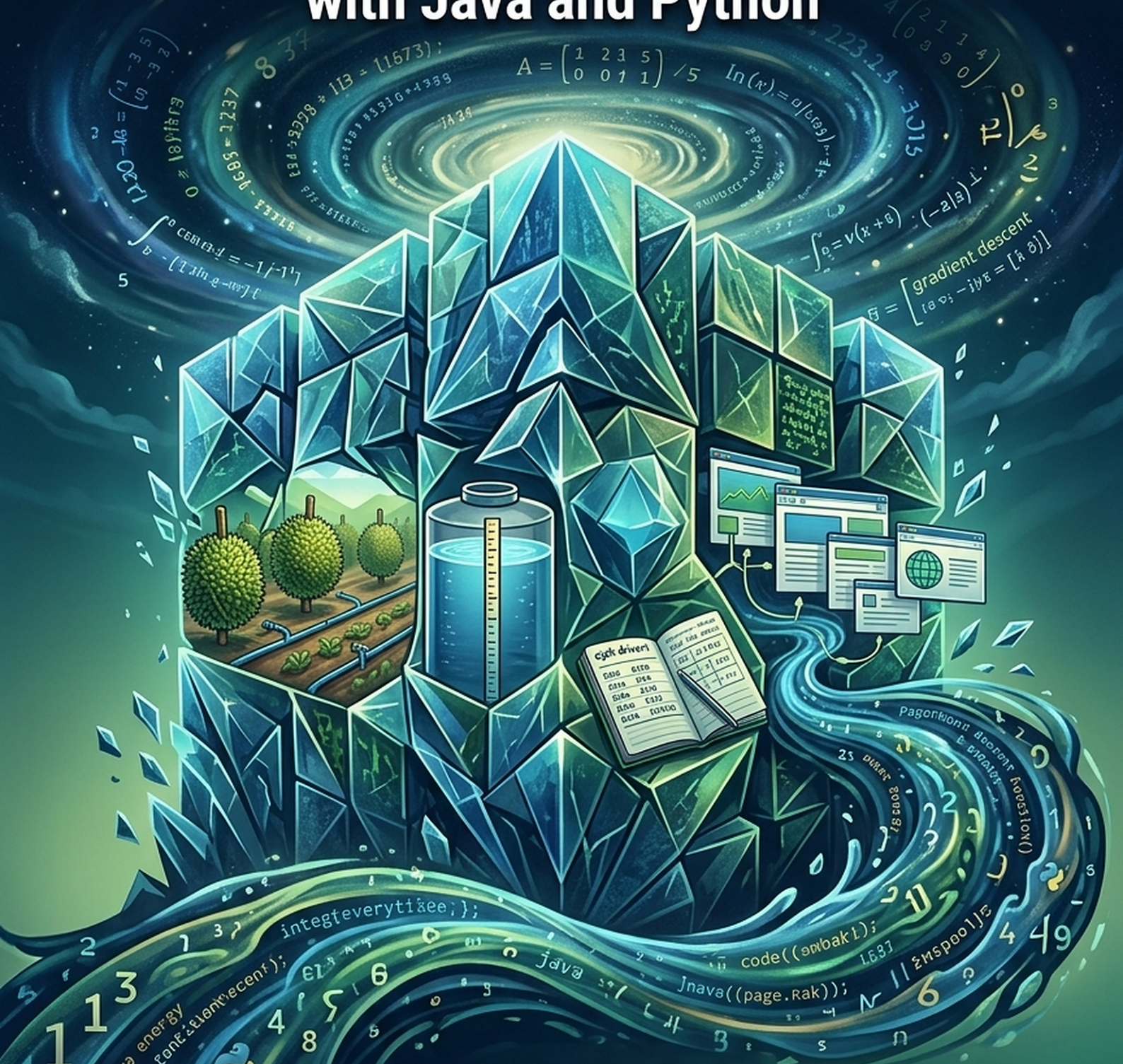
Numerical Methods

From Error Analysis to Machine Learning, with Java and Python



Irfan Subakti

Profitina
profitina.com

Numerical Methods

From Error Analysis to Machine Learning, with Java and Python

Irfan Subakti



Copyright © 2026 Irfan Subakti. All rights reserved.

Published by Profitina · profitina.com

First edition, September 2026.

Every program in this book was compiled and run before its output was printed. The complete source code is in the accompanying Code folder. The programs are offered for teaching and learning; Irfan Subakti and the publisher accept no liability for their use elsewhere.

Contents

Preface ..... **xxiv**
Who this book is for xxiv
One idea, two languages xxiv
What is in it xxv
How to read it xxvi
About the author xxvi
Acknowledgments xxvi

CHAPTER 1 Numbers in the Machine: Errors, Floating Point, and Your Workbench **1**
1.1 A Receipt That Does Not Add Up 1
1.2 What Numerical Methods Are 2
1.3 Your Workbench: What We Install and Why 3
1.4 Installing on Windows 10 and 11 4
1.4.1 The JDK 4
1.4.2 Python 4
1.4.3 VS Code 5
1.5 Installing on macOS 5
1.5.1 The JDK 5
1.5.2 Python 5
1.5.3 VS Code 5
1.6 Installing on Linux 6
1.7 VS Code, a Virtual Environment, NumPy and SciPy 6
1.8 Your First Programs 8
1.8.1 Hello, in Java 8
1.8.2 Hello, in Python 8
1.8.3 Checking the whole workbench 9
1.9 Measuring Error 12
1.9.1 True error and relative error 12
1.9.2 Approximate error and the stopping rule 12
1.9.3 Two kinds of error 13
1.10 How the Machine Stores a Number 13
1.10.1 Binary fractions 13
1.10.2 The IEEE 754 double 13
1.10.3 Machine epsilon and the rounding bound 14
1.11 Five Experiments with FloatLab 15
1.12 Round-off and Truncation Together 19
1.12.1 Taylor series and truncation 19
1.12.2 The step-size dilemma 22
1.13 The Laundry Ledger: Adding Up Money Safely 24
1.14 Comparing with the Libraries 27
1.15 Numerical Methods in Practice: Profitina 29
1.16 Using It for Real 29
1.17 Common Mistakes, and How to Avoid Them 30
1.18 Chapter Summary 30
1.19 Exercises 31
1.20 Solutions to the Exercises 31
References 33

CHAPTER 2 Root Finding I: Bisection and Regula Falsi **35**
2.1 Mas Dimas and the “1.67% a Month” Phone 35

2.2 The Idea in Plain Words 36
2.2.1 Money today and money later 36
2.2.2 Turning the question into a root 36
2.2.3 A table of guesses 36
2.3 Bisection 37
2.3.1 Why a sign change guarantees a root 37
2.3.2 The method 37
2.3.3 How many steps: the error bound 38
2.3.4 Finding a bracket in the first place 39
2.3.5 Small details that matter 39
2.4 Regula Falsi 40
2.4.1 The chord 40
2.4.2 Why regula falsi can crawl 41
2.4.3 The Illinois modification 42
2.5 Building It in Java and Python 42
2.6 Running It 47
2.6.1 What 2.92% a month means in rupiah 49
2.7 Where Regula Falsi Crawls 50
2.8 Testing It 51
2.9 Comparing with SciPy 53
2.10 Numerical Methods in Practice: Profitina 54
2.11 Using It for Real 55
2.12 Common Mistakes, and How to Avoid Them 55
2.13 Chapter Summary 56
2.14 Exercises 56
2.15 Solutions to the Exercises 57
References 59

CHAPTER 3 Root Finding II: Newton-Raphson, Secant and Fixed-Point Iteration 60
3.1 Pak Harun's Water Truck 60
3.2 The Idea in Plain Words 61
3.2.1 The volume of water in a lying tank 61
3.2.2 Follow the tangent 62
3.3 Newton-Raphson 62
3.3.1 Derivation 62
3.3.2 Why it is so fast: quadratic convergence 63
3.3.3 When Newton fails 64
3.4 The Secant Method 65
3.5 Fixed-Point Iteration 65
3.5.1 Rewriting $f(x) = 0$ as $x = g(x)$ 65
3.5.2 When and how fast it converges 66
3.6 Building It in Java and Python 67
3.7 Running It 72
3.8 Measuring the Order of Convergence 72
3.9 When Newton Goes Wrong 76
3.10 Testing It 79
3.11 Comparing with SciPy 81
3.12 Choosing a Method 82
3.13 Numerical Methods in Practice: Profitina 83
3.14 Using It for Real 83
3.15 Common Mistakes, and How to Avoid Them 84
3.16 Chapter Summary 84
3.17 Exercises 85
3.18 Solutions to the Exercises 85

