, b0 + sum (y_i - mu)^2 / (2 v_i))
        s2 <- 1 / rgamma(1, shape = a0 + n / 2, rate = b0 + sum((y - mu)^2 / (2 * v)))
        out[t, ] <- c(mu, sqrt(s2))
    }
    out
}

### exact log posterior of (mu, sigma), up to a constant, for the grid check
log_post_cauchy <- function(mu, sigma, y, mu0 = 0, tau = 10, a0 = 1, b0 = 1) {
    sum(dcauchy(y, mu, sigma, log = TRUE)) + dnorm(mu, mu0, tau, log = TRUE) -
        (a0 + 1) * log(sigma^2) - b0 / sigma^2 + log(2 * sigma)
}

set.seed(7)
y_cauchy <- rcauchy(30, location = 2, scale = 1)
draws_cauchy <- gibbs_cauchy(20000, y_cauchy, mu = -5, s2 = 25)   # deliberately poor start
keep_c <- -(1:1000)

mu_grid <- seq(0, 4, length.out = 300); sg_grid <- seq(0.2, 3.5, length.out = 300)
lp <- outer(mu_grid, sg_grid, Vectorize(function(m, s) log_post_cauchy(m, s, y_cauchy)))
post <- exp(lp - max(lp)); post <- post / sum(post)
```

```
par(mfrow = c(1, 3), mar = c(4, 4, 2, 1))
contour(mu_grid, sg_grid, post, nlevels = 8, drawlabels = FALSE, col = "gray55",
        xlim = c(-5.5, 4), ylim = c(0, 5.5), xlab = expression(mu), ylab = expression(sigma),
        main = "first 15 iterations")
path <- rbind(c(-5, 5), draws_cauchy[1:15, ])
for (t in 1:15) {   # mu moves first (horizontal leg), then sigma (vertical leg)
    segments(path[t, 1], path[t, 2], path[t + 1, 1], path[t, 2], col = "darkorange", lwd = 1.5)
    segments(path[t + 1, 1], path[t, 2], path[t + 1, 1], path[t + 1, 2], col = "steelblue", lwd
    ↪  = 1.5)
```

```
}
points(-5, 5, pch = 17, cex = 1.4)
plot(draws_cauchy[1:500, "mu"], type = "l", col = "darkorange", xlab = "iteration",
     ylab = expression(mu), main = "trace of mu")
plot(draws_cauchy[1:500, "sigma"], type = "l", col = "steelblue", xlab = "iteration",
     ylab = expression(sigma), main = "trace of sigma")
```

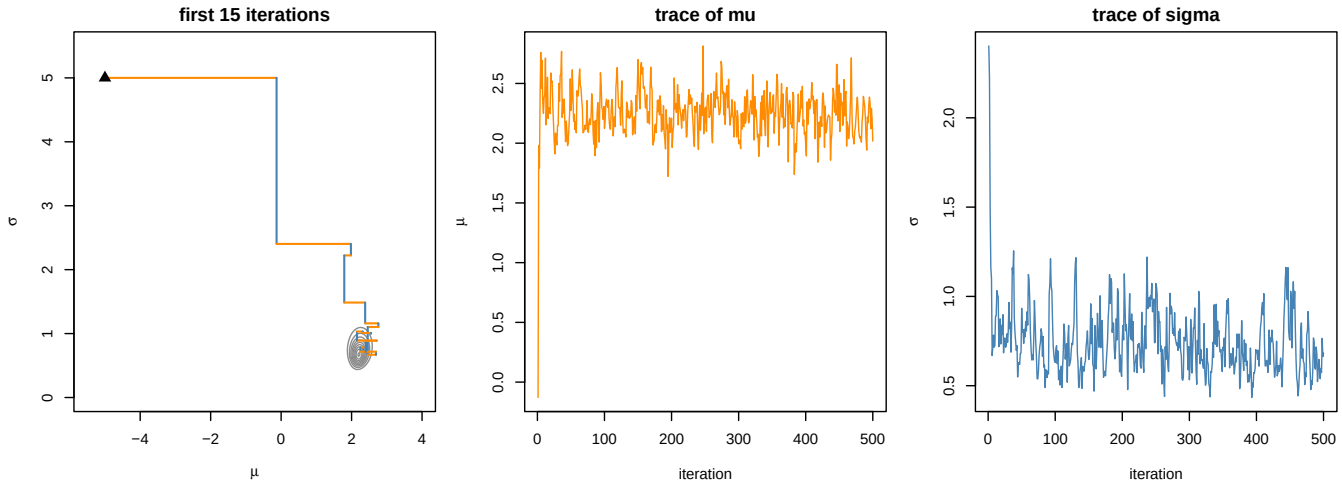


Figure 12.8: Demonstrates the data-augmentation Gibbs sampler for Cauchy data with unknown location and scale, checked against the exact posterior. Left: the first 15 iterations from a poor start (triangle), drawn as the staircase of mu- and sigma-updates, over the contours of the exact posterior computed on a grid. Middle and right: trace plots of mu and sigma for the first 500 iterations; the chain reaches the posterior within a few iterations.

```
grid_mean <- function(g, margin) sum(apply(post, margin, sum) * g)
grid_sd   <- function(g, margin) sqrt(sum(apply(post, margin, sum) * g^2) - grid_mean(g,
 ↪  margin)^2)
knitr::kable(data.frame(
    quantity = c("posterior mean of mu", "posterior sd of mu",
                 "posterior mean of sigma", "posterior sd of sigma"),
    gibbs = c(mean(draws_cauchy[keep_c, "mu"]), sd(draws_cauchy[keep_c, "mu"]),
              mean(draws_cauchy[keep_c, "sigma"]), sd(draws_cauchy[keep_c, "sigma"])),
    exact = c(grid_mean(mu_grid, 1), grid_sd(mu_grid, 1),
              grid_mean(sg_grid, 2), grid_sd(sg_grid, 2))
), digits = 3, col.names = c("quantity", "Gibbs", "exact (grid)"), align = c("l", "r", "r"))
```

Table 12.4: Checks the data-augmentation Gibbs sampler for the Cauchy model against the exact posterior computed on a grid. Posterior means and standard deviations of mu and sigma from 19,000 Gibbs draws (after a 1,000-iteration burn-in) and from the exact grid posterior. The sample median and the sample standard deviation are shown for contrast: the outliers inflate the standard deviation, while the Cauchy posterior is not affected by them.

quantity	Gibbs	exact (grid)
posterior mean of mu	2.228	2.228
posterior sd of mu	0.179	0.179
posterior mean of sigma	0.768	0.765
posterior sd of sigma	0.181	0.178

The Gibbs estimates agree with the exact posterior to Monte Carlo error. For comparison, the sample mean of these data is 3.68 and the sample standard deviation is 6.43, both dragged far off by the outliers, while the posterior mean of μ stays near the centre of the bulk of the data. A shinylive app runs this sampler one conditional update at a time: it shows the v, μ and σ updates as separate steps on the exact posterior, and the latent variances v_i of the outliers.

12.3 Metropolis–Hastings

12.3.1 The algorithm

Gibbs sampling needs full conditionals that are both known and directly sample-able — a strong requirement outside conjugate models. **Metropolis–Hastings (MH)** needs only a target $\pi(\theta)$ computable up to a constant and *any* proposal distribution $\tilde{T}(\theta^* \mid \theta)$ we can sample from. Starting from $\theta^{(0)}$, repeat:

1. Draw a proposal $\theta^* \sim \tilde{T}(\cdot \mid \theta^{(t-1)})$.
2. Compute the acceptance ratio $r = \min\left\{\dfrac{\pi(\theta^*)\,\tilde{T}(\theta^{(t-1)} \mid \theta^*)}{\pi(\theta^{(t-1)})\,\tilde{T}(\theta^* \mid \theta^{(t-1)})},\ 1\right\}$.
3. With probability r set $\theta^{(t)} = \theta^*$ (accept); otherwise set $\theta^{(t)} = \theta^{(t-1)}$ (reject **and repeat the current value**).

Only the ratio $\pi(\theta^*)/\pi(\theta^{(t-1)})$ is needed, so any normalizing constant of π cancels — exactly as in rejection sampling, but here a rejected proposal still contributes a draw (the repeated current value) rather than being discarded, which is what lets MH work even when π has no available envelope. One can show (by checking detailed balance directly, splitting into the "propose and accept" and "reject and stay" cases and using the $\min\{\cdot,\cdot\}$ identity $\min\{a,b\} = \min\{b,a\}$) that this T leaves π invariant regardless of what $\tilde{T}$ is, as long as its support is compatible with π's.

12.3.2 Random-walk Metropolis

If $\tilde{T}$ is symmetric, $\tilde{T}(\theta^* \mid \theta) = \tilde{T}(\theta \mid \theta^*)$ — e.g. $\theta^* \sim N(\theta^{(t-1)}, \varsigma^2)$ — the proposal terms cancel and $r = \min\{\pi(\theta^*)/\pi(\theta^{(t-1)}), 1\}$: uphill proposals are always accepted, downhill ones accepted in proportion to how much worse they are. ς, the **step size**, is the only tuning parameter.

```
met_gauss <- function(iters, log_f, stepsizes, ini_value, ...) {
    state <- ini_value; no_var <- length(state)
    logf <- log_f(ini_value, ...)
    if (!is.finite(logf)) stop("Initial value has zero probability")
    out <- matrix(NA_real_, iters, no_var); n_acc <- 0
    for (t in seq_len(iters)) {
        proposal <- rnorm(no_var, state, stepsizes)
        new_logf <- log_f(proposal, ...)
        if (log(runif(1)) < new_logf - logf) {
            state <- proposal; logf <- new_logf; n_acc <- n_acc + 1
        }
        out[t, ] <- state
    }
    attr(out, "accept_rate") <- n_acc / iters
    out
}
```

12.3.3 Choosing the step size

The chain's draws are dependent, and what determines the precision of an ergodic average is not the acceptance rate itself but the resulting **autocorrelation**: for a stationary chain with lag-k autocorrelation ρ_k,

$$\begin{aligned} \operatorname{Var}\Big(\tfrac{1}{T}\sum_{t=1}^{T}\theta^{(t)}\Big) &\approx \frac{\sigma^2}{T}\Big(1+2\sum_{k\ge 1}\rho_k\Big) \\ &= \frac{\sigma^2}{T_{\text{eff}}}, \end{aligned} \tag{12.28}$$

so the tuning goal is small autocorrelation, not a high acceptance rate as such. A step too small is accepted almost every time but crawls (highly autocorrelated); a step too large is almost always rejected, so the chain barely moves either. Both extremes waste the same T iterations on far fewer effective ones — the figure below (three step sizes on a $N(0,1)$ target) makes this visible directly in the trace and the ACF.

```
mh_std_normal <- function(sig, n = 2000) {
    th <- numeric(n); th[1] <- 0; acc <- 0
    for (t in 2:n) {
        p <- th[t - 1] + rnorm(1, 0, sig)
        if (runif(1) < dnorm(p) / dnorm(th[t - 1])) { th[t] <- p; acc <- acc + 1 } else th[t]
         ↪  <- th[t - 1]
    }
    list(th = th, acc = acc / (n - 1))
}
res <- lapply(c(0.1, 2.4, 40), mh_std_normal)
labs <- c("too small", "about right", "too large")
par(mfrow = c(2, 3), mar = c(4, 4, 2, 1))
for (k in 1:3) plot(res[[k]]$th[1:500], type = "l", col = "steelblue", xlab = "t", ylab =
 ↪  expression(theta^(t)),
                    ylim = c(-3.5, 3.5), main = sprintf("%s: accept %.2f", labs[k],
                     ↪  res[[k]]$acc))
```

```r
for (k in 1:3) acf(res[[k]]$th, lag.max = 50, main = "ACF")
```

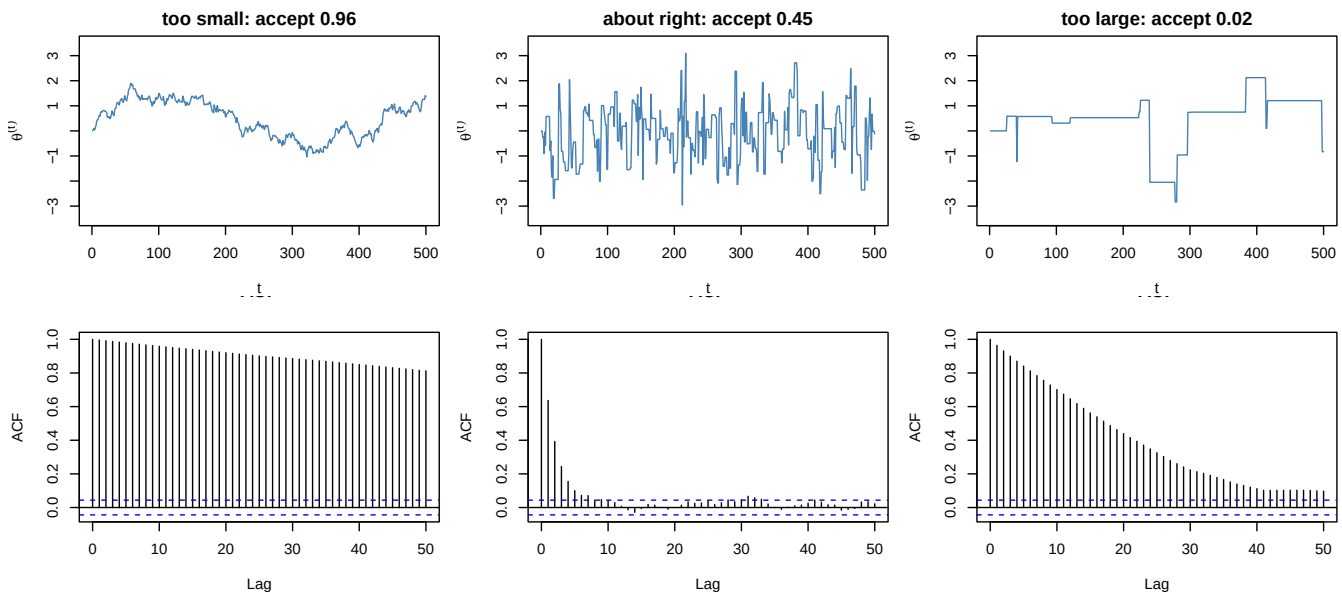


Figure 12.9: Illustrates the trade-off in choosing the step size of a random-walk Metropolis-Hastings sampler, using trace plots and autocorrelation. Step size too small, about right, and too large: trace (top) and autocorrelation (bottom) for a random walk targeting N(0, 1).

For random-walk Metropolis on a smooth target, theory recommends an acceptance rate near 0.234 in high dimension (up to about 0.44 in one dimension) as the rate that minimizes autocorrelation for a given computational budget — a useful rule of thumb, not a target to hit exactly.

12.3.4 Metropolis-within-Gibbs

The two ideas combine directly. If $\theta = (\theta_1, \theta_2)$ and $\pi(\theta_1 \mid \theta_2)$ has no convenient sampler while $\pi(\theta_2 \mid \theta_1)$ does, replace only the first Gibbs step by a Metropolis–Hastings update that targets $\pi(\theta_1 \mid \theta_2)$ (evaluated at the *current* θ_2) and leave the second step as an exact conditional draw. Each block transition leaves π invariant on its own — the Gibbs argument applies verbatim to any $\pi(\theta_j \mid \theta_{-j})$-invariant transition, not only an exact conditional draw — so the composite sweep leaves the full joint π invariant. This "Gibbs where conjugate, Metropolis elsewhere" pattern, iterated one parameter or block at a time, is how most general-purpose Bayesian software before Hamiltonian methods actually worked.

12.3.5 Example 5: A Normal Truncated to Its Tail

The problem. Draw from $N(0, 2^2)$ truncated to $[4, 5]$, a sliver of the far right tail,

$$\pi(\theta) \propto \varphi\Big(\frac{\theta}{2}\Big)\, I(4 \le \theta \le 5). \tag{12.29}$$

Sampling from $N(0, 2^2)$ and keeping the draws that land in $[4, 5]$ is exact but wasteful: only a fraction $\Phi(2.5) - \Phi(2) \approx 0.017$ of the draws is kept, about one in 60, and a target further in the tail would be worse still.

Random-walk Metropolis. The indicator needs no special handling: a proposal landing outside $[4, 5]$ has $\pi(\theta^*) = 0$, hence $r = 0$, and is automatically rejected, while inside the interval the acceptance probability is the ratio of normal densities. Starting inside the interval, the chain never leaves it, and every iteration produces a draw. Figure 12.10 runs `met_gauss` with three step sizes and reports the acceptance rate and the effective sample size T_{eff} of each chain.

```
log_tnorm <- function(x, mu, sigma, lb, ub) {
    if (x > lb && x < ub) dnorm(x, mu, sigma, log = TRUE) else -Inf
}
set.seed(6); steps <- c(0.1, 1, 2)
mcs <- lapply(steps, function(st) met_gauss(5000, log_tnorm, stepsizes = st, ini_value = 4.5,
                                            mu = 0, sigma = 2, lb = 4, ub = 5))
ess_fn <- function(th) {                  # effective sample size T / (1 + 2 sum rho_k), summed
 ↪  to the first rho_k < 0
    rho <- acf(th, lag.max = 200, plot = FALSE)$acf[-1]
    k <- which(rho < 0)[1]; if (is.na(k)) k <- length(rho) + 1
    length(th) / (1 + 2 * sum(rho[seq_len(k - 1)]))
}
ess <- sapply(mcs, ess_fn)
par(mfrow = c(1, 3), mar = c(4, 4, 2, 1))
for (k in 1:3) plot(mcs[[k]], type = "l", col = "steelblue", xlab = "iteration", ylab =
 ↪  expression(theta),
    main = sprintf("step = %.1f, accept = %.2f, ESS = %.0f", steps[k], attr(mcs[[k]],
     ↪  "accept_rate"), ess[k]))
```

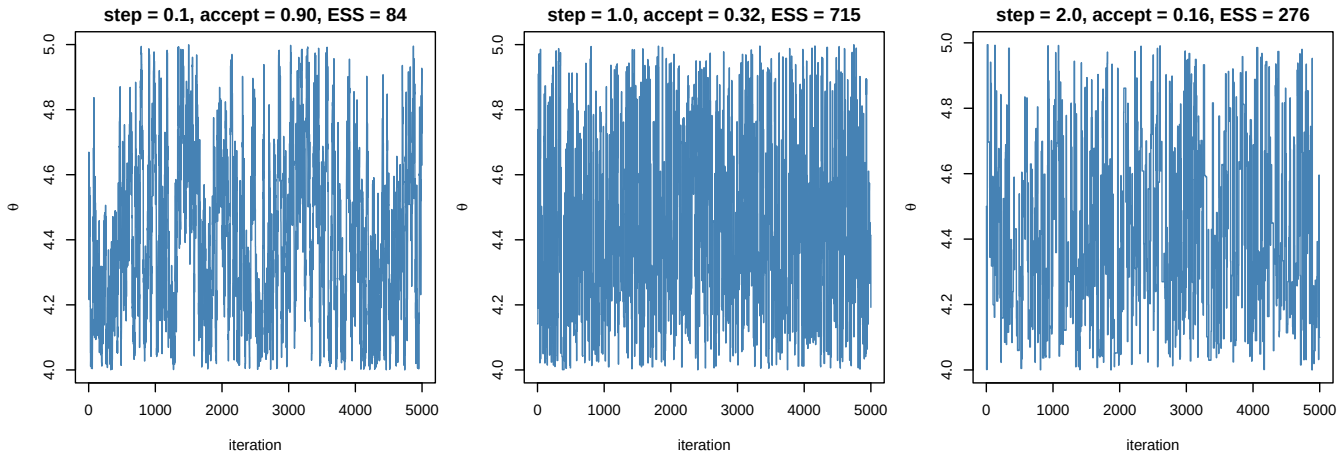


Figure 12.10: Demonstrates random-walk Metropolis-Hastings on a normal distribution truncated to its tail, where sampling the untruncated normal and keeping the draws in range is wasteful. Random-walk MH targeting N(0, 2^2) truncated to [4, 5], for three step sizes, with the acceptance rate and effective sample size (ESS) of each chain of 5,000 iterations.

The three chains differ in how efficiently they explore $[4, 5]$, as measured by the effective sample size; as in Figure 12.9, the smallest step is accepted most often but moves least. The chain with the largest ESS, step 1,

reproduces the exact truncated density (Figure 12.11), including its decline across the interval: φ is decreasing there, so states near 4 are visited about 3.1 times as often as states near 5.

```
best <- which.max(ess); th_best <- mcs[[best]][, 1]
par(mar = c(4, 4, 2, 1))
hist(th_best, breaks = 40, freq = FALSE, col = "gray85", border = "white", xlim = c(3.9, 5.1),
     xlab = expression(theta), ylab = "density",
     main = sprintf("step = %.1f (accept %.2f)", steps[best], attr(mcs[[best]],
      ↪  "accept_rate")))
curve(dnorm(x, 0, 2) / (pnorm(5, 0, 2) - pnorm(4, 0, 2)), 4, 5, add = TRUE, col = "firebrick",
 ↪  lwd = 2, lty = 2)
legend("topright", bty = "n", lwd = c(NA, 2), lty = c(NA, 2), pch = c(15, NA), pt.cex = 2,
       col = c("gray85", "firebrick"), legend = c("histogram of the draws", "exact truncated
        ↪  normal density"))
```

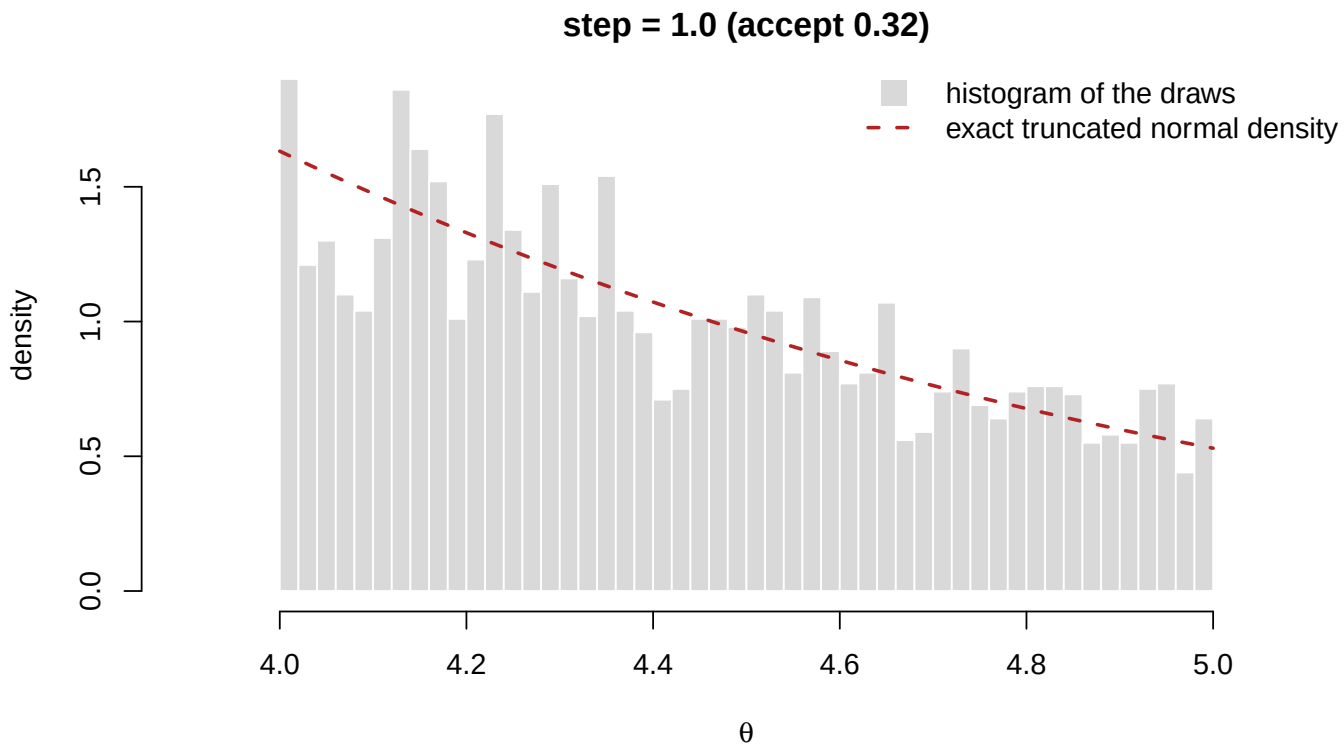


Figure 12.11: Checks the random-walk Metropolis-Hastings sampler for the truncated normal against the exact density. Histogram of the 5,000 states of the chain with the largest effective sample size, with the exact N(0, 2^2) density truncated to [4, 5] (dashed).

12.3.6 Example 6: Normal Data with Unknown Mean and Variance, Revisited

This is the model, prior and data of Example 2,

$$x_1, \dots, x_n \mid \mu, \sigma^2 \overset{iid}{\sim} N(\mu, \sigma^2), \qquad \mu \sim N(\mu_0, \sigma_0^2), \qquad \sigma^2 \sim \text{IG}(a_0, \lambda_0), \tag{12.30}$$

now sampled by random-walk Metropolis–Hastings.

The model on the log scale. $\sigma^2 > 0$ makes a symmetric normal random walk directly on σ^2 wasteful (many proposals land below zero and are rejected), so reparametrize

$$w = \log \sigma^2 \in \mathbb{R}, \qquad \sigma^2 = e^w. \tag{12.31}$$

By the change-of-variables formula, if $X \sim f_X$ and $Y = t(X)$, then $f_Y(y) = f_X(t^{-1}(y)) \left| \frac{dt^{-1}(y)}{dy} \right|$. Here $t^{-1}(w) = e^w$, so the Jacobian is e^w, and

$$P(\mu, w \mid x) \;\propto\; \underbrace{e^{-\frac{(\mu-\mu_0)^2}{2\sigma_0^2}}}_{p(\mu)} \cdot \underbrace{(e^w)^{-(a_0+1)} e^{-\lambda_0 e^{-w}}}_{p_{\sigma^2}(e^w)} \cdot \underbrace{(e^w)^{-n/2} e^{-\sum_i (x_i-\mu)^2 e^{-w}/2}}_{\text{likelihood}} \cdot \underbrace{e^w}_{\text{Jacobian}} . \tag{12.32}$$

Collecting the powers of e^w, $-(a_0+1) - \frac{n}{2} + 1 = -(a_0 + \frac{n}{2})$, so

$$\log P(\mu, w \mid x) = -\frac{(\mu-\mu_0)^2}{2\sigma_0^2} - \Big(a_0 + \frac{n}{2}\Big) w - \Big(\lambda_0 + \frac{1}{2}\sum_{i=1}^{n} (x_i - \mu)^2\Big) e^{-w} + \text{const.} \tag{12.33}$$

Omitting the Jacobian e^w would target the wrong distribution.

The sampler. A random-walk proposal on (μ, w) jointly, with `met_gauss` from above; the log posterior in the code is the formula above, term by term:

```
log_post_mu_w <- function(mu, w)                    # prior constants mu0, s02, a0, lam0 from
 ↪  Example 2
    -(mu - mu0)^2 / (2 * s02) - (a0 + length(x) / 2) * w - (lam0 + sum((x - mu)^2) / 2) *
     ↪  exp(-w)
mh_norm <- function(iters, stepsizes) {
    mc <- met_gauss(iters, function(th) log_post_mu_w(th[1], th[2]), stepsizes, ini_value =
 ↪  c(0, 0))
    list(mu = mc[, 1], sd = exp(mc[, 2] / 2), accept_rate = attr(mc, "accept_rate"))   # sigma
     ↪  = e^{w/2}
}
```

```
set.seed(5)
scales <- c(20, 10, 4, 2)
fits <- lapply(scales, function(s) mh_norm(10000, c(s, s) / sqrt(length(x))))
par(mfcol = c(2, 4), mar = c(4, 4, 2, 1))
for (k in seq_along(fits)) {
    plot(fits[[k]]$mu[1:1000], type = "l", col = "steelblue", xlab = "t", ylab =
     ↪  expression(mu),
         main = sprintf("scale %g, accept %.2f", scales[k], fits[[k]]$accept_rate))
    acf(fits[[k]]$mu[-(1:500)], main = "ACF")
}
```

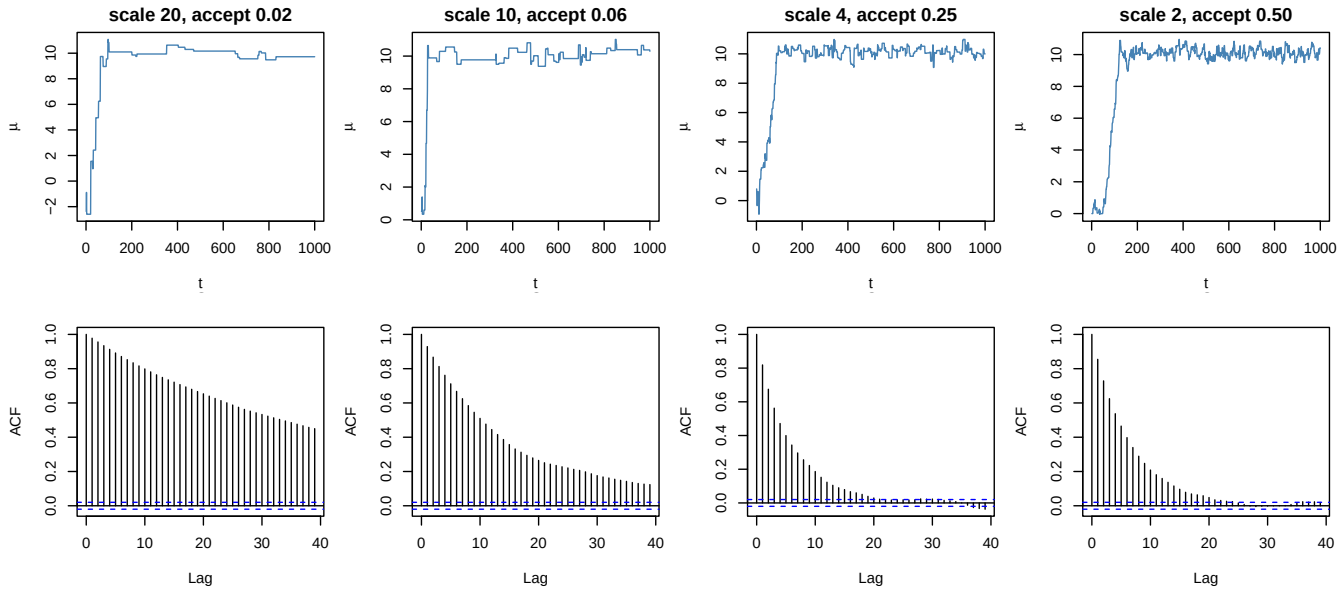


Figure 12.12: Illustrates how to tune the step size in practice, by comparing short preliminary runs at several scales. Preliminary MH runs at four step-size scales, targeting mu (top: trace, bottom: ACF after discarding 500 iterations).

Scale 4 gives the least autocorrelated trace among the four, consistent with an acceptance rate closer to the 0.2–0.4 range than the very low or very high rates at the extremes. The posterior densities estimated from that chain already match those of the Gibbs sampler (Figure 12.13).

```
best5 <- fits[[3]]; keep5 <- -(1:500)
par(mfrow = c(1, 2), mar = c(4, 4, 2, 1))
plot(density(best5$mu[keep5]), col = "steelblue", lwd = 2, xlab = expression(mu), main =
 ↪  "posterior density of mu")
lines(density(mu_g), col = "gray30", lwd = 2, lty = 2)
legend("topright", bty = "n", lwd = 2, lty = c(1, 2), col = c("steelblue", "gray30"), legend =
 ↪  c("MH, scale 4", "Gibbs"))
plot(density(best5$sd[keep5]), col = "firebrick", lwd = 2, xlab = expression(sigma), main =
 ↪  "posterior density of sigma")
lines(density(sd_g), col = "gray30", lwd = 2, lty = 2)
legend("topright", bty = "n", lwd = 2, lty = c(1, 2), col = c("firebrick", "gray30"), legend =
 ↪  c("MH, scale 4", "Gibbs"))
```

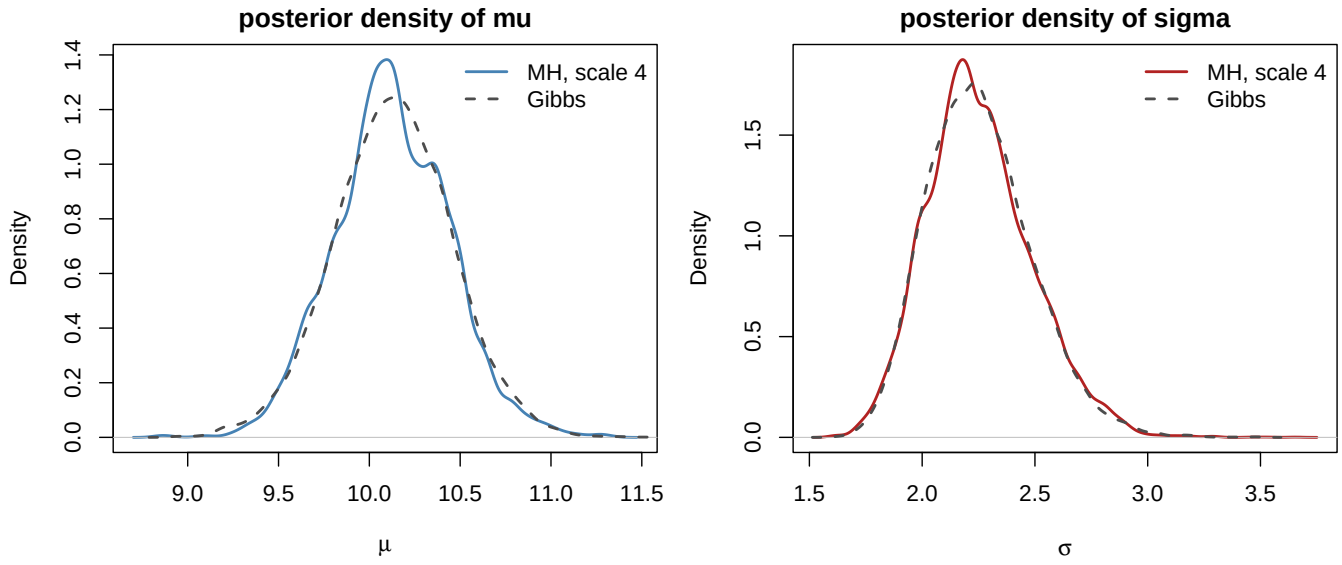


Figure 12.13: Compares the posterior densities from the best-tuned Metropolis-Hastings chain with those from the Gibbs sampler. Kernel densities of mu and sigma from the scale-4 chain of the preliminary runs (10,000 iterations, first 500 discarded; solid) and from the Gibbs sampler of Example 2 (dashed).

Running a long chain at that scale and comparing to the Gibbs sampler on the same data:

```
mh_long <- mh_norm(50000, c(4, 4) / sqrt(length(x)))
keep_mh <- -(1:1000)
coherence <- data.frame(
    method = c("Metropolis-Hastings", "Gibbs"),
    mean_mu = c(mean(mh_long$mu[keep_mh]), mean(mu_g)),
    sd_mu   = c(sd(mh_long$mu[keep_mh]), sd(mu_g)),
    mean_sigma = c(mean(mh_long$sd[keep_mh]), mean(sd_g)),
    sd_sigma   = c(sd(mh_long$sd[keep_mh]), sd(sd_g))
)
knitr::kable(coherence, digits = 3, align = c("l", "r", "r", "r", "r"))
```

Table 12.5: Checks the Metropolis-Hastings sampler against Gibbs sampling on the same problem. Posterior mean and sd from Metropolis-Hastings versus Gibbs sampling, same data and priors: the two exact samplers of the same posterior agree up to Monte Carlo error.

method	mean_mu	sd_mu	mean_sigma	sd_sigma
Metropolis-Hastings	10.128	0.320	2.257	0.225
Gibbs	10.130	0.324	2.255	0.230

12.4 Hamiltonian Monte Carlo and Stan

12.4.1 Why random walks struggle

A random-walk proposal has no sense of direction: reaching a point one posterior standard deviation away takes $O(d)$ steps in d dimensions, and far more along a narrow, correlated ridge, since most proposed directions immediately step off the ridge and get rejected. Gibbs suffers the same way when parameters are strongly correlated, since each step moves along only one coordinate axis. **Hamiltonian Monte Carlo (HMC)** uses the *gradient* $\nabla \log \pi(\theta)$ to propose points far away with high probability of acceptance.

Introduce an auxiliary **momentum** variable $p \in \mathbb{R}^d$ and the joint density $\pi(\theta, p) \propto \exp\{-H(\theta, p)\}$ with

$$H(\theta, p) = \underbrace{-\log \pi(\theta)}_{\text{potential } U(\theta)} + \underbrace{\tfrac{1}{2} p^\top M^{-1} p}_{\text{kinetic } K(p)}, \tag{12.34}$$

so that the θ-marginal is exactly the target and $p \sim N_d(0, M)$ independently. Physically: a particle at position θ with momentum p rolling frictionlessly on the potential surface $U(\theta)$. Hamilton's equations move (θ, p) so as to *conserve* H exactly and are volume-preserving and reversible; discretizing them with the **leapfrog** integrator (step size ϵ, L steps),

$$\begin{aligned} p &\leftarrow p + \tfrac{\epsilon}{2} \nabla \log \pi(\theta), \qquad \theta \leftarrow \theta + \epsilon M^{-1} p, \\ p &\leftarrow p + \tfrac{\epsilon}{2} \nabla \log \pi(\theta), \end{aligned} \tag{12.35}$$

keeps volume-preservation and reversibility exactly (negating p and re-running gives back the start) while introducing only an $O(\epsilon^2)$ error in H, which a Metropolis accept/reject step then corrects exactly.

12.4.2 The algorithm

Starting from $\theta^{(0)}$, repeat: draw a fresh $p \sim N_d(0, M)$; run L leapfrog steps from $(\theta^{(t-1)}, p)$ to reach (θ^*, p^*); accept $\theta^{(t)} = \theta^*$ with probability $\min\{\exp[H(\theta^{(t-1)}, p) - H(\theta^*, p^*)], 1\}$, else repeat $\theta^{(t-1)}$. Drawing p is a Gibbs step from its own conditional (it doesn't depend on θ), and the leapfrog-plus-accept step is a Metropolis update with a deterministic, reversible proposal — so each half leaves $\pi(\theta, p)$ invariant, and because H is nearly conserved, acceptance stays high (typically 0.65–0.8) even though θ^* may be far from $\theta^{(t-1)}$.

```
hmc <- function(log_pi, grad_log_pi, theta, eps, L, iters, M = diag(length(theta))) {
    d <- length(theta); Minv <- solve(M); R <- chol(M)
    out <- matrix(NA_real_, iters, d); n_acc <- 0
    for (t in seq_len(iters)) {
        p0 <- as.vector(t(R) %*% rnorm(d))
        th <- theta; p <- p0 + eps / 2 * grad_log_pi(th)
        for (l in seq_len(L)) {
            th <- th + eps * as.vector(Minv %*% p)
            if (l < L) p <- p + eps * grad_log_pi(th)
        }
        p <- p + eps / 2 * grad_log_pi(th)
        H0 <- -log_pi(theta) + 0.5 * sum(p0 * (Minv %*% p0))
        H1 <- -log_pi(th)    + 0.5 * sum(p  * (Minv %*% p))
        if (log(runif(1)) < H0 - H1) { theta <- th; n_acc <- n_acc + 1 }
        out[t, ] <- theta
```

```
    }
    attr(out, "accept_rate") <- n_acc / iters
    out
}
```

12.4.3 Example 7: Bivariate Normal with Correlation 0.98

A strongly correlated bivariate normal is the simplest target on which a random walk struggles. Both samplers below get the same budget of 2,000 iterations and start from the same point.