References 88

CHAPTER 4 Linear Systems I: Gaussian Elimination and Pivoting 89

4.1 Bu Rahma's Durian Trees 89
4.2 The Idea in Plain Words 90
4.2.1 Three receipts, three prices 90
4.2.2 Elimination by hand 90
4.3 Gaussian Elimination 91
4.3.1 Forward elimination 91
4.3.2 Back substitution 92
4.3.3 Counting the work 92
4.3.4 When there is no unique solution 94
4.3.5 Many right-hand sides 94
4.4 Pivoting 94
4.4.1 A zero pivot 94
4.4.2 A tiny pivot 95
4.4.3 Partial pivoting 95
4.5 Building It in Java and Python 95
4.6 Running It: The Receipts 100
4.7 The Durian Orchard's Fertilizer 102
4.8 Conditioning: A Small Residual Is Not a Small Error 105
4.8.1 Two lines that are almost parallel 105
4.8.2 The condition number 106
4.8.3 Residual and error 107
4.9 Why Pivoting Matters, in Code 109
4.10 Testing It 110
4.11 Comparing with NumPy 112
4.12 Numerical Methods in Practice: Profitina 113
4.13 Using It for Real 114
4.14 Common Mistakes, and How to Avoid Them 114
4.15 Chapter Summary 115
4.16 Exercises 115
4.17 Solutions to the Exercises 116
References 117

CHAPTER 5 Linear Systems II: Jacobi, Gauss-Seidel and SOR 118

5.1 Pak Joni's Griddle 118
5.2 The Idea in Plain Words 119
5.2.1 Each point is the average of its neighbors 119
5.2.2 Four points by hand 119
5.2.3 Jacobi: everybody updates from the old picture 120
5.2.4 Gauss-Seidel: use the newest numbers at once 121
5.2.5 SOR: go a little further than you are told 121
5.3 The Methods Precisely 122
5.3.1 A general linear system 122
5.3.2 Matrix splittings 123
5.4 When and How Fast They Converge 123
5.4.1 The error obeys the same iteration 123
5.4.2 A test you can do by eye: diagonal dominance 124
5.4.3 The model problem: exact spectral radii 125
5.4.4 How many sweeps? 125
5.4.5 When to stop, and what a small change really means 126
5.4.6 Why bother: the cost of a sweep 126
5.5 Building It in Java and Python 126
5.5.1 The small system 127

5.5.2 The griddle without a matrix ... 133
5.6 Running It ... 139
5.7 Checking Against the Library ... 144
5.8 Making It Better ... 145
5.9 Using It for Real ... 145
5.10 Numerical Methods in Practice: Profitina ... 146
5.11 Common Mistakes, and How to Avoid Them ... 146
5.12 Chapter Summary ... 147
5.13 Exercises ... 147
5.14 Solutions to the Exercises ... 148
References ... 149

CHAPTER 6 Matrix Decompositions: LU, Cholesky and QR ... 151
6.1 Pak Harun's Fertilizer Question ... 151
6.2 The Idea in Plain Words ... 152
6.2.1 The equations ... 152
6.2.2 Elimination remembers: finding L and U by hand ... 152
6.2.3 Two triangles per block ... 153
6.3 LU Decomposition ... 154
6.3.1 Where the factors come from ... 154
6.3.2 What it costs ... 155
6.3.3 Determinant and inverse for free ... 156
6.3.4 How accurate is it? ... 156
6.3.5 What goes wrong without pivoting ... 156
6.4 Cholesky: Half the Work for Symmetric Positive Definite Matrices ... 157
6.4.1 A fourth nutrient ... 157
6.4.2 The normal equations ... 157
6.4.3 Symmetric positive definite matrices ... 158
6.4.4 Why Cholesky needs no pivoting ... 159
6.5 QR: Least Squares Without Squaring ... 159
6.5.1 What the normal equations cost in accuracy ... 159
6.5.2 Orthogonal matrices do not stretch ... 159
6.5.3 Householder reflections ... 160
6.5.4 A check that connects the two methods ... 160
6.5.5 Gram-Schmidt, the other road to QR ... 160
6.6 Building It in Java and Python ... 161
6.6.1 LU in Python and Java ... 161
6.6.2 Cholesky and QR in Python and Java ... 167
6.7 Running It ... 175
6.8 Checking Against the Library ... 177
6.9 Making It Better ... 179
6.10 Using It for Real ... 179
6.11 Numerical Methods in Practice: Profitina ... 180
6.12 Common Mistakes, and How to Avoid Them ... 180
6.13 Chapter Summary ... 181
6.14 Exercises ... 181
6.15 Solutions to the Exercises ... 182
References ... 183

CHAPTER 7 Interpolation: Lagrange, Newton and Splines ... 184
7.1 Pak Darto's Dipstick ... 184
7.2 The Idea in Plain Words ... 185
7.2.1 A straight line between two marks ... 185
7.2.2 A curve through more marks ... 185
7.3 Polynomial Interpolation ... 186

7.3.1 One polynomial, and only one 186
7.3.2 The Lagrange form 186
7.3.3 The Newton form and divided differences 187
7.3.4 Evaluating cheaply: nested multiplication 188
7.3.5 How wrong is an interpolating polynomial? 188
7.3.6 Runge's phenomenon and Chebyshev points 188
7.3.7 When the slopes are known too: Hermite interpolation 189
7.4 Cubic Splines 190
7.4.1 Piecewise, but smooth 190
7.4.2 Deriving the tridiagonal system 190
7.4.3 The two end conditions 191
7.4.4 The Thomas algorithm 191
7.4.5 How accurate is a spline? 191
7.4.6 One spline reading by hand 191
7.4.7 Reading the table backward 192
7.5 Building It in Java and Python 192
7.6 Running It 205
7.7 Checking Against the Library 208
7.8 Making It Better 209
7.9 Using It for Real 210
7.10 Numerical Methods in Practice: Profitina 210
7.11 Common Mistakes, and How to Avoid Them 211
7.12 Chapter Summary 211
7.13 Exercises 212
7.14 Solutions to the Exercises 212
References 213

CHAPTER 8 Curve Fitting: Least Squares Regression 215

8.1 Mas Dimas's Notebook 215
8.2 The Idea in Plain Words 216
8.2.1 Misses, and why we square them 216
8.2.2 Five days by hand 217
8.3 The Straight Line Precisely 217
8.3.1 The normal equations for a line 217
8.3.2 Two facts about residuals 218
8.3.3 How good is the fit? $R2$ and the standard error 218
8.3.4 How sure is the slope? 218
8.3.5 Predicting one new day 219
8.3.6 Looking at the residuals 219
8.4 More Predictors: the Design Matrix 219
8.4.1 Fuel from distance and trips 219
8.4.2 Why QR, even here 220
8.4.3 What the coefficients mean 220
8.5 Curves That Are Linear in Their Coefficients 220
8.5.1 Polynomials 220
8.5.2 The best speed 221
8.5.3 Exponentials through logarithms 221
8.6 Overfitting, and How to Choose a Model 222
8.7 Building It in Java and Python 223
8.8 Running It 233
8.9 Checking Against the Library 235
8.10 Making It Better 236
8.11 Using It for Real 237
8.12 Numerical Methods in Practice: Profitina 237
8.13 Common Mistakes, and How to Avoid Them 238

8.14 Chapter Summary 238
8.15 Exercises 239
8.16 Solutions to the Exercises 240
References 241

CHAPTER 9 Numerical Differentiation 242
9.1 Mas Dimas's Braking Report 242
9.2 The Idea in Plain Words 243
9.2.1 Speed from two readings 243
9.2.2 Acceleration from three readings 243
9.3 Deriving the Formulas from Taylor's Theorem 244
9.3.1 Taylor's theorem 244
9.3.2 Forward and backward differences 244
9.3.3 The central difference 245
9.3.4 The second derivative 246
9.3.5 One-sided formulas for the ends 246
9.3.6 The formulas side by side 247
9.4 Two Kinds of Error: Truncation and Round-off 247
9.5 Richardson Extrapolation 251
9.6 Building the Trip Analyzer 254
9.7 Making It Better: Smooth Before You Differentiate 259
9.7.1 Fit a parabola, then differentiate the parabola 259
9.7.2 The smoothed analyzer 259
9.8 Checking Against the Library, and Testing 263
9.9 Numerical Methods in Practice: Profitina 264
9.10 Using It for Real 265
9.11 Common Mistakes, and How to Avoid Them 266
9.12 Chapter Summary 266
9.13 Exercises 267
9.14 Solutions to the Exercises 267
References 269