```
set.seed(4)
Sigma <- matrix(c(1, 0.98, 0.98, 1), 2); Sinv <- solve(Sigma)
log_pi_corr  <- function(z) -0.5 * sum(z * (Sinv %*% z))
grad_pi_corr <- function(z) -as.vector(Sinv %*% z)

rw <- matrix(NA_real_, 2000, 2); cur <- c(-2, -2); lp <- log_pi_corr(cur); acc <- 0
for (t in 1:2000) {
    pr <- cur + 0.3 * rnorm(2); lpp <- log_pi_corr(pr)
    if (log(runif(1)) < lpp - lp) { cur <- pr; lp <- lpp; acc <- acc + 1 }
    rw[t, ] <- cur
}
draws_hmc <- hmc(log_pi_corr, grad_pi_corr, theta = c(-2, -2), eps = 0.05, L = 30, iters =
↪  2000)

g <- seq(-3.5, 3.5, length.out = 80)
Z <- outer(g, g, Vectorize(function(a, b) exp(-0.5 * c(a, b) %*% Sinv %*% c(a, b))))
par(mfrow = c(1, 3), mar = c(4, 4, 2, 1))
for (nm in c("random walk", "HMC")) {
    d <- if (nm == "random walk") rw else draws_hmc
    ar <- if (nm == "random walk") acc / 2000 else attr(draws_hmc, "accept_rate")
    contour(g, g, Z, nlevels = 6, drawlabels = FALSE, col = "gray60",
            xlab = expression(theta[1]), ylab = expression(theta[2]),
            main = sprintf("%s: accept %.2f", nm, ar))
    lines(d[1:100, ], col = if (nm == "random walk") "firebrick" else "steelblue", lwd = 1.5)
    points(d[1:100, ], pch = 19, cex = 0.5, col = if (nm == "random walk") "firebrick" else
    ↪  "steelblue")
}
a1 <- acf(rw[, 1], lag.max = 60, plot = FALSE)$acf
a2 <- acf(draws_hmc[, 1], lag.max = 60, plot = FALSE)$acf
plot(0:60, a1, type = "h", lwd = 2, col = "firebrick", xlab = "lag", ylab = "ACF", ylim =
↪  c(-0.2, 1), main = "autocorrelation")
lines((0:60) + 0.3, a2, type = "h", lwd = 2, col = "steelblue")
legend("topright", bty = "n", lwd = 2, col = c("firebrick", "steelblue"), legend = c("random
↪  walk", "HMC"))
```

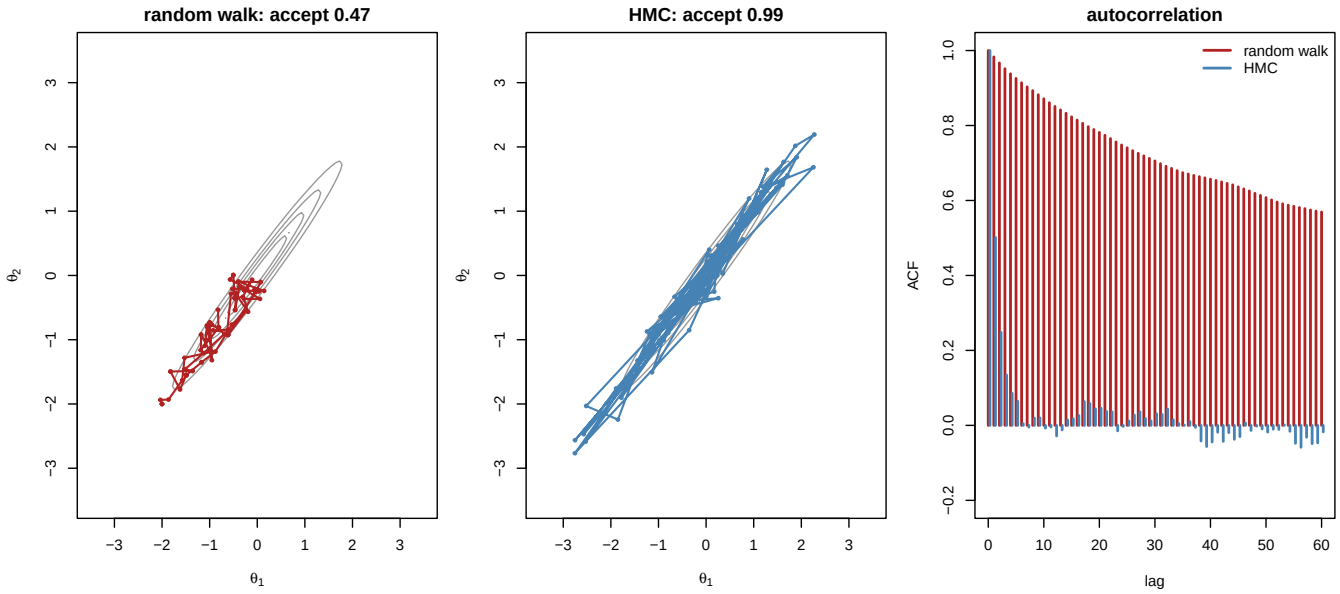


Figure 12.14: Compares Hamiltonian Monte Carlo with random-walk Metropolis on a highly correlated target, where the random walk struggles. Target: bivariate normal with correlation 0.98. Left/middle: first 100 draws over the posterior contours for random-walk Metropolis and HMC, same iteration budget. Right: autocorrelation of the first coordinate – HMC crosses the ridge in one trajectory and its draws decorrelate almost immediately.

Chi Feng's MCMC simulators let you tune the step size and number of leapfrog steps of Hamiltonian Monte Carlo, and compare it with other samplers, to see how they affect its exploration of a correlated target.

12.4.4 Stan: automating HMC

Every sampler above needed something derived by hand: Gibbs needed full conditionals worked out algebraically; Metropolis–Hastings needed only a target density but still needed step sizes tuned by trial and error; HMC needed an analytic gradient and needed ϵ and L tuned by hand. **Stan** is a probabilistic programming language that automates all of this: given a model written declaratively (data, parameters, and a log-density), it computes gradients by automatic differentiation and runs an adaptive variant of HMC — the **No-U-Turn Sampler (NUTS)**, which picks a good L on the fly by extending the leapfrog trajectory until it starts to double back — with step size tuned during a warm-up phase. The user never derives a conditional or a gradient.

12.4.5 Where Stan earns its keep

The normal-mean/variance model of Examples 2 and 6 is simple enough that Gibbs sampling is not just possible but easy, so Stan's advantage is not visible yet. Its real value shows up on models where hand-deriving conditionals or gradients would be a serious undertaking: hierarchical models with many levels of latent structure, time-series models with an unobserved latent state at every time point, or anything with a non-conjugate likelihood/prior combination. Example 9 gives a small taste of this: the Cauchy model needed a scale-mixture trick and three derivations for Gibbs sampling, but only its model statement in Stan.

12.4.6 Example 8: Normal Data with Unknown Mean and Variance, in Stan

To confirm Stan lands on the same posterior as the Gibbs and MH samplers of Examples 2 and 6, fit the identical normal-mean/variance model with the identical priors and data:

```
stan_normal_code <- "
data {
  int<lower=0> n;
  vector[n] x;
  real mu0;
  real<lower=0> s02;
  real<lower=0> a0;
  real<lower=0> lam0;
}
parameters {
  real mu;
  real<lower=0> sigma2;
}
model {
  mu ~ normal(mu0, sqrt(s02));
  sigma2 ~ inv_gamma(a0, lam0);
  x ~ normal(mu, sqrt(sigma2));
}
"
```

```
fit_stan <- stan(
    model_code = stan_normal_code,
    data = list(n = length(x), x = x, mu0 = mu0, s02 = s02, a0 = a0, lam0 = lam0),
    chains = 4, iter = 2000, warmup = 1000, seed = 5, refresh = 0
)
```

```
stan_summary <- summary(fit_stan, pars = c("mu", "sigma2"))$summary
stan_sigma_draws <- sqrt(as.matrix(fit_stan, pars = "sigma2")[, 1])
three_way <- data.frame(
    method = c("Gibbs", "Metropolis-Hastings", "Stan (NUTS)"),
    mean_mu = c(mean(mu_g), mean(mh_long$mu[keep_mh]), stan_summary["mu", "mean"]),
    sd_mu   = c(sd(mu_g), sd(mh_long$mu[keep_mh]), stan_summary["mu", "sd"]),
    mean_sigma = c(mean(sd_g), mean(mh_long$sd[keep_mh]), mean(stan_sigma_draws)),
    sd_sigma   = c(sd(sd_g), sd(mh_long$sd[keep_mh]), sd(stan_sigma_draws))
)
knitr::kable(three_way, digits = 3, align = c("l", "r", "r", "r", "r"))
```

Table 12.6: Validates Stan's sampler against the samplers written by hand earlier in the chapter. Posterior mean and sd: hand-written Gibbs and Metropolis-Hastings versus Stan's NUTS sampler, same data, priors, and model. All three samplers target the identical posterior and agree up to Monte Carlo error.

method	mean_mu	sd_mu	mean_sigma	sd_sigma
Gibbs	10.141	0.330	2.293	0.239
Metropolis-Hastings	10.141	0.325	2.298	0.236
Stan (NUTS)	10.132	0.326	2.295	0.245

The three completely independent implementations — hand-derived conditionals, a hand-tuned random walk, and Stan's automatic NUTS — agree to within Monte Carlo noise, which is exactly the coherence check this book has used throughout: independent methods with no shared code path landing on the same answer is evidence that none of them is wrong.

12.4.7 Example 9: Cauchy Data with Unknown Location and Scale, in Stan

Example 4 needed a page of algebra: a scale-mixture representation, an augmented posterior, and three full conditionals. In Stan none of that is needed. The model is written exactly as stated in Example 4, with the Cauchy likelihood used directly and no latent variances, and NUTS samples the two-dimensional posterior of (μ, σ^2) using gradients from automatic differentiation. The data and priors are those of Example 4: $\mu \sim N(0, 10^2)$, $\sigma^2 \sim \text{IG}(1, 1)$ and the same 30 observations.

```
stan_cauchy_code <- "
data {
  int<lower=0> n;
  vector[n] y;
  real mu0;
  real<lower=0> tau;
  real<lower=0> a0;
  real<lower=0> b0;
}
parameters {
  real mu;
  real<lower=0> sigma2;
}
transformed parameters {
  real<lower=0> sigma = sqrt(sigma2);
}
model {
  mu ~ normal(mu0, tau);
  sigma2 ~ inv_gamma(a0, b0);
  y ~ cauchy(mu, sigma);
}
"
```

```
fit_cauchy_stan <- stan(
    model_code = stan_cauchy_code,
    data = list(n = length(y_cauchy), y = y_cauchy, mu0 = 0, tau = 10, a0 = 1, b0 = 1),
    chains = 4, iter = 2000, warmup = 1000, seed = 8, refresh = 0
)
```

```
st_c <- as.matrix(fit_cauchy_stan, pars = c("mu", "sigma"))
knitr::kable(data.frame(
    quantity = c("posterior mean of mu", "posterior sd of mu",
                 "posterior mean of sigma", "posterior sd of sigma"),
    gibbs = c(mean(draws_cauchy[keep_c, "mu"]), sd(draws_cauchy[keep_c, "mu"]),
              mean(draws_cauchy[keep_c, "sigma"]), sd(draws_cauchy[keep_c, "sigma"])),
    stan = c(mean(st_c[, "mu"]), sd(st_c[, "mu"]), mean(st_c[, "sigma"]), sd(st_c[, "sigma"])),
    exact = c(grid_mean(mu_grid, 1), grid_sd(mu_grid, 1),
              grid_mean(sg_grid, 2), grid_sd(sg_grid, 2))
), digits = 3, col.names = c("quantity", "Gibbs (Example 4)", "Stan (NUTS)", "exact (grid)"),
   align = c("l", "r", "r", "r"))
```

Table 12.7: Checks Stan's NUTS sampler for the Cauchy model against the data-augmentation Gibbs sampler of Example 4 and the exact grid posterior. Posterior means and standard deviations of mu and sigma, same data and priors. Stan samples the Cauchy likelihood directly, without latent variables; all three agree up to Monte Carlo error.

quantity	Gibbs (Example 4)	Stan (NUTS)	exact (grid)
posterior mean of mu	2.228	2.234	2.228
posterior sd of mu	0.179	0.177	0.179
posterior mean of sigma	0.768	0.764	0.765
posterior sd of sigma	0.181	0.180	0.178

```
summary(fit_cauchy_stan, pars = c("mu", "sigma"))$summary[, c("mean", "sd", "n_eff", "Rhat")]
```