CHAPTER 10 Numerical Integration: Trapezoid, Simpson, Romberg and Gauss 271
10.1 How Big Is the Orchard? 271
10.2 The Idea in Plain Words 272
10.3 The Trapezoidal Rule 273
10.3.1 Derivation 273
10.3.2 The composite rule 273
10.4 Simpson's Rules 275
10.4.1 Simpson's 1/3 rule 275
10.4.2 Simpson's 3/8 rule 276
10.5 Building the Orchard Calculator 276
10.6 Checking the Order 280
10.7 Romberg Integration 283
10.8 Gauss Quadrature: Choosing Where to Measure 286
10.8.1 The idea 286
10.8.2 Legendre polynomials and the general rule 286
10.8.3 The program 287
10.9 Making It Better: Adaptive Simpson 291
10.10 Checking Against the Library 293
10.11 Numerical Methods in Practice: Profitina 294
10.12 Using It for Real 295
10.13 Common Mistakes, and How to Avoid Them 296
10.14 Chapter Summary 296

10.15 Exercises ... 296
10.16 Solutions to the Exercises ... 297
References ... 298

CHAPTER 11 Ordinary Differential Equations: Euler to Runge-Kutta ... 299
11.1 Will the Tank Last Until Ten? ... 299
11.2 The Idea in Plain Words ... 300
11.3 The Model and Its Exact Solution ... 300
11.3.1 The mass balance ... 300
11.3.2 An exact solution to check against ... 301
11.4 Euler's Method ... 301
11.4.1 Derivation ... 301
11.4.2 Local and global error ... 302
11.4.3 The first program ... 302
11.5 Runge–Kutta Methods ... 305
11.5.1 Second order: Heun and midpoint ... 305
11.5.2 The classical fourth-order method ... 306
11.5.3 Measuring the orders ... 307
11.6 Building the Morning Simulator ... 310
11.7 Making It Better: Planning the Pump ... 316
11.8 Systems and Stability: The Motorbike Seat ... 318
11.8.1 From one equation to a system ... 318
11.8.2 Why Euler blew up: stability ... 322
11.9 Checking Against the Library ... 323
11.10 Numerical Methods in Practice: Profitina ... 324
11.11 Using It for Real ... 324
11.12 Common Mistakes, and How to Avoid Them ... 325
11.13 Chapter Summary ... 325
11.14 Exercises ... 326
11.15 Solutions to the Exercises ... 326
References ... 328

CHAPTER 12 Eigenvalues: Power Method and Its Uses (e.g. PageRank) ... 330
12.1 Three Laundries in One Kampung ... 330
12.2 The Idea in Plain Words ... 331
12.2.1 Directions that a matrix only stretches ... 331
12.2.2 Why repeated multiplication finds the largest one ... 331
12.3 The Laundry Market as a Markov Chain ... 332
12.3.1 The switching matrix ... 332
12.3.2 What the eigenvalues say ... 332
12.3.3 The program ... 332
12.3.4 A fourth state: washing at home ... 336
12.4 The Power Method in General ... 336
12.4.1 The algorithm and the Rayleigh quotient ... 336
12.4.2 Inverse and shifted inverse iteration ... 337
12.4.3 Inverse iteration by hand ... 337
12.4.4 When the power method fails ... 337
12.5 How a Shophouse Sways ... 338
12.5.1 The model ... 338
12.5.2 Three methods, three modes ... 338
12.5.3 A heavier roof ... 344
12.5.4 Why the frequencies matter: resonance ... 345
12.6 PageRank: The Eigenvector of the Web ... 345
12.6.1 The random reader ... 345
12.6.2 A village-tourism network ... 346

12.7 Checking Against the Library 350
12.8 Numerical Methods in Practice: Profitina 351
12.9 Using It for Real 352
12.10 Common Mistakes, and How to Avoid Them 353
12.11 Chapter Summary 353
12.12 Exercises 354
12.13 Solutions to the Exercises 354
References 356

CHAPTER 13 Optimization: Golden Section, Newton and Gradient Descent 357
13.1 Mbak Rina's Price 357
13.2 The Idea in Plain Words 358
13.2.1 From finding zeros to finding peaks 358
13.2.2 Squeezing the peak between two probes 358
13.2.3 Following the slope: Newton and gradient descent 359
13.3 The Methods, Derived 359
13.3.1 Golden-section search 359
13.3.2 Newton's method for a maximum 360
13.3.3 Several variables: gradient and Hessian 361
13.3.4 Gradient descent and its step size 362
13.3.5 Backtracking: letting the method choose the step 363
13.3.6 Newton's method in several variables 364
13.4 The Programs 364
13.4.1 Golden section for one price 364
13.4.2 Newton's method with and without a safeguard 367
13.4.3 Two prices: descent and Newton 369
13.5 Running the Programs 375
13.6 Comparing with SciPy 378
13.7 Making It Better 379
13.8 Using It for Real 380
13.9 Numerical Methods in Practice: Profitina 380
13.10 Common Mistakes, and How to Avoid Them 381
13.11 Chapter Summary 381
13.12 Exercises 382
13.13 Solutions to the Exercises 382
References 383

CHAPTER 14 Numerical Methods for Machine Learning 385
14.1 Mas Dimas's Two Questions 385
14.2 The Idea in Plain Words 386
14.3 The Methods, Derived 387
14.3.1 Least squares: normal equations or QR 387
14.3.2 Gradient descent and the scale of the features 387
14.3.3 Mini-batches, momentum and Adam 388
14.3.4 Logistic regression and Newton's method 389
14.3.5 Formulas that do not overflow 390
14.3.6 Checking gradients, and the width of a float 391
14.4 The Programs 391
14.4.1 The building blocks 391
14.4.2 Fitting the trip time 398
14.4.3 Training by gradient methods 400
14.4.4 Predicting cancellations 405
14.5 Running the Programs 411
14.6 Comparing with NumPy and SciPy 414
14.7 Making It Better 416

14.8 Using It for Real 416
14.9 Numerical Methods in Practice: Profitina 417
14.10 Common Mistakes, and How to Avoid Them 417
14.11 Chapter Summary 418
14.12 Exercises 418
14.13 Solutions to the Exercises 418
References 419

CHAPTER 15 Fast and Parallel Numerical Computing 421
15.1 An Hour for One Picture 421
15.2 The Idea in Plain Words 422
15.2.1 Where the time goes 422
15.2.2 Doing several things at once 422
15.2.3 The same sum, a different answer 422
15.3 The Methods, Derived 423
15.3.1 A cost model: operations and memory 423
15.3.2 Vectorization 423
15.3.3 Parallel speedup and its limits 424
15.3.4 Doing less work: the number of sweeps 424
15.3.5 Accurate and reproducible sums 425
15.4 The Programs 426
15.4.1 Summation 426
15.4.2 The griddle with loops 429
15.4.3 Vectorized, and split over threads 430
15.4.4 Fewer sweeps: red-black SOR 433
15.5 Running the Programs 436
15.6 Comparing with NumPy and SciPy 438
15.7 Making It Better 439
15.8 Using It for Real 440
15.9 Numerical Methods in Practice: Profitina 440
15.10 Common Mistakes, and How to Avoid Them 441
15.11 Chapter Summary 441
15.12 Exercises 442
15.13 Solutions to the Exercises 442
References 443

CHAPTER 16 Capstone: Simulating a Real System End to End 445
16.1 Pak Hadi's Dry Season 445
16.2 The Idea in Plain Words 446
16.3 The Model, Part by Part 447
16.3.1 The tank: a table and two splines 447
16.3.2 The pump: a fitted curve and an operating point 448
16.3.3 Weather: a Markov chain and a quantile table 448
16.3.4 The trees and the well through the season 449
16.3.5 The drip line: a small differential equation 449
16.3.6 One day, and a season 450
16.3.7 Many seasons, reproducibly 450
16.3.8 The decision and a formula behind it 450
16.4 The Programs 451
16.4.1 The toolbox 451
16.4.2 The orchard model 463
16.4.3 Checking the parts 471
16.4.4 Checking the drip model against the orchard 473
16.4.5 Many seasons in parallel 475
16.4.6 Choosing the tank 478

16.5 Running the Simulation 480
16.5.1 The parts 481
16.5.2 The drip model against the sensor 481
16.5.3 Four hundred seasons 482
16.5.4 The best tank 483
16.6 Comparing with SciPy 484
16.7 Making It Better 485
16.8 Using It for Real 486
16.9 Numerical Methods in Practice: Profitina 486
16.10 Common Mistakes, and How to Avoid Them 487
16.11 Chapter Summary 487
16.12 Exercises 488
16.13 Solutions to the Exercises 488
References 489

List of Figures

Figure 1.1 How a Java program and a Python program run. Java is compiled once by `javac` into bytecode, which the Java Virtual Machine runs; Python is read and run by the interpreter every time. 4

Figure 1.2 The parts of Visual Studio Code used in this book: (1) Activity Bar, (2) Explorer, (3) editor, (4) Run button, (5) integrated terminal, (6) Status Bar. 7

Figure 1.3 The 64 bits of the double nearest to 0.1: sign 0, exponent 01111111011 (1019, meaning $2 - 4$) and 52 fraction bits in which the pattern 1001 repeats until it is cut off and rounded. 14

Figure 1.4 The error of the forward difference of x at $x = 2$ against the step h: truncation error falls with h, round-off error grows as $1/h$, and the total is smallest near $h = 10 - 8$ 24

Figure 2.1 The loan function $f(r)$ of Equation (2.3) and the bracket after each bisection step, starting from $[1\%, 5\%]$: every step keeps the half that contains the sign change (amber dot). 38

Figure 2.2 The error $ck - r*$ of the three bracketing methods on the loan function, continued past the tolerance until the error reaches the limit of double precision. 49

Figure 3.1 The water truck's tank seen from its end (left): the water fills a circular segment of depth h and surface width w. The volume Equation (3.1) as a function of the depth (right) is S-shaped, steepest at half depth. 61

Figure 3.2 Newton's method for the 1,000-litre mark: from the box guess $x0$ the tangent overshoots slightly to $x1$, and the second tangent lands almost exactly on the root $x2$. 63

Figure 3.3 The error of four methods computing 2: bisection and fixed-point iteration fall along straight lines (linear convergence), the secant and Newton curves bend downward ever more steeply (orders 1.618 and 2). A line stops where the iterate equals the double nearest to 2. 76

Figure 4.1 Gaussian elimination on four equations: at each stage the pivot (amber) clears its column below it and only the shaded entries change; back substitution then turns the right-hand column into the solution. 91

Figure 4.2 A well-conditioned system (left) and an ill-conditioned one (right): when the lines are almost parallel, a change of 0.01 in the data moves the solution from (1,1) to (0,2) 106

Figure 5.1 The coarsest griddle: four unknown inner temperatures, each the mean of its four neighbors; two of the neighbors are always edges with known temperatures. 120

Figure 5.2 Largest error of the 20 × 20 griddle after each sweep, against a reference solution. Straight lines on this scale mean a constant factor per sweep: $\rho J = 0.989$, $\rho GS = 0.978$, and about 0.74 for SOR with $\omega = 1.741$. 142

Figure 5.3 Sweeps needed by SOR on the 20 × 20 griddle for relaxation factors from 1 to 1.95. The minimum sits at Young's value ωopt = 1.741. 143

Figure 5.4 Steady temperature of the griddle on a 40 × 40 grid (burner on the left, warm front edge at the bottom), with the 140 °C and 170 °C lines that bound the martabak zone. 143

Figure 6.1 Factor once, solve many: the grade matrix is factored as $PA = LU$ once, and each block's recommendation costs only a forward and a back substitution. 154

Figure 6.2 Multiplications needed for K right-hand sides with a 100 × 100 matrix: eliminating every time grows like $Kn3/3$, factoring once like $n3/3 + Kn2$. 156

Figure 6.3 The least squares blend: Ax is the point of the column space closest to the target b, and the miss $r = b - Ax$ is perpendicular to every column. 158

Figure 6.4 Largest error of the least squares solution for polynomial columns of degree d at 20 points: the normal equations lose about twice as many digits as Householder QR, and from degree 12 Cholesky fails. 177

Figure 7.1 The lying tank of the fuel kiosk: the dipstick measures the depth h, and the volume is the area of the circular segment times the length. 185

Figure 7.2 Runge's function and its degree-10 interpolants: through 11 equally spaced points the polynomial swings wildly near the ends; through 11 Chebyshev points it follows the function closely. 189

Figure 7.3 Error of piecewise linear and degree-6 polynomial interpolation from the 20 cm marks, measured against the exact tank formula. 206

Figure 7.4 Error of the natural spline, the clamped spline (end slopes 0) and the degree-6 polynomial through the 20 cm marks. Near the ends, where the tank's curvature is infinite, the splines suffer most. 207

Figure 8.1 Forty days of Mas Dimas's notebook: fuel used against distance ridden, the least squares line, and the residuals, the vertical misses that the line makes as small as possible in the sum of squares. 216

Figure 8.2 RMSE of polynomial fits of degree 1 to 8 on the 24 fitting days and on the 12 held-back days. The fitting error only falls; the held-back error is smallest at degree 2 and grows sharply from degree 5. 222

Figure 8.3 Fuel economy against average speed on 36 days, with the degree-2 fit (whose peak at 45.8 km/h is marked) and the degree-8 fit, which follows individual days instead of the trend. 234

Figure 8.4 Battery health of Mas Dimas's phone every two months, the exponential fit through logarithms, and the 80% level it reaches after about 23.4 months. 235

Figure 9.1 Error of the forward and central differences of x at $x = 4$ against the step h (both axes logarithmic, h shrinking to the right). Truncation error falls like h or $h2$; round-off error rises like ε/h. 250

Figure 9.2 Speed of Mas Dimas's trip (top) and its acceleration (bottom) from the plain three-point formulas and from the five-point least-squares formulas; only the smoothed curve separates the real hard brake from the false alarm. 263

Figure 10.1 The orchard between the straight village road and the river bank, with the seventeen offsets measured every 10 m; the highlighted strip is one of the sixteen trapezoids of the trapezoidal rule. 272

Figure 10.2 Error against the number of function values for $01dx/(1 + x) = \ln 2$: trapezoid (order 2), Simpson (order 4), Romberg and Gauss–Legendre. 290

Figure 11.1 The draining tank: the exact solution, Euler's method with a 10-minute step (each step follows the tangent at its start and drains a little too much), and the classical Runge–Kutta method with the same step. 305

Figure 11.2 Water level on two mornings: with the pump left on until 07:30 the tank overflows for three quarters of an hour; with the pump switched off at 06:42, as planned by bisection, it peaks at 98 cm and never overflows. 318

Figure 11.3 The seat with a worn shock absorber after a speed bump: RK4 shows the swing dying away, while Euler's method with the same step makes it grow, an instability of the method, not of the motorbike. 322

Figure 12.1 Shares of the three laundries month by month from 40%, 35% and 25%; the dotted lines are the stationary shares, the eigenvector of the switching matrix for the eigenvalue 1. 335

Figure 12.2 The three mode shapes of the shophouse model: sideways displacement of each floor (ground at 0), scaled so that the roof moves one unit, with the natural frequency of each mode. 344

Figure 12.3 The village-tourism network: arrows are links, circle areas are proportional to PageRank, and the waterfall guide (amber) is a dangling page with no links out. 349

Figure 13.1 Mbak Rina's daily profit and the golden-section intervals of steps 0 to 6: each bar is 0.618 times as long as the one above it, and all of them contain the best price. 376

Figure 13.2 Contours of the daily profit (thousand rupiah) over the two prices, with the paths of gradient descent (step 0.008), a fixed step above the stability limit (0.012), and Newton's method, all starting from (3, 1). 378

Figure 14.1 Loss above the least-squares optimum during training: full-batch gradient descent on raw and on standardized inputs, and mini-batch SGD, momentum and Adam on standardized inputs (both axes logarithmic). 413

Figure 14.2 Probability that an order is cancelled, as learned by logistic regression, against the pickup distance for four combinations of rain and rush hour; squares show the observed share in half-kilometer bins over all conditions. 414