```
           mean        sd    n_eff      Rhat
mu    2.2343723 0.1774277 2268.452 0.9999983
sigma 0.7638365 0.1799397 2618.336 1.0004121
```

The three answers agree, and $\hat{R} \approx 1$ with effective sample sizes in the thousands shows that the four chains have mixed. The comparison shows the trade-off between the two approaches. The Gibbs sampler of Example 4 needs no tuning and each iteration is cheap, but it needed the scale-mixture trick and three derivations. Stan needed only the model statement, and it would work unchanged with a different prior or a Student-t likelihood with any degrees of freedom, where the Gibbs derivation would have to be redone.

12.5 A Final Practical Example: Sparse Logistic Regression

The examples so far were chosen to show how each sampler works, on models small enough to check against an exact answer. This section puts the pieces together on a problem where Bayesian computation earns its place in

practice: a logistic regression with as many unknowns as there are cases, where maximum likelihood breaks down, the prior decides the quality of the answer, and Stan makes the fit routine. The posterior is compared with two standard non-Bayesian fits, `glm` and the LASSO, on new test data.

12.5.1 The problem

Binary responses $y_1, \dots, y_n$ depend on p candidate predictors through the logistic regression model

$$y_i \mid \alpha, \beta \sim \text{Bernoulli}\big(\text{logit}^{-1}(\alpha + x_i^\top \beta)\big), \qquad \text{logit}^{-1}(\eta) = \frac{1}{1+e^{-\eta}}. \tag{12.36}$$

Here there are $p = 50$ predictors for only $n = 100$ cases. The predictors are standard normal and only two carry signal: the true values are $\alpha = 0$, $\beta_1 = 1$, $\beta_2 = -1$ and $\beta_3 = \cdots = \beta_{50} = 0$.

12.5.2 Why maximum likelihood fails

With 51 unknowns and only 100 cases, it is very likely, and true for these data, that some linear combination $\alpha + x_i^\top \beta$ is positive for every case with $y_i = 1$ and negative for every case with $y_i = 0$: the data are perfectly **separable**. Multiplying such a β by a larger and larger constant pushes every fitted probability towards its observed 0 or 1, so the likelihood keeps increasing and the MLE does not exist. `glm` stops after a fixed number of iterations with very large coefficients (here up to about 60 in absolute value) and warns that the algorithm did not converge and that fitted probabilities are numerically 0 or 1.

12.5.3 A sparse prior: the horseshoe

A Bayesian fit always has a proper posterior, but the prior now matters a great deal. A vague prior such as $\beta_j \sim N(0, 2.5^2)$ lets all 48 null coefficients wander, and their sum swamps the signal. We want a prior that says *most coefficients are near 0, but a few may be large*. The **horseshoe prior** does this with two levels of scale:

$$\beta_j = \tau \lambda_j z_j, \qquad z_j \sim N(0,1), \qquad \lambda_j \sim \text{Cauchy}^+(0,1), \qquad \tau \sim \text{Cauchy}^+(0, 0.1), \tag{12.37}$$

where Cauchy$^+$ is the Cauchy distribution restricted to positive values. The **global** scale τ is small, so it shrinks every coefficient strongly towards 0. The **local** scale λ_j has a heavy tail, so a coefficient that the data support can take a large λ_j and escape the shrinkage. The intercept gets $\alpha \sim N(0, 5^2)$. Writing β_j as a product with a standard normal z_j is the non-centred form used for Stan, and a smaller step size (`adapt_delta = 0.995`) helps NUTS handle the very different scales of τ and the λ_j, which otherwise cause many divergent transitions. If a handful remain, Stan reports them as a warning; with only a few out of 8,000 draws, $\hat{R}$ near 1 and large effective sample sizes, they would not noticeably affect the summaries.

12.5.4 Fitting in Stan

The horseshoe model in Stan, and the same model with the vague normal prior for comparison:

```
stan_logit_hs_code <- "
data {
  int<lower=0> n;
  int<lower=1> p;
  matrix[n, p] X;
  array[n] int<lower=0, upper=1> y;
}
parameters {
  real alpha;
  real<lower=0> tau;             // global scale
  vector<lower=0>[p] lambda;     // local scales
  vector[p] z;
}
transformed parameters {
  vector[p] beta = tau * lambda .* z;
}
model {
  alpha ~ normal(0, 5);
  tau ~ cauchy(0, 0.1);
  lambda ~ cauchy(0, 1);
  z ~ std_normal();
  y ~ bernoulli_logit(alpha + X * beta);
}
"
### the same model with a vague normal prior on every coefficient, for comparison
stan_logit_normal_code <- "
data {
  int<lower=0> n;
  int<lower=1> p;
  matrix[n, p] X;
  array[n] int<lower=0, upper=1> y;
}
parameters {
  real alpha;
  vector[p] beta;
}
model {
  alpha ~ normal(0, 5);
  beta ~ normal(0, 2.5);
  y ~ bernoulli_logit(alpha + X * beta);
}
"
```

```
set.seed(2024)
n_lr <- 100; p_lr <- 50
X_lr <- matrix(rnorm(n_lr * p_lr), n_lr, p_lr)
beta_true <- c(1, -1, rep(0, p_lr - 2))
y_lr <- rbinom(n_lr, 1, plogis(drop(X_lr %*% beta_true)))
data_lr <- list(n = n_lr, p = p_lr, X = X_lr, y = y_lr)
fit_logit <- stan(model_code = stan_logit_hs_code, data = data_lr,
                  chains = 4, iter = 3000, warmup = 1000, seed = 9, refresh = 0,
```

```
                  control = list(adapt_delta = 0.995))
fit_logit_normal <- stan(model_code = stan_logit_normal_code, data = data_lr,
                         chains = 4, iter = 2000, warmup = 1000, seed = 9, refresh = 0)
```

12.5.5 The horseshoe posterior

Table 12.8 summarizes the horseshoe posterior of the first ten coefficients: the two signals and eight of the 48 nulls.

```
sm_logit <- summary(fit_logit, pars = sprintf("beta[%d]", 1:10))$summary
knitr::kable(data.frame(
    coefficient = sprintf("beta[%d]", 1:10),
    true = beta_true[1:10],
    mean = sm_logit[, "mean"], sd = sm_logit[, "sd"],
    q025 = sm_logit[, "2.5%"], q975 = sm_logit[, "97.5%"],
    n_eff = round(sm_logit[, "n_eff"]), Rhat = sm_logit[, "Rhat"],
    row.names = NULL),
    digits = c(0, 0, 3, 3, 3, 3, 0, 3),
    col.names = c("coefficient", "true", "mean", "sd", "2.5%", "97.5%", "n_eff", "Rhat"),
    align = c("l", rep("r", 7)))
```

Table 12.8: Summarizes the posterior distribution of the first ten coefficients of a logistic regression with 50 predictors and 100 cases, under the horseshoe prior. Only the first two coefficients carry signal. Posterior mean, standard deviation and 95% credible interval from 8,000 NUTS draws, next to the true value. Rhat near 1 and large effective sample sizes show that the chains have mixed.

coefficient	true	mean	sd	2.5%	97.5%	n_eff	Rhat
beta[1]	1	0.977	0.391	0.043	1.727	2713	1.001
beta[2]	-1	-1.215	0.372	-1.983	-0.521	5868	1.000
beta[3]	0	-0.035	0.123	-0.404	0.124	6675	1.001
beta[4]	0	-0.036	0.119	-0.386	0.118	7601	1.000
beta[5]	0	0.032	0.115	-0.129	0.363	7517	1.000
beta[6]	0	-0.023	0.101	-0.302	0.129	6936	1.000
beta[7]	0	0.002	0.085	-0.173	0.190	8124	1.001
beta[8]	0	-0.037	0.121	-0.404	0.112	7818	1.000
beta[9]	0	-0.012	0.089	-0.248	0.152	7483	1.000
beta[10]	0	-0.009	0.087	-0.227	0.156	7305	1.000

The null coefficients are shrunk tightly around 0, with posterior standard deviations near 0.1. The two signals escape the shrinkage: their posterior means are close to the true values 1 and -1, and both 95% intervals exclude 0. Figure 12.15 shows the posterior draws of the two signals and two of the nulls.

```r
draws_beta <- as.matrix(fit_logit, pars = "beta")
par(mfrow = c(2, 2), mar = c(4, 4, 2.5, 1))
for (j in 1:4) {
    hist(draws_beta[, j], breaks = 60, freq = FALSE, col = "gray80", border = "white",
         xlim = range(c(quantile(draws_beta[, j], c(0.001, 0.999)), 0, beta_true[j])),
         main = bquote(beta[.(j)] ~ "(true value" ~ .(beta_true[j]) * ")"),
         xlab = bquote(beta[.(j)]))
    abline(v = beta_true[j], col = "firebrick", lwd = 2.5)
    abline(v = 0, lty = 2)
}
```

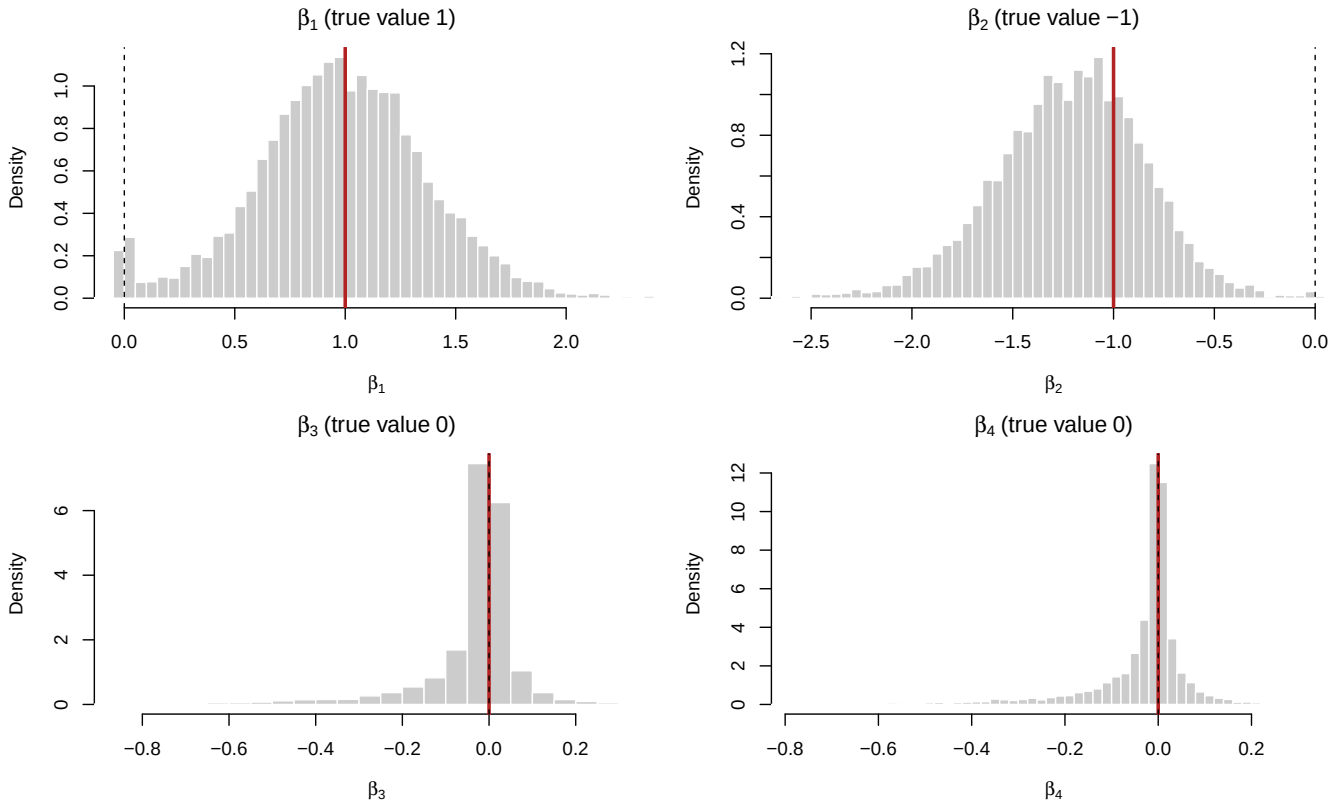


Figure 12.15: Shows the posterior distributions of two signal and two null coefficients in a logistic regression with 50 predictors and 100 cases, under the horseshoe prior. Histograms of 8,000 NUTS draws of beta_1 and beta_2 (signals, top) and beta_3 and beta_4 (nulls, bottom). The red line marks the true value and the dashed line marks 0. The null posteriors are concentrated at 0; the signal posteriors are centred near the true values, with only a small spike at 0 for beta_1.

12.5.6 Prediction on test data

To see how well each method predicts, simulate a test set of $m = 1000$ new cases from the same design, independently of the training data. For a test case with predictors $\tilde{x}$, the **posterior predictive probability** averages the success

probability over the posterior:

$$P(\tilde{y}=1 \mid \tilde{x}, \text{data}) = \int \text{logit}^{-1}(\alpha + \tilde{x}^\top \beta)\, p(\alpha, \beta \mid \text{data})\, d(\alpha, \beta) \approx \frac{1}{S}\sum_{s=1}^{S} \text{logit}^{-1}\big(\alpha^{(s)} + \tilde{x}^\top \beta^{(s)}\big), \tag{12.38}$$

an average over the S posterior draws. No extra package (such as `brms`) is needed: the Stan draws and one matrix product give all m probabilities at once. We compare five sets of predicted probabilities $\hat{p}_i$:

- the posterior predictive probabilities under the horseshoe prior;
- the same under the vague $N(0, 2.5^2)$ prior;
- the plug-in probabilities from the **LASSO**, which maximizes the penalized log-likelihood $\log L(\alpha, \beta) - \lambda \sum_j |\beta_j|$, with the penalty λ chosen by 10-fold cross-validation (`glmnet::cv.glmnet`);
- the plug-in probabilities $\text{logit}^{-1}(\hat{\alpha} + \tilde{x}^\top \hat{\beta})$ from `glm`;
- the true probabilities $\text{logit}^{-1}(\tilde{x}^\top \beta_{\text{true}})$, the best that any method could do.

Each is scored on the test responses in two ways:

- the **test deviance** $-2\sum_{i=1}^{m} \log \hat{P}(\tilde{y}_i)$, where $\hat{P}(\tilde{y}_i) = \hat{p}_i$ if $\tilde{y}_i = 1$ and $1 - \hat{p}_i$ if $\tilde{y}_i = 0$. It rewards well-calibrated probabilities, and smaller is better. Predicting $\hat{p}_i = 0.5$ for every case gives $2m \log 2 \approx 1386.3$, a useful baseline;
- the **error rate** of the classification $\hat{y}_i = 1\{\hat{p}_i > 0.5\}$, summarized together with its **confusion matrix** of predicted against actual classes.