Figure 15.1 Time to solve the 400 × 400 griddle on the two-core test machine: Jacobi (estimated from the measured time per sweep and the number of sweeps needed), red-black SOR and SciPy's sparse direct solver (measured). The scale is logarithmic. 438

Figure 15.2 Steady temperature of the 40 cm griddle on a 1 mm grid, with the left edge at 200, the right at 60, the top at 80 and the bottom at 150 degrees Celsius; white lines are isotherms. 438

Figure 16.1 The orchard's water system as a simulation: every part is computed by methods of earlier chapters, the parts meet in the tank, and many simulated seasons feed the choice of tank size. 446
Figure 16.2 One simulated dry season (the same weather) with a 13,000-liter and a 30,000-liter tank: water in the tank after the morning watering, days on which the trees got less than their ration (crosses), and daily rain (bars, right axis). 483
Figure 16.3 Yearly cost against tank capacity from 200 simulated seasons per size: tank cost, value of missed water, and total, with the capacities tried by the golden-section search and the chosen size. 484

List of Tables

Table 1.1 The tools of the workbench and the versions used throughout the book. 3
Table 1.2 The two IEEE 754 formats used in this book. 15
Table 2.1 The loan function Equation (2.3) at five monthly rates: it changes sign between 2% and 3%. 37
Table 2.2 Dimas's debt month by month at the effective rate: the interest falls as the debt falls, and the twelfth instalment clears the debt exactly. 49
Table 3.1 The root finders of Chapters 2 and 3 compared: what each needs, how fast it converges near a simple root, and whether it is guaranteed to succeed. 82
Table 5.1 Sweeps needed to bring the largest change below $10 - 6$ on the $n \times n$ griddle. Gauss-Seidel needs about half of Jacobi's sweeps, as $\rho GS = \rho J2$ predicts; optimal SOR grows only linearly with n. 142
Table 6.1 Grams of each nutrient in 100 g of each fertilizer, and the recommendation for one young tree. 152
Table 6.2 The three decompositions of this chapter, what they need, what they cost and what they are for. 161
Table 7.1 The divided-difference table for the first four marks; the top entry of each column is a coefficient of the Newton form. 187
Table 8.1 The least squares line through the first five days, by hand: deviations from the means, their squares and their products. 217
Table 9.1 Five distance readings from Mas Dimas's trip, one per second, around the hard brake. 243
Table 9.2 Finite-difference formulas for the first and second derivative, with the number of function values they need and their truncation error (ξ omitted). 247
Table 13.1 The optimizers of this chapter: what they need, what a step costs, and how fast they converge. 364
Table 16.1 Where each chapter's methods work in the orchard simulation. 447

List of Listings

Listing 1.1 `Hello.java`: the first Java program, which also reports the Java version and the operating system. 8
Listing 1.2 `hello.py`: the first Python program. 8
Listing 1.3 `check_setup.py`: checks Python, the virtual environment, NumPy, SciPy and Java. 9
Listing 1.4 `CheckSetup.java`: checks the Java version and looks for Python. 11
Listing 1.5 `float_lab.py`: five experiments with double-precision numbers. 15
Listing 1.6 `FloatLab.java`: the same experiments in Java. 16
Listing 1.7 `exp_series.py`: *ex* from its Taylor series with the stopping rule of Equation (1.4). 19

Listing 1.8 `ExpSeries.java`: the same series in Java. 20
Listing 1.9 `diff_step.py`: the forward difference of x with ever smaller steps. 22
Listing 1.10 `DiffStep.java`: the same experiment in Java. 22
Listing 1.11 `laundry_ledger.py`: a year of laundry receipts added up as integers, as doubles, as floats, and as floats with Kahan's compensation. 25
Listing 1.12 `LaundryLedger.java`: the same ledger in Java. 25
Listing 1.13 `compare_library.py`: machine epsilon, the next double, float32 and three ways to sum, with NumPy 2.x. 27
Listing 1.14 `CompareLibrary.java`: the same facts from the Java standard library. 28
Listing 1.15 `harmonic32.py`: the harmonic sum in single precision, forward and backward. 32
Listing 1.16 `Harmonic32.java`: the same in Java. 32
Listing 2.1 `roots.py`: bisection, regula falsi and the Illinois variant, with an optional iteration table. 42
Listing 2.2 `Roots.java`: the same three methods in Java, returning a `Result` record with the root and the number of iterations. 44
Listing 2.3 `loan_rate.py`: the hidden monthly rate of an instalment plan by three methods. 45
Listing 2.4 `LoanRate.java`: the same program in Java. 46
Listing 2.5 `stagnation.py`: the three methods on $x10 - 1$. 50
Listing 2.6 `Stagnation.java`: the same comparison in Java. 50
Listing 2.7 `test_roots.py`: unit tests of the bracketing methods. 51
Listing 2.8 `RootsTest.java`: the same checks in Java. 52
Listing 2.9 `compare_scipy.py`: the loan rate with SciPy's `bisect`, `brentq` and TOMS 748. 53
Listing 2.10 `sphere_tank.py`: the depth of water in a spherical tank for given volumes. 57
Listing 2.11 `SphereTank.java`: the same in Java. 58
Listing 3.1 `roots2.py`: Newton-Raphson, secant and fixed-point iteration. 67
Listing 3.2 `Roots2.java`: the same three methods in Java. 68
Listing 3.3 `dipstick.py`: the depth marks of a lying cylindrical tank by Newton-Raphson and secant. 69
Listing 3.4 `Dipstick.java`: the same program in Java. 70
Listing 3.5 `convergence.py`: 2 by three methods, with the measured ratio and order of convergence. 72
Listing 3.6 `Convergence.java`: the same measurement in Java. 73
Listing 3.7 `failures.py`: divergence, a cycle, a flat tangent, and a good start. 76
Listing 3.8 `Failures.java`: the same four cases in Java. 77
Listing 3.9 `test_roots2.py`: unit tests of the open methods and the dipstick. 79
Listing 3.10 `Roots2Test.java`: the same checks in Java. 80
Listing 3.11 `compare_scipy.py`: the 3,000-litre mark with SciPy's `newton` (with and without the derivative) and `brentq`. 82
Listing 3.12 `loan_secant.py`: Dimas's loan rate by the secant method. 86
Listing 3.13 `LoanSecant.java`: the same in Java. 86
Listing 4.1 `op_count.py`: counted flops of Gaussian elimination against the formula Equation (4.6) 92
Listing 4.2 `OpCount.java`: the same count in Java. 93
Listing 4.3 `gauss.py`: Gaussian elimination with optional partial pivoting, back substitution, the residual, and a data reader. 96
Listing 4.4 `Gauss.java`: the same module in Java. 97
Listing 4.5 `receipts.py`: three receipts, three prices, with every elimination step printed. 101
Listing 4.6 `Receipts.java`: the same program in Java. 101
Listing 4.7 `fertilizer.py`: the grams of each fertilizer per tree, and the kilograms for the orchard. 102
Listing 4.8 `Fertilizer.java`: the same program in Java. 103
Listing 4.9 `conditioning.py`: Hilbert matrices of growing size: condition number, residual and error. 107
Listing 4.10 `Conditioning.java`: the same experiment in Java. 107
Listing 4.11 `pivot_demo.py`: the zero pivot and the tiny pivot, with and without partial pivoting. 109

Listing 4.12 `PivotDemo.java`: the same demonstration in Java. 109
Listing 4.13 `test_gauss.py`: unit tests of Gaussian elimination. 110
Listing 4.14 `GaussTest.java`: the same checks in Java. 111
Listing 4.15 `compare_numpy.py`: the fertilizer system and the Hilbert matrices with NumPy and SciPy. 113
Listing 5.1 `iterative_solvers.py`: Jacobi, Gauss-Seidel and SOR sweeps for a small dense system, with a diagonal-dominance test and a residual check. 127
Listing 5.2 `IterativeSolvers.java`: the same three methods in Java. 129
Listing 5.3 `griddle.py`: the $n \times n$ griddle solved by Jacobi, Gauss-Seidel and SOR sweeps on a bordered grid, with sweep counts, true errors, an ω scan and a temperature map. 133
Listing 5.4 `Griddle.java`: the griddle solver in Java. 136
Listing 5.5 `compare_scipy.py`: the griddle checked against SciPy's sparse direct solver and its conjugate gradient method. 144
Listing 6.1 `lu_blend.py`: LU decomposition with partial pivoting, the determinant, and the blend for every block from one factorization. 161
Listing 6.2 `LuBlend.java`: the same factor-once, solve-many program in Java. 163
Listing 6.3 `least_blend.py`: least squares by the normal equations with Cholesky and by Householder QR, for the orchard and for polynomial columns. 167
Listing 6.4 `LeastBlend.java`: the same two least squares methods in Java. 170
Listing 6.5 `compare_scipy.py`: our factors and solutions compared with SciPy's `lu`, `lu_factor`/`lu_solve`, `cho_factor`/`cho_solve` and NumPy's `lstsq` and `qr`. 177
Listing 7.1 `interp.py`: linear, Lagrange and Newton interpolation of the dipstick table, errors against the exact tank formula, and Runge's experiment with equally spaced and Chebyshev points. 192
Listing 7.2 `Interp.java`: the same interpolation program in Java. 195
Listing 7.3 `spline.py`: natural and clamped cubic splines by the tridiagonal system of curvatures and the Thomas algorithm. 200
Listing 7.4 `Spline.java`: the cubic splines in Java. 201
Listing 7.5 `compare_scipy.py`: our interpolants against `numpy.interp`, `BarycentricInterpolator` and `CubicSpline`. 208
Listing 8.1 `lsq.py`: CSV reading, least squares by Householder QR, prediction and $R2$, shared by the programs of this chapter. 223
Listing 8.2 `Lsq.java`: the shared helpers in Java. 225
Listing 8.3 `line_fit.py`: the least squares line with $R2$, standard error and slope uncertainty; two predictors by QR; the exponential battery curve. 227
Listing 8.4 `LineFit.java`: the same fits in Java. 228
Listing 8.5 `poly_fit.py`: polynomial fits of degree 1 to 8 judged on held-back days, and the best cruising speed from the parabola. 230
Listing 8.6 `PolyFit.java`: the polynomial fits in Java. 231
Listing 8.7 `compare_numpy.py`: the fits checked against `linregress`, `lstsq`, `Polynomial.fit` and a direct nonlinear `curve_fit` of the battery curve. 235
Listing 9.1 `diff_steps.py`: forward and central differences of x at $x = 4$ for ever smaller steps. 248
Listing 9.2 `DiffSteps.java`: the same experiment in Java. 248
Listing 9.3 `richardson.py`: the Richardson table for the central difference. 252
Listing 9.4 `Richardson.java`: the same table in Java. 252
Listing 9.5 `speed.py`: speed and acceleration from the distance log, with harsh braking flagged. 254
Listing 9.6 `Speed.java`: the same analyzer in Java. 256
Listing 9.7 `smooth.py`: speed and acceleration from a five-point least-squares parabola, next to the plain formulas. 259
Listing 9.8 `Smooth.java`: the same comparison in Java. 260
Listing 9.9 `compare_numpy.py`: the trip speeds from `np.gradient` and the derivative of x from `scipy.differentiate.derivative`. 264
Listing 9.10 `uneven.py`: speeds from unevenly spaced readings. 268

Listing 9.11 `Uneven.java`: the same computation in Java. 269
Listing 10.1 `orchard.py`: the orchard's area by the trapezoidal and Simpson rules, with error estimates from halving the data. 276
Listing 10.2 `Orchard.java`: the same calculator in Java. 277
Listing 10.3 `quadrature.py`: composite trapezoid and Simpson rules for a known integral, with the ratios of successive errors. 280
Listing 10.4 `Quadrature.java`: the same check in Java. 281
Listing 10.5 `romberg.py`: the Romberg table, adding only the new midpoints on each row. 283
Listing 10.6 `Romberg.java`: the same table in Java. 284
Listing 10.7 `gauss.py`: Gauss–Legendre quadrature with one to five points. 287
Listing 10.8 `Gauss.java`: the same rules in Java, with the nodes in a three-dimensional array. 288
Listing 10.9 `adaptive.py`: adaptive Simpson integration, compared with uniform Simpson. 291
Listing 10.10 `Adaptive.java`: the same comparison in Java. 292
Listing 10.11 `compare_scipy.py`: the orchard and the test integrals with SciPy. 294
Listing 11.1 `tank_euler.py`: Euler's method for the draining tank, compared with the exact solution. 302
Listing 11.2 `TankEuler.java`: the same in Java. 303
Listing 11.3 `ode_orders.py`: Euler, Heun, midpoint and RK4 as interchangeable step functions, with the error ratios for halved steps. 307
Listing 11.4 `OdeOrders.java`: the same comparison in Java. 308
Listing 11.5 `morning.py`: one morning of the drip-irrigation tank with RK4, including the pump switch, overflow and running dry. 311
Listing 11.6 `Morning.java`: the same simulator in Java. 312
Listing 11.7 `pump_plan.py`: bisection on the pump-off time, with the RK4 simulation as the function whose root is sought. 316
Listing 11.8 `PumpPlan.java`: the same planner in Java. 316
Listing 11.9 `bump.py`: the seat after a speed bump as a system of two equations, with Euler and RK4. 319
Listing 11.10 `Bump.java`: the same system in Java, with the vector operations written out in a helper `axpy`. 319
Listing 11.11 `compare_scipy.py`: the tank and the seat with `solve_ivp`. 323
Listing 11.12 `semi_implicit.py`: semi-implicit Euler for the worn shock. 327
Listing 11.13 `SemiImplicit.java`: the same in Java. 327
Listing 12.1 `market.py`: the laundry shares month by month, the power method on the switching matrix. 332
Listing 12.2 `Market.java`: the same in Java. 333
Listing 12.3 `power.py`: power, inverse and shifted inverse iteration for the three modes of the shophouse. 338
Listing 12.4 `Power.java`: the same methods in Java; the result of an iteration is a small `record`. 340
Listing 12.5 `pagerank.py`: PageRank by the power method, with dangling pages handled explicitly. 346
Listing 12.6 `PageRank.java`: the same in Java; the ranking is sorted with a comparator. 346
Listing 12.7 `compare_numpy.py`: the market, the shophouse and PageRank with `numpy.linalg`. 350
Listing 12.8 `deflation.py`: Hotelling deflation to find the second highest mode. 354
Listing 12.9 `Deflation.java`: the same in Java. 355
Listing 13.1 `golden.py`: golden-section search for the best tea price. 364
Listing 13.2 `Golden.java`: the same search in Java. 366
Listing 13.3 `newton1d.py`: Newton's method for the best price, plain and safeguarded. 367
Listing 13.4 `Newton1D.java`: the same in Java. 368
Listing 13.5 `twoprices.py`: gradient descent with fixed and backtracking steps, and Newton's method, for two prices. 369
Listing 13.6 `TwoPrices.java`: the same optimizers in Java. 372
Listing 13.7 `check_scipy.py`: the same problems solved by SciPy's optimizers. 378

Listing 14.1 `mlcore.py`: data, least squares by normal equations and by QR, a portable random-number generator, and stable logistic functions. 391
Listing 14.2 `MlCore.java`: the same building blocks in Java. 394
Listing 14.3 `linfit.py`: the trip-time model by the normal equations and by QR, and the polynomial experiment. 398
Listing 14.4 `LinFit.java`: the same fits in Java. 399
Listing 14.5 `train_gd.py`: full-batch, mini-batch, momentum and Adam training of the trip-time model. 401
Listing 14.6 `TrainGd.java`: the same experiment in Java. 403
Listing 14.7 `logistic.py`: logistic regression by Newton's method (IRLS) and by gradient descent, with a gradient check and stable formulas. 406
Listing 14.8 `Logistic.java`: the same in Java. 408
Listing 14.9 `check_numpy.py`: the chapter's results checked with NumPy and SciPy, the condition numbers, and a float32 experiment. 414
Listing 15.1 `summation.py`: naive, chunked, pairwise and Kahan summation. 426
Listing 15.2 `Summation.java`: the same sums in Java. 427
Listing 15.3 `griddle_loops.py`: Jacobi sweeps on the fine griddle with plain loops. 429
Listing 15.4 `griddle_numpy.py`: vectorized Jacobi sweeps, optionally split over threads. 430
Listing 15.5 `Griddle.java`: Jacobi sweeps in Java on one thread or several. 431
Listing 15.6 `griddle_rb.py`: red-black SOR with NumPy. 433
Listing 15.7 `GriddleRb.java`: red-black SOR in Java on one thread or several. 434
Listing 15.8 `check_numpy.py`: exact and pairwise sums, and the griddle by a sparse direct solver. 438
Listing 16.1 `toolbox.py`: the numerical methods of Chapters 2 to 15 that the orchard simulation reuses. 451
Listing 16.2 `Toolbox.java`: the same methods in Java. 457
Listing 16.3 `orchard.py`: the orchard's water system as one model. 464
Listing 16.4 `Orchard.java`: the orchard model in Java. 466
Listing 16.5 `components.py`: every part of the model, checked against a second method. 471
Listing 16.6 `Components.java`: the same checks in Java. 472
Listing 16.7 `sensor.py`: the drip coefficient estimated from the level sensor. 473
Listing 16.8 `Sensor.java`: the same estimate in Java. 474
Listing 16.9 `montecarlo.py`: many dry seasons in parallel, reproducibly. 475
Listing 16.10 `MonteCarlo.java`: the same Monte Carlo run with Java threads. 477
Listing 16.11 `design.py`: the tank size by golden-section search on the yearly cost. 478
Listing 16.12 `Design.java`: the same search in Java. 479
Listing 16.13 `check_scipy.py`: the parts of the model recomputed with NumPy and SciPy. 484