```
set.seed(2025)
m_test <- 1000
X_test <- matrix(rnorm(m_test * p_lr), m_test, p_lr)
y_test <- rbinom(m_test, 1, plogis(drop(X_test %*% beta_true)))

### posterior predictive probabilities: average logit^{-1}(alpha + x beta) over the draws
post_pred <- function(fit) {
    a <- as.matrix(fit, pars = "alpha")[, 1]
    b <- as.matrix(fit, pars = "beta")
    colMeans(plogis(sweep(b %*% t(X_test), 1, a, "+")))      # S x m, then average
}
p_hs <- post_pred(fit_logit)
p_normal <- post_pred(fit_logit_normal)

### maximum likelihood with glm: the data are separable, so glm warns and
### returns very large coefficients; its plug-in probabilities
fit_glm <- suppressWarnings(glm(y ~ ., family = binomial, data = data.frame(y = y_lr, X_lr)))
p_glm <- suppressWarnings(predict(fit_glm, newdata = data.frame(X_test), type = "response"))

### the LASSO: lambda by 10-fold cross-validation (minimum CV deviance)
set.seed(10)
fit_lasso <- glmnet::cv.glmnet(X_lr, y_lr, family = "binomial", alpha = 1, nfolds = 10)
p_lasso <- as.vector(predict(fit_lasso, X_test, s = "lambda.min", type = "response"))
```

```
### the true probabilities, the best any method could do
p_true <- plogis(drop(X_test %*% beta_true))

### scores: test deviance -2 sum log P(y_i), error rate, confusion matrix
test_deviance <- function(p, y) -2 * sum(dbinom(y, 1, p, log = TRUE))
error_rate    <- function(p, y) mean((p > 0.5) != y)
confusion     <- function(p, y) table(predicted = factor(as.integer(p > 0.5), 0:1), actual = y)
methods <- list("Bayes, horseshoe prior" = p_hs, "Bayes, N(0, 2.5^2) prior" = p_normal,
                "LASSO (CV)" = p_lasso, "glm (MLE)" = p_glm, "true probabilities" = p_true)
```

```
conf <- t(sapply(methods, function(p) {
    cm <- confusion(p, y_test)                  # rows: predicted class, columns: actual class
    c(cm["0", "0"], cm["1", "0"], cm["0", "1"], cm["1", "1"])
}))
knitr::kable(data.frame(method = names(methods), conf, row.names = NULL),
             col.names = c("method", "0 as 0", "0 as 1", "1 as 0", "1 as 1"),
             align = c("l", "r", "r", "r", "r"))
```

Table 12.9: Summarizes the classifications of 1,000 test cases by each method in flattened confusion matrices. One row per method. Column 'a as b' counts the test cases of actual class a classified as b (predicted probability above 0.5 for class 1); '0 as 0' and '1 as 1' are correct, the other two are errors.

method	0 as 0	0 as 1	1 as 0	1 as 1
Bayes, horseshoe prior	369	106	180	345
Bayes, N(0, 2.5^2) prior	330	145	222	303
LASSO (CV)	394	81	229	296
glm (MLE)	299	176	225	300
true probabilities	337	138	134	391

```
knitr::kable(data.frame(
    method = names(methods),
    deviance = sapply(methods, test_deviance, y = y_test),
    error = sapply(methods, error_rate, y = y_test), row.names = NULL),
    digits = c(0, 1, 3), col.names = c("method", "test deviance", "error rate"),
    align = c("l", "r", "r"))
```

Table 12.10: Compares the test-set predictions of the Bayesian logistic regressions with those of the LASSO and glm, by test deviance and error rate, when there are as many predictors as cases. m = 1,000 test cases. Test deviance is -2 times the sum of the log predicted probabilities of the observed responses (smaller is better); predicting 0.5 for every case gives 1386.3. The last row uses the true probabilities, the best possible score.

method	test deviance	error rate
Bayes, horseshoe prior	1104.3	0.286
Bayes, N(0, 2.5^2) prior	2643.8	0.367

Table 12.10: Compares the test-set predictions of the Bayesian logistic regressions with those of the LASSO and glm, by test deviance and error rate, when there are as many predictors as cases. m = 1,000 test cases. Test deviance is -2 times the sum of the log predicted probabilities of the observed responses (smaller is better); predicting 0.5 for every case gives 1386.3. The last row uses the true probabilities, the best possible score.

method	test deviance	error rate
LASSO (CV)	1170.8	0.310
glm (MLE)	25472.5	0.401
true probabilities	1062.2	0.272

The comparison is very different from what one sees with few predictors. `glm` predicts badly: its error rate of 0.401 is far above the 0.272 of the true probabilities, and its test deviance of 25472 is about 18 times the 1386.3 of always predicting 0.5. Its fitted probabilities are almost all 0 or 1, so every wrong prediction costs a huge $-2 \log \hat{P}(\tilde{y}_i)$. A Bayesian fit is not automatically better: with the vague normal prior the test deviance, 2644, is also worse than the baseline. The LASSO, with test deviance 1171 and error rate 0.310, beats the baseline but not the horseshoe; the next two subsections show why. The horseshoe prior, which encodes the belief that only a few predictors matter, reduces the test deviance to 1104 and the error rate to 0.286, against 1062 and 0.272 for the true probabilities. When the number of predictors is a large fraction of the number of cases, the prior is not a detail to be made as vague as possible but the part of the model that makes good prediction possible.

12.5.7 What `glm` and the LASSO estimate

The test-set scores are explained by the fitted coefficients. Table 12.11 shows the maximum-likelihood fit of the first ten coefficients, with Wald 95% intervals $\hat{\beta}_j \pm 1.96\,\widehat{\mathrm{SE}}_j$.

```
cf <- summary(fit_glm)$coefficients[sprintf("X%d", 1:10), ]
knitr::kable(data.frame(
    coefficient = sprintf("beta[%d]", 1:10), true = beta_true[1:10],
    estimate = sprintf("%.1f", cf[, "Estimate"]), se = sprintf("%.0fk", cf[, "Std. Error"] /
     ↪  1000),
    ci = sprintf("(%.0fk, %.0fk)", (cf[, "Estimate"] - 1.96 * cf[, "Std. Error"]) / 1000,
                 (cf[, "Estimate"] + 1.96 * cf[, "Std. Error"]) / 1000),
    row.names = NULL),
    col.names = c("coefficient", "true", "estimate", "SE", "95% Wald interval"),
    align = c("l", "r", "r", "r", "r"))
```

Table 12.11: Shows the maximum-likelihood estimates of the first ten coefficients of the logistic regression with 50 predictors and 100 separable cases. Estimates from glm with standard errors and Wald 95% intervals; k = thousands. The algorithm did not converge, and the intervals carry no information.

coefficient	true	estimate	SE	95% Wald interval
beta[1]	1	38.4	111k	(-218k, 218k)
beta[2]	-1	-63.8	70k	(-138k, 138k)

Table 12.11: Shows the maximum-likelihood estimates of the first ten coefficients of the logistic regression with 50 predictors and 100 separable cases. Estimates from glm with standard errors and Wald 95% intervals; k = thousands. The algorithm did not converge, and the intervals carry no information.

coefficient	true	estimate	SE	95% Wald interval
beta[3]	0	22.3	91k	(-179k, 179k)
beta[4]	0	-30.5	89k	(-174k, 174k)
beta[5]	0	6.4	154k	(-301k, 301k)
beta[6]	0	10.4	93k	(-183k, 183k)
beta[7]	0	10.7	79k	(-155k, 155k)
beta[8]	0	-13.1	105k	(-205k, 205k)
beta[9]	0	18.2	169k	(-332k, 332k)
beta[10]	0	-35.0	82k	(-161k, 161k)

`glm` stopped after 25 iterations without converging, because on separable data the likelihood keeps rising as $\|\beta\| \to \infty$. Its estimates are large and arbitrary (the null coefficient β_4 is estimated as -30.5), and the standard errors are of order 10^5: the intervals contain every plausible value and say nothing about which predictors matter.

The LASSO penalty $\lambda \sum_j |\beta_j|$ has a corner at 0, so it sets small coefficients **exactly** to 0. Figure 12.16 shows the 10-fold cross-validated deviance as a function of λ, and Table 12.12 the resulting estimates next to the horseshoe posterior means.

```
par(mar = c(4, 4, 3, 1)); plot(fit_lasso)
```

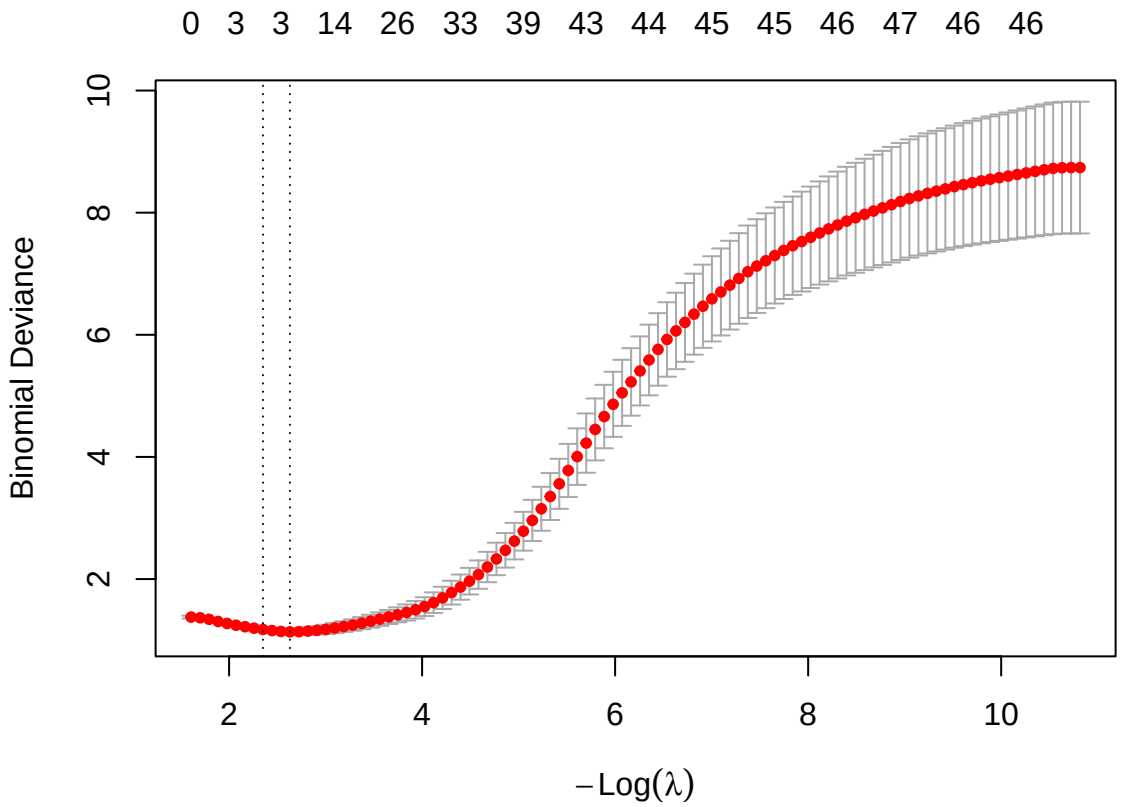


Figure 12.16: Shows how the LASSO penalty is chosen by cross-validation for the logistic regression with 50 predictors. 10-fold cross-validated binomial deviance (with one-standard-error bars) against -log(lambda); the numbers on top are the numbers of predictors kept. The right dashed line marks the lambda with the smallest cross-validated deviance, which is used here; the left one the largest lambda within one standard error of it.