List of Pseudocode

Pseudocode 2.1 The bisection method with a tolerance on the change of the midpoint. 37
Pseudocode 2.2 Regula falsi, with the Illinois modification switched on by a flag (lines 10 and 14). 40
Pseudocode 3.1 The Newton-Raphson method with its three safeguards. 63
Pseudocode 4.1 Gaussian elimination with optional partial pivoting (lines 2 to 4) and back substitution. 92
Pseudocode 5.1 Successive over-relaxation; with omega = 1 it is the Gauss-Seidel method. 122
Pseudocode 6.1 LU decomposition with partial pivoting, stored in place, and the two triangular solves. 154
Pseudocode 13.1 Golden-section search for the maximum of a unimodal function on $[a, b]$. 360

Pseudocode 15.1 Kahan's compensated summation: *c* recovers the rounding error of each addition and feeds it into the next. 426

List of Equations

(1.1) 12
(1.2) 12
(1.3) 12
(1.4) 12
(1.5) 13
(1.6) 13
(1.7) 14
(1.8) 14
(1.9) 18
(1.10) 19
(1.11) 19
(1.12) 19
(1.13) 22
(1.14) 24
(1.15) 24
(1.16) 27
(2.1) 36
(2.2) 36
(2.3) 36
(2.4) 39
(2.5) 39
(2.6) 40
(2.7) 41
(2.8) 42
(3.1) 61
(3.2) 62
(3.3) 62
(3.4) 64
(3.5) 64
(3.6) 65
(3.7) 65
(3.8) 65
(3.9) 66
(3.10) 72
(4.1) 90
(4.2) 90
(4.3) 91
(4.4) 91
(4.5) 92
(4.6) 92
(4.7) 94
(4.8) 94
(4.9) 95

(4.10) 95
(4.11) 102
(4.12) 106
(4.13) 106
(5.1) 119
(5.2) 120
(5.3) 122
(5.4) 122
(5.5) 122
(5.6) 122
(5.7) 123
(5.8) 123
(5.9) 123
(5.10) 124
(5.11) 124
(5.12) 125
(5.13) 125
(5.14) 125
(5.15) 126
(6.1) 152
(6.2) 153
(6.3) 153
(6.4) 154
(6.5) 155
(6.6) 156
(6.7) 157
(6.8) 157
(6.9) 157
(6.10) 158
(6.11) 159
(6.12) 160
(6.13) 160
(7.1) 184
(7.2) 185
(7.3) 186
(7.4) 187
(7.5) 187
(7.6) 188
(7.7) 188
(7.8) 188
(7.9) 189
(7.10) 190
(7.11) 190
(7.12) 191
(7.13) 191
(8.1) 216
(8.2) 217
(8.3) 218
(8.4) 218
(8.5) 218
(8.6) 219

(8.7) 219
(8.8) 220
(8.9) 220
(8.10) 221
(8.11) 221
(8.12) 221
(8.13) 222
(9.1) 244
(9.2) 245
(9.3) 246
(9.4) 246
(9.5) 247
(9.6) 250
(9.7) 251
(9.8) 259
(10.1) 273
(10.2) 274
(10.3) 274
(10.4) 275
(10.5) 275
(10.6) 276
(10.7) 276
(10.8) 283
(10.9) 286
(10.10) 287
(11.1) 300
(11.2) 301
(11.3) 301
(11.4) 302
(11.5) 302
(11.6) 306
(11.7) 306
(11.8) 318
(11.9) 322
(11.10) 324
(12.1) 331
(12.2) 332
(12.3) 332
(12.4) 336
(12.5) 337
(12.6) 338
(12.7) 345
(13.1) 357
(13.2) 358
(13.3) 359
(13.4) 359
(13.5) 360
(13.6) 360
(13.7) 361
(13.8) 361
(13.9) 361

(13.10) 361
(13.11) 362
(13.12) 362
(13.13) 362
(13.14) 362
(13.15) 363
(13.16) 363
(13.17) 364
(14.1) 386
(14.2) 386
(14.3) 387
(14.4) 387
(14.5) 387
(14.6) 387
(14.7) 388
(14.8) 388
(14.9) 389
(14.10) 389
(14.11) 389
(14.12) 390
(14.13) 390
(14.14) 390
(14.15) 390
(14.16) 390
(14.17) 391
(15.1) 423
(15.2) 424
(15.3) 425
(15.4) 425
(15.5) 425
(15.6) 425
(16.1) 448
(16.2) 448
(16.3) 448
(16.4) 448
(16.5) 448
(16.6) 449
(16.7) 449
(16.8) 450
(16.9) 450
(16.10) 450
(16.11) 451

Preface

A laundry's spreadsheet says that 4.1 kg at Rp7,500 a kilogram is not quite Rp30,750. A water seller marks a dipstick for a tank that lies on its side. A martabak seller wants to know where his griddle is hot enough. An ojek driver wonders what "1.67% a month" really costs him. None of these questions has a neat formula for its answer, and every one of them can be answered, to as many correct digits as anyone needs, by a short program that repeats a simple step until the answer stops changing. That is what numerical methods are: the craft of getting answers that are wrong by an amount we know and can accept.

This book grew out of the course Numerical Methods at ITS, where the subject has been taught to students of informatics for many years, and it has been written from the ground up for 2026. Its methods are the classic ones — bisection and Newton's method, Gaussian elimination and its iterative cousins, interpolation, least squares, differentiation and integration, differential equations, eigenvalues and optimization — because they are the foundation of every scientific library and every machine-learning framework in use today. Its applications are new, and each of them answers a question that somebody in a laundry, an orchard, a market stall or on the back of a motorbike really asks.

Who this book is for

The book is self-contained, and you are assumed to be a beginner. If you have never written a line of code, never installed a programming language and do not remember much calculus, you are exactly the reader this book was written for. Every term is introduced in plain words before it is used, every method is first worked by hand on a small example with real numbers, the mathematics it needs is explained where it is needed, and every program in the book was run before its output was printed here. Chapter 1 begins with a receipt that does not add up and ends with a workbench that is ready for everything that follows.

Readers who already program, and students who need the rigor of a university course, will find the other half of the book: each method derived properly, with its error and its rate of convergence analyzed, measured in code and compared with what the textbooks predict, and complete, tested programs ready to be adapted to their own problems. That principle makes the book long. A shorter book would assume you already knew the hard parts, and such a book is no use to the person who does not.

One idea, two languages

Every program in this book is written twice, in **Java** and in **Python**, and the two versions do the same thing, read the same data and print identical output, digit for digit. Java is the language of large, long-lived systems, of Android phones and of many banks and government services; its compiler catches many mistakes before the program runs. Python is the language of quick experiments, of data work and of most scientific computing; it is short and easy to read. Seeing one method written both ways is the quickest route to understanding the method itself rather than the habits of one language. If you know neither, start with Python and read the Java version after it; the comparison will teach you Java quickly.