```
b_lasso <- as.vector(coef(fit_lasso, s = "lambda.min"))[-1]
knitr::kable(data.frame(
    coefficient = sprintf("beta[%d]", 1:10), true = beta_true[1:10],
    lasso = b_lasso[1:10],
    hs = summary(fit_logit, pars = sprintf("beta[%d]", 1:10))$summary[, "mean"],
    row.names = NULL),
    digits = c(0, 0, 2, 2), col.names = c("coefficient", "true", "LASSO", "horseshoe mean"),
    align = c("l", "r", "r", "r"))
```

Table 12.12: Compares the LASSO estimates of the first ten coefficients with the horseshoe posterior means. The LASSO penalty is chosen by 10-fold cross-validation; only the first two coefficients carry signal.

coefficient	true	LASSO	horseshoe mean
beta[1]	1	0.47	0.98
beta[2]	-1	-0.60	-1.22
beta[3]	0	0.00	-0.04
beta[4]	0	0.00	-0.04

Table 12.12: Compares the LASSO estimates of the first ten coefficients with the horseshoe posterior means. The LASSO penalty is chosen by 10-fold cross-validation; only the first two coefficients carry signal.

coefficient	true	LASSO	horseshoe mean
beta[5]	0	0.00	0.03
beta[6]	0	0.00	-0.02
beta[7]	0	0.00	0.00
beta[8]	0	0.00	-0.04
beta[9]	0	0.00	-0.01
beta[10]	0	0.00	-0.01

The LASSO keeps 3 of the 50 predictors, including both signals, and sets the nulls among $\beta_3, \ldots, \beta_{10}$ to exactly 0. But the same penalty that removes the nulls also shrinks the signals toward 0 ($1 \to 0.47$ and $-1 \to -0.60$), which is why its predicted probabilities are too cautious and its test deviance trails the horseshoe; it also gives no interval estimates. The horseshoe shrinks the nulls just as hard, but its heavy-tailed local scales λ_j let the signals escape the shrinkage.

12.6 Summary

- A Markov chain converges to a target π from any starting point if its transition T leaves π invariant and is aperiodic and irreducible; **detailed balance**, a pairwise condition, is the standard route to invariance.
- **Gibbs sampling** cycles through exact draws from each full conditional; free of tuning, but needs conditionals that are both known and directly sample-able, and data augmentation extends it to models with latent variables.
- **Metropolis–Hastings** accepts or rejects a proposal from *any* distribution with probability depending on the target-density ratio (times a proposal-symmetry correction); random-walk MH needs a tuned step size, aimed at low autocorrelation rather than high acceptance.
- **Hamiltonian Monte Carlo** adds momentum and follows gradient-driven trajectories, discretized by the leapfrog integrator and corrected by a Metropolis step, giving distant, nearly independent proposals even on strongly correlated targets — at the cost of needing a gradient and two tuning parameters (ϵ, L).
- **Stan** automates gradient computation and adaptive HMC (NUTS) for essentially any model expressible in its declarative syntax, turning what would otherwise be a hand-derivation (as the Cauchy model of Examples 4 and 9 shows) into a direct transcription of the model.
- In the **sparse logistic regression** with as many predictors as cases, maximum likelihood fails on separable data and the LASSO over-shrinks the signals, while the horseshoe posterior, fitted by Stan, comes closest to the true probabilities: when the data are thin, the prior is part of what makes good prediction possible.

This is the last of the sampling-methods chapters in the book: quadrature (Chapter 8) and Laplace's method (Chapter 9) work in low dimension by grid or by a local Gaussian approximation; rejection and importance sampling (Chapters 10–11) draw independently from a fixed proposal; Gibbs, Metropolis–Hastings, HMC, and Stan (this chapter) let a Markov chain do the work instead, and are, in practice, how most modern Bayesian computation is actually done.

A List of Figures, Tables, and Apps

A consolidated index of every numbered figure and table in the book, plus the interactive Shiny apps. Click any entry to jump to it in context.

A.1 List of Shinylive Apps

The book's interactive Shinylive apps are collected in a standalone archive: Shinylive Apps. Each app runs in the browser and carries its R source and the notes that accompany it.

A.2 List of Figures

A.2.1 Introduction to Statistical Computation

- Figure 1.1 — Illustrates the two components of estimator quality, bias and variance, by contrasting the sampling distributions of two estimators of the same parameter.
- Figure 1.2 — Shows how the error rates and the p-value of a hypothesis test all arise from the null and alternative distributions of a single test statistic.
- Figure 1.3 — Previews Bayesian updating, in which the posterior combines the prior with the likelihood, here in a case where the data dominate the prior.
- Figure 1.4 — Demonstrates Monte Carlo simulation as a way of approximating a distribution, using an example whose exact answer is known so that the approximation can be checked.
- Figure 1.5 — A one-dimensional warm-up for numerical optimization, showing what it means to maximize a function before moving to problems that cannot be plotted.
- Figure 1.6 — Introduces numerical quadrature, the deterministic alternative to Monte Carlo for computing integrals, by turning the area under a density into a sum of rectangle areas.

A.2.2 Introduction to R and Quarto

- Figure 2.1 — Demonstrates how to summarize a single numeric variable in R with a histogram, which shows the shape, centre, and spread of the class scores.
- Figure 2.2 — Demonstrates how to compare a numeric variable across groups in R, using boxplots with jittered points so that both the group summaries and the individual scores are visible.
- Figure 2.3 — Demonstrates how to visualize a relationship separately within groups in R, by fitting one regression line per program to a scatterplot.

A.2.3 Computer Arithmetic

- Figure 3.1 — Illustrates how floating-point numbers are spaced on the real line in a toy number system: the gaps between representable numbers grow with magnitude, so precision is relative rather than absolute.

A.2.4 Random Numbers and Monte Carlo Methods

- Figure 4.1 — Checks the quality of a pseudo-random number generator written from scratch by comparing it with R's built-in generator on three diagnostics: behaviour over time, autocorrelation, and uniformity.
- Figure 4.2 — Shows the simplest transformation of uniform random numbers, a change of location and scale.
- Figure 4.3 — Shows geometrically how the inverse-CDF method turns uniform draws into draws from a target distribution.
- Figure 4.4 — Verifies the inverse-CDF method on a distribution whose inverse CDF has a closed form.
- Figure 4.5 — Shows the two facts behind the Box-Muller transform: the angle of a pair of independent standard normals is uniform, and its squared radius is exponential. 3000 pairs (Z_1, Z_2) of iid $N(0,1)$ draws (left), with red circles enclosing 50%, 90% and 99% of the points; the squared radius R^2 follows Exp$(1/2)$ (top right) and the angle Θ is uniform on $(0, 2\pi)$ (bottom right).
- Figure 4.6 — Shows how uniform random numbers can be transformed into normal ones, and verifies the Box-Muller construction against R's rnorm().
- Figure 4.7 — Illustrates the two theorems behind Monte Carlo methods: the Law of Large Numbers, under which averages converge to the mean, and the Central Limit Theorem, under which their fluctuations are approximately normal.
- Figure 4.8 — Shows the Central Limit Theorem as a statement about distributions: the standardized mean of skewed data becomes approximately normal as the sample size grows.
- Figure 4.9 — Illustrates Monte Carlo integration with a geometric example, estimating π as the proportion of random points that fall inside a circle. 1,000 uniform points ('darts') thrown at the $[-1, 1]^2$ square; the fraction landing inside the inscribed circle gives a Monte Carlo estimate of π.
- Figure 4.10 — Shows the Monte Carlo error bound at work along a single simulation run: the running estimate of π stays within a band that narrows at rate $1/\sqrt{n}$.
- Figure 4.11 — Uses simulation to compare two estimators of a binomial proportion by their mean squared error, showing when a small bias pays off through lower variance.
- Figure 4.12 — Shows that the p-value is a tail area of the null distribution of the test statistic, and equivalently the null survival function evaluated at the observed statistic.
- Figure 4.13 — Illustrates the basic simulation check of a test: p-values are uniform under the null hypothesis and concentrate near 0 under the alternative. p-values of the one-sample t-test for $n = 20$ normal observations under $H_0 : \mu = 0$ (left) and $H_1 : \mu = 0.5$ (right); red bars are the rejections at $\alpha = 0.05$.
- Figure 4.14 — Uses simulation to check the size of the t-test: under the null hypothesis a valid test gives uniform p-values, and here the data are nearly normal.
- Figure 4.15 — Repeats the size check of the t-test for heavy-tailed data to show how violating the normality assumption distorts the p-values.
- Figure 4.16 — Pushes the size check to the extreme case of Cauchy data, which have no mean, to show where the t-test breaks down completely.
- Figure 4.17 — Shows that a larger sample size can rescue the t-test for heavy-tailed data, because the Central Limit Theorem makes the sample mean nearly normal.

- Figure 4.18 — Uses simulation to estimate the power of the t-test, its ability to detect a true mean of 0.5, when the data are moderately heavy-tailed. t-test p-values at $\nu = 5$ with true mean $\mu = 0.5$, for $n = 30$ and $n = 60$; the red rejection rate is the power.
- Figure 4.19 — Repeats the power study for heavier-tailed data to show how tail weight affects the ability of the t-test to detect a real effect. t-test p-values at $\nu = 3$ with true mean $\mu = 0.5$, for $n = 30$ and $n = 60$; the red rejection rate is the power.
- Figure 4.20 — Verifies by simulation that the permutation test for correlation has the correct size without any normality assumption, and estimates its power.
- Figure 4.21 — Demonstrates that choosing the best of many predictors using the data and then testing it as if it had been chosen in advance gives an invalid p-value. p-values of `test_lm_sel` under H_0 (left) and $H_1 : \beta = 0.5$ (right); under H_0 they are far from uniform, with a rejection rate far above 0.05.
- Figure 4.22 — Shows why the selection step must be repeated inside the permutation loop: the selected predictor depends on the response, so every permuted response selects its own predictor.
- Figure 4.23 — Shows that a permutation test which repeats the whole selection step on each shuffled data set restores valid p-values while keeping good power.
- Figure 4.24 — Shows how plugging estimated parameters into a textbook null distribution biases a goodness-of-fit test, and how a parametric bootstrap that re-estimates the parameters on every simulated data set removes the bias. p-values of the KS normality test with estimated mean and variance for $n = 50$: the naive test (top) against the Monte Carlo test (bottom), for normal data (H_0), t_3 data and exponential data; red bars are the rejections at $\alpha = 0.05$.

A.2.5 Optimization for Maximum Likelihood Estimation

- Figure 5.1 — Introduces the likelihood and the negative log-likelihood with the simplest possible example, showing that the parameter value making the data most probable is the one that minimizes the function every optimizer in this chapter works with.
- Figure 5.2 — Relates the three central objects of maximum likelihood, the negative log-likelihood, its gradient, and its curvature, and shows how each changes with sample size.
- Figure 5.3 — Shows how the Cauchy location negative log-likelihood, which is harder to minimize than the normal one, concentrates around the true value as the sample size grows.
- Figure 5.4 — Shows the sampling variability of the negative log-likelihood itself: different data sets give different curves, and the minima of those curves, the MLEs, become less variable as n grows. 50 repeated Cauchy negative log-likelihood curves at $n = 20$ versus $n = 200$: the curves cluster more tightly, and their minima concentrate, as n grows.
- Figure 5.5 — Sets up the running example of a zero-truncated Poisson model by showing the negative log-likelihood that every method in this section minimizes, together with its gradient, whose root is the minimizer.
- Figure 5.6 — Illustrates fixed-point iteration, the simplest scheme for solving the stationarity equation, and how its iterates converge to the solution.
- Figure 5.7 — Illustrates bisection, a slow but robust root-finding scheme that only needs an interval known to contain the root.
- Figure 5.8 — Illustrates golden section search, a derivative-free method for optimizing a one-dimensional function, and why it needs only one new function evaluation per step.
- Figure 5.9 — Tests golden section search on a smooth likelihood and on a non-differentiable function to show that the method needs no derivatives.

- Figure 5.10 — Illustrates Newton-Raphson, which replaces the gradient of the negative log-likelihood by its tangent line at each step, and contrasts it with Fisher scoring, which uses the expected rather than the observed curvature.
- Figure 5.11 — Explains why Newton-Raphson can fail on the Cauchy likelihood by showing the shape of the contribution of a single observation.
- Figure 5.12 — Compares the convergence speed of three iterative algorithms on the same problem by showing how quickly each reduces its error.
- Figure 5.13 — Shows how the curvature of the negative log-likelihood at the MLE yields a standard error and a Wald confidence interval, by approximating it with a parabola.
- Figure 5.14 — Uses simulation to show why the MLE is worth computing: ignoring the truncation gives a biased estimator, whereas the MLE is centred on the truth.
- Figure 5.15 — Shows the trial points of one Nelder-Mead iteration, which are all on the line through the worst vertex and the centroid of the other two.
- Figure 5.16 — Shows how the Nelder-Mead simplex follows the curved valley of the Rosenbrock function using only function values.
- Figure 5.17 — Gives a geometric picture of the gradient and the Hessian in two dimensions, explaining why the direction of steepest descent generally does not point at the minimum.
- Figure 5.19 — Shows the simulated logistic-regression data used as a running example for multivariate optimization.
- Figure 5.20 — Shows what a hidden layer buys in a Poisson regression, by displaying the three scaled hidden units and the non-monotone mean curve they combine into, together with counts simulated from it.
- Figure 5.23 — Illustrates the main weakness of gradient descent: on an elongated objective it makes slow zig-zag progress.
- Figure 5.24 — Shows how conjugate gradient repairs the zig-zag behaviour of gradient descent by choosing search directions that do not undo earlier progress.

A.2.6 EM Algorithms

- Figure 6.1 — Illustrates the Q function of the EM algorithm as the expectation of the complete-data log-likelihood over the unobserved data, approximated here by averaging simulated completions.
- Figure 6.2 — Introduces the two-component normal mixture model by showing that its density is the weighted sum of the two component densities.
- Figure 6.3 — Shows that a mixture density can look unimodal or bimodal depending on its parameters, which affects how easily those parameters can be estimated.
- Figure 6.4 — Illustrates the E-step of EM for a mixture: the responsibility of a component is the posterior probability that an observation came from it.
- Figure 6.5 — Shows the simulated data used to demonstrate EM for a mixture; the colours reveal the component labels, which are unobserved in practice and must be inferred.
- Figure 6.6 — Illustrates the conditional distributions needed in the E-step when an observation is only known to lie on one side of a cut point.
- Figure 6.7 — Illustrates the E-step for censored Poisson counts: the missing count follows the Poisson distribution restricted to the values consistent with the censoring information.
- Figure 6.8 — Illustrates how the shape k and the scale λ change the Weibull density.
- Figure 6.9 — Illustrates the conditional distribution of a censored lifetime in the E-step.
- Figure 6.10 — Illustrates the data of Simulation 1 with the Weibull density that generated the lifetimes.

- Figure 6.11 — Illustrates EM for right-censored Weibull data from three starting values: every run climbs the observed log-likelihood to the same maximum.
- Figure 6.12 — Illustrates the four data sets of Simulation 2 ($n = 400$, $k = 2$, $\lambda = 10$), from light to heavy censoring.
- Figure 6.13 — Illustrates one simulated data set ($n = 100$, $k = 2$, $\lambda = 10$) at each of the four censoring levels of Simulation 3.
- Figure 6.14 — Illustrates logistic regression as a thresholded latent variable and the conditional distributions of the latent variable used in the E-step.
- Figure 6.15 — Illustrates that the convergence rate of EM for logistic regression is set by the fraction of missing information, which is larger when the effects are strong.
- Figure 6.16 — Illustrates model-based clustering, the multivariate extension of the normal mixture model, in which EM assigns observations to clusters of different shapes.
- Figure 6.17 — Illustrates a hidden Markov model, whose unobserved state sequence generates the data, as a further example to which EM applies.
- Figure 6.18 — Illustrates the key identity for a single value of the hidden data: the complete log-likelihood and the conditional log-density of the hidden data differ by the observed log-likelihood.
- Figure 6.19 — Illustrates the key identity: the average complete log-likelihood under a fixed distribution of the hidden data differs from the observed log-likelihood by a cross-entropy.
- Figure 6.20 — Illustrates the ascending property: moving from the current estimate to the next one raises Q and also raises the cross-entropy gap, so the observed log-likelihood rises by at least the gain in Q.
- Figure 6.21 — Illustrates EM as alternating maximization of F on the censored Poisson example: the E-step lifts the lower bound until it touches the observed log-likelihood at the current estimate, and the M-step moves to the maximum of that bound.

A.2.7 Introduction to Bayesian Inference

- Figure 7.1 — Shows how Bayes' rule combines a prior with a likelihood into a posterior, and how a credible interval is read off from the posterior.
- Figure 7.2 — Shows how the posterior predictive distribution is built by averaging the sampling density over the posterior, so that it accounts for uncertainty about the parameter.
- Figure 7.3 — Shows the range of prior beliefs about a probability that the Beta family can express, to guide the choice of a prior.
- Figure 7.4 — Shows how the influence of the prior fades as data accumulate, so that the posterior concentrates around the MLE.
- Figure 7.5 — Shows why the posterior predictive distribution is more dispersed than a plug-in prediction: it also reflects the remaining uncertainty about the Poisson rate.
- Figure 7.6 — Shows why the posterior predictive distribution is wider than a plug-in prediction: it also reflects the remaining uncertainty about the mean.

A.2.8 Numerical Quadrature

- Figure 8.1 — Shows how a change of variable maps an integral over the whole real line onto a finite interval, so that a rule with equally spaced points can be applied.
- Figure 8.2 — Shows how the error of each quadrature rule shrinks as the number of panels grows, which reveals the order of the rule.

- Figure 8.3 — Illustrates that the marginal likelihood $P(y)$ is the integral of the likelihood times the prior, and that it changes with the prior even when the likelihood is fixed.
- Figure 8.4 — Illustrates how a two-dimensional posterior integral is computed by transforming it to the unit square and applying a product midpoint grid.
- Figure 8.5 — Illustrates the curse of dimensionality for quadrature: a grid wastes most of its points on negligible regions, and the number of evaluations explodes with the dimension.

A.2.9 Laplace Approximation

- Figure 9.1 — Illustrates the geometric meaning of |H| in the Laplace approximation: the contours of the matching Gaussian are ellipses whose area is proportional to |H|^(-1/2).
- Figure 9.2 — Shows how the Laplace approximation replaces a posterior by a normal density centred at the mode, and how well it fits at small and large sample sizes.
- Figure 9.4 — Shows how the Laplace-adapted product grid is built: equal cells on the unit square are mapped to normal quantiles, then sheared by the Cholesky factor onto the posterior ellipse.
- Figure 9.5 — Shows that the marginal likelihood picks the true number of predictors in a logistic regression, for two prior scales.

A.2.10 Rejection Sampling

- Figure 10.1 — Illustrates the theorem that a point uniform under the graph of g has marginal density proportional to g, and conversely.
- Figure 10.2 — Illustrates the rejection sampling algorithm: candidate points are drawn uniformly under an envelope and kept when they fall under the target density.
- Figure 10.3 — Verifies that the Gamma rejection sampler produces draws from the correct distribution.
- Figure 10.4 — Verifies that adaptive rejection sampling produces draws from the correct truncated normal distribution, even far out in the tail.
- Figure 10.5 — Verifies that sampling Y = log X by adaptive rejection sampling and returning X = exp(Y) gives exact draws from the truncated Gamma distribution, including for shapes below 1.

A.2.11 Importance Sampling

- Figure 11.5 — Demonstrates a silent failure of importance sampling: when the proposal misses a mode of the target, the weights can look well behaved while the estimate is wrong.
- Figure 11.7 — Shows that the marginal likelihood, estimated by importance sampling from the Laplace approximation, picks the true number of predictors in a logistic regression, for two prior scales.
- Figure 11.3 — Compares a naive bounding-box proposal with a proposal matched to the target's support for estimating the area of an irregular region. 1,500 darts thrown at the crescent $A = D_1 \setminus D_2$ (outlined in black) from a generous bounding box (left) and from D_1 itself (right); blue points land in A, red points miss.

A.2.12 Markov Chain Monte Carlo

- Figure 12.2 — Illustrates the long-run behaviour of Markov chains by contrasting two random walks on ten states with different transition rules.
- Figure 12.3 — Shows the structure of the normal model with unknown mean and variance as a directed graph.
- Figure 12.6 — Shows the structure of the interval-censored normal model as a directed graph, with the exact values as latent variables.
- Figure 12.7 — Shows the structure of the augmented Cauchy model as a directed graph.
- Figure 12.4 — Demonstrates the Gibbs sampler on a normal model with unknown mean and standard deviation, using trace plots to check convergence and densities to summarize the posterior.
- Figure 12.8 — Demonstrates the data-augmentation Gibbs sampler for Cauchy data with unknown location and scale, checked against the exact posterior.
- Figure 12.15 — Shows the posterior distributions of two signal and two null coefficients in a logistic regression with 50 predictors and 100 cases, under the horseshoe prior.
- Figure 12.10 — Demonstrates random-walk Metropolis-Hastings on a difficult target, a truncated normal distribution, and shows how the step size affects the result.
- Figure 12.9 — Illustrates the trade-off in choosing the step size of a random-walk Metropolis-Hastings sampler, using trace plots and autocorrelation.
- Figure 12.12 — Illustrates how to tune the step size in practice, by comparing short preliminary runs at several scales.
- Figure 12.14 — Compares Hamiltonian Monte Carlo with random-walk Metropolis on a highly correlated target, where the random walk struggles.

A.3 List of Tables

A.3.1 Introduction to R and Quarto

- Table 2.1 — Shows the kind of tidy data frame used to practise data handling in R, with one row per student and one column per assessment category.

A.3.2 Computer Arithmetic

- Table 3.1 — Compares a naive Taylor-series implementation of the exponential function with the built-in exp() to show that the series is accurate for positive arguments but fails for negative ones.
- Table 3.2 — Exposes the internals of the failing computation of exp(-50), showing that very large terms of alternating sign cancel and leave only roundoff error.
- Table 3.3 — Shows that reformulating the computation as 1/exp(|x|) removes the cancellation and restores accuracy for negative arguments.
- Table 3.4 — Demonstrates catastrophic cancellation in the one-pass variance formula, which breaks down when the data have a large mean, while the two-pass formula stays accurate.

A.3.3 Random Numbers and Monte Carlo Methods

- Table 4.1 — Shows how the accuracy of the Monte Carlo estimate of π, as measured by its margin of error, improves with the number of points.
- Table 4.2 — Compares three estimators of the centre of a symmetric distribution under normal and heavy-tailed data, to show that the best estimator depends on the data distribution.

A.3.4 EM Algorithms

- Table 6.4 — Shows that the error ratio of the closed-form EM for known shape is close to the censored fraction, so heavier censoring needs more iterations.
- Table 6.5 — Shows what ignoring the censoring costs: EM stays near the truth while the two naive analyses are biased.
- Table 6.6 — Shows that the EM algorithm built on the latent logistic variable reaches the same estimates and log-likelihood as `glm`, but needs many more iterations, especially when the effects are strong.

A.3.5 Introduction to Bayesian Inference

- Table 7.1 — Collects every quantity reported in a conjugate Beta-Binomial analysis, from the prior and MLE to the posterior, credible interval, and predictive distribution.
- Table 7.2 — Collects every quantity reported in a conjugate Poisson-Gamma analysis, from the prior and MLE to the posterior, credible interval, and predictive distribution.
- Table 7.3 — Collects every quantity reported in a conjugate Normal-Normal analysis, from the prior and MLE to the posterior, credible interval, and predictive distribution.

A.3.6 Numerical Quadrature

- Table 8.2 — Compares the accuracy of the midpoint, trapezoidal, and Simpson rules on the same integral with the same small number of panels.
- Table 8.3 — Compares quadrature with naive Monte Carlo for computing the log marginal likelihood, to show that a deterministic grid is much more accurate in low dimensions.
- Table 8.4 — Shows how the quadrature estimate converges as the grid is refined, which indicates how many points are needed.
- Table 8.5 — Shows how the log marginal likelihood, the quantity used for Bayesian model comparison, changes with the prior mean and prior standard deviation.
- Table 8.6 — Shows how the marginal likelihood can be used to compare models, here t models with different degrees of freedom fitted to Gaussian and to heavy-tailed data.
- Table 8.7 — Shows a failure mode of grid quadrature and naive Monte Carlo: when the posterior mass lies far from where the grid or the prior draws are placed, both estimates are poor.

A.3.7 Laplace Approximation

- Table 9.1 — Assesses the accuracy of the Laplace approximation by comparing it with the exact marginal likelihood in a conjugate example where the exact answer is known.
- Table 9.2 — Compares interval estimates based on the Laplace approximation with the exact posterior intervals, to see how much accuracy is lost.
- Table 9.3 — Compares the Laplace approximation with grid quadrature and naive Monte Carlo against the true value as the posterior moves away from the prior mean, to show where each method succeeds or fails.

A.3.8 Rejection Sampling

- Table 10.1 — Verifies the rejection sampler for the normal distribution by comparing its acceptance rate and sample moments with theory.
- Table 10.2 — Checks that a proposed envelope really lies above the target density, which rejection sampling requires, for several parameter values.
- Table 10.3 — Shows how the efficiency of the Gamma rejection sampler, measured by its acceptance rate, changes with the shape parameter.
- Table 10.4 — Compares the cost of adaptive rejection sampling with that of naive rejection sampling for truncated normal distributions.

A.3.9 Importance Sampling

- Table 11.2 — Shows how the choice of bounding constants affects importance sampling of a Student-t interval, including a case where the weights have infinite variance.
- Table 11.3 — Compares importance sampling proposals for estimating a marginal likelihood, showing that a Laplace-shaped proposal is more reliable than the prior as n grows.
- Table 11.1 — Shows that both proposals are unbiased for the true area, and that only the effective sample size distinguishes them: the disk proposal wastes draws only on D_2, while the box proposal also wastes draws outside D_1 entirely.

A.3.10 Markov Chain Monte Carlo

- Table 12.2 — Summarizes the posterior distribution obtained from the Gibbs sampler.
- Table 12.3 — Shows that Gibbs sampling extends to interval-censored data by treating the exact values as latent variables, and that the parameters are still recovered.
- Table 12.4 — Checks the data-augmentation Gibbs sampler for the Cauchy model against the exact posterior computed on a grid.
- Table 12.5 — Checks the Metropolis-Hastings sampler against Gibbs sampling on the same problem.
- Table 12.6 — Validates Stan's sampler against the samplers written by hand earlier in the chapter.
- Table 12.7 — Checks Stan's NUTS sampler for the Cauchy model against the data-augmentation Gibbs sampler and the exact grid posterior.
- Table 12.8 — Summarizes the posterior distribution of the first ten coefficients of a logistic regression with 50 predictors and 100 cases, under the horseshoe prior.
- Table 12.9 — Summarizes the classifications of 1,000 test cases by each method as confusion matrices.

- Table 12.10 — Compares the test-set predictions of the Bayesian logistic regressions with those of glm, by test deviance and error rate, when there are as many predictors as cases.