Our own programs use only the standard libraries of the two languages, because the point is to see every step of a method. But a numerical answer deserves a second opinion, so every chapter checks its results against **NumPy** and **SciPy**, the Python libraries that wrap decades of carefully tested numerical code, and explains any difference in the last digits. All of it is written and run in **Visual Studio Code**, a free editor that works the same way on Windows, macOS and Linux; Chapter 1 installs it, together with JDK 25, Python 3.14, NumPy and SciPy, step by step on all three systems.

What is in it

Chapter 1 explains why a computer's answers are almost always slightly wrong: how numbers are stored in floating point, what round-off and truncation errors are, and how to measure them. It sets up the workbench and ends with a small laboratory of experiments on the machine's arithmetic.

Chapters 2 and 3 find roots of equations. Chapter 2 helps Mas Dimas, an ojek driver, discover the real interest rate hidden in a phone's instalment plan with bisection and regula falsi, methods that can never lose the root once it is bracketed. Chapter 3 marks the dipstick of Pak Harun's water truck with Newton-Raphson, the secant method and fixed-point iteration, measures how fast each one converges, and shows when Newton goes wrong.

Chapters 4 to 6 solve systems of linear equations. Chapter 4 works out how much of each fertilizer Bu Rahma's durian trees need with Gaussian elimination, shows why pivoting matters, and explains why a small residual does not guarantee a small error. Chapter 5 finds the steady temperature over Pak Joni's martabak griddle with the iterative methods of Jacobi, Gauss-Seidel and SOR. Chapter 6 factors a matrix once and reuses it, with LU, Cholesky and QR decompositions, to answer the fertilizer question for every block of an orchard.

Chapters 7 and 8 draw curves through data. Chapter 7 turns the tank maker's table for Pak Darto's fuel kiosk into accurate litres at every depth with Lagrange and Newton interpolation and cubic splines. Chapter 8 fits models to Mas Dimas's notebook of fuel and trips with least squares, and shows how to choose a model without overfitting.

Chapters 9 to 11 are about change. Chapter 9 explains how a ride-hailing app can tell a harsh brake from a gentle stop, with finite differences derived from Taylor's theorem and Richardson extrapolation. Chapter 10 measures a durian orchard whose boundary is a winding river with the trapezoidal rule, Simpson's rules, Romberg integration and Gauss quadrature. Chapter 11 decides when to switch off the orchard's pump with Euler and Runge–Kutta methods for ordinary differential equations, and looks at systems and stability.

Chapters 12 to 14 take the methods toward modern computing. Chapter 12 follows three laundries competing in one kampung with the power method for eigenvalues, and shows how the same idea ranks the pages of the web with PageRank. Chapter 13 finds the most profitable price for a cup of iced tea with golden-section search, Newton's method and gradient descent. Chapter 14 shows that machine learning is numerical methods at heart: least squares, gradient descent with mini-batches, momentum and Adam, and logistic regression trained by Newton's method, predicting the length of an ojek trip and whether a customer will cancel.

Chapter 15 makes the programs fast: where the time goes, vectorization, parallel threads and their limits, sums that are accurate and reproducible, and smarter methods that simply do less work. Chapter 16, the capstone, simulates the dry season of a durian orchard end to end — tank,

pump, weather, trees, well and drip line — with methods from across the book, runs hundreds of seasons, and helps Pak Hadi choose his water tank.

Every chapter follows the same path: the problem and who it helps, the idea in plain words with numbers worked by hand, the method derived with its error and convergence analyzed, the programs built step by step in Java and Python, running them, checking the answers against NumPy and SciPy, making them better, using them for real, numerical methods in practice at Profitina, common mistakes, a summary, exercises with complete solutions, and references.

Every program in this book was run before its output was printed. The programs are collected, chapter by chapter, in the accompanying `Code` folder, from `Chapter01` to `Chapter16`, each with a `java` and a `python` folder, the data it needs, and a README that explains how to run it in a terminal and in VS Code on Windows, macOS and Linux.

How to read it

With VS Code open and the programs running. Chapter 1 comes first for everyone, because it builds the workbench and explains the errors that every later chapter measures; after that the chapters climb from easy to hard, and each one names the earlier chapters it builds on. Work the small examples by hand before you read the code, and type the programs in rather than copying them; the typing is where the errors happen, and the errors are where the learning happens. When a program works, change it — a tolerance, a step size, one number in its data — and watch how the error and the number of iterations change: there is no quicker way to feel how a numerical method behaves. The exercises at the end of each chapter all have complete solutions — use them after your own attempt, not instead of it.

Numbered figures, tables, listings, pseudocode and equations are all live links: in the electronic edition, click “Listing 5.3” or “[4]” anywhere in the text and you are taken straight to it. The contents and the lists at the front of the book work the same way.

About the author

Irfan Subakti describes himself in one line — *Father | Educator | Entrepreneur | Bridging Tech & Nature: CS Lecturer × Crypto/Digital Business × AI/ML Specialist × Fruit Farmer | Storyteller* — and every part of that line has found its way into this book. He has taught in the Department of Informatics at ITS since 1998. He supervises and examines undergraduate theses, teaches MLOps for the IBM SkillsBuild program, and builds products: he founded Profitina, an AI-visibility platform, whose everyday use of numerical methods closes every chapter of this book. Away from the keyboard he tends the family fruit and durian orchard, where he learned that good software, like a good orchard, is grown patiently, one well-tended piece at a time. The storyteller in him is the reason every chapter of this book begins with a person and a problem rather than with a definition.

Acknowledgments

To the students who worked through the earlier forms of this material in the Numerical Methods course and asked the questions that showed where an explanation was still thin, and to the colleagues at ITS with whom numerical methods have been taught and refined over the years. To the launderers, growers, drivers, water sellers and stall keepers whose everyday problems gave each chapter its story. The book owes its standard, too, to the official documentation of Java,

Python, NumPy, SciPy and VS Code, and to the classic textbooks and research literature on numerical analysis, which are cited wherever they are used.

Surabaya, September 2026



Irfan Subakti · yifana@gmail.com

Profitina

profitina.com

Chapter 1

Numbers in the Machine: Errors, Floating Point, and Your Workbench

A computer is fast and tireless, but it does not calculate with the real numbers of mathematics. It calculates with a finite set of binary numbers that only approximate them, and almost every answer it gives is slightly wrong. Numerical methods are the craft of getting answers that are wrong by an amount we know and can accept. This chapter lays the foundation for the whole book. It sets up your workbench, step by step, on Windows, macOS and Linux: the Java Development Kit (JDK) 25, Python 3.14, Visual Studio Code, and the NumPy and SciPy libraries that later chapters use to check our own programs. Then it opens the machine and looks at how numbers are stored. By the end of the chapter you will be able to install and check a professional Java and Python setup; measure an error as absolute, relative and approximate error, and decide when an iteration may stop; explain how a double-precision number is stored in 64 bits and why 0.1 + 0.2 is not 0.3; tell round-off error from truncation error and find the step size where the two balance; recognize and repair catastrophic cancellation; and add up a year of shop receipts without losing a single rupiah.

1.1 A Receipt That Does Not Add Up

Bu Wati runs a small laundry near a university campus. Students bring their clothes in plastic bags, she weighs each bag on a digital scale that shows kilograms with one decimal, and she charges Rp7,500 per kilogram. A bag of 4.1 kg costs Rp30,750. At the end of each day she copies the orders from her notebook into a spreadsheet that her nephew set up on an old laptop, and at the end of each month she compares the spreadsheet's total with the money in the cash box.

One evening her nephew shows her something strange. In a cell of the spreadsheet he typed the weight 4.1 and the price 7500 and asked for the product, then asked whether the product equals 30750. The spreadsheet answered FALSE. When he widened the column to show more decimals, the product appeared as 30749.999999999996. Nobody made a typing mistake. The laptop simply cannot hold the number 4.1 exactly, and the tiny difference survives the multiplication.

For one receipt, four trillionths of a rupiah do not matter. But Bu Wati's laundry takes about sixty orders a day, every day of the year: more than twenty thousand receipts. The nephew also wrote a small cash-register program for a cheap tablet that stores amounts in a shorter number format, to save memory, and there the yearly total came out Rp826 short. That is not a fortune, but it is a total that is *wrong*, and a total that is wrong for no visible reason destroys trust. If the program can lose Rp826, what else can it lose?

Bu Wati's question, "why does the computer not add up my receipts correctly?", is the question with which numerical methods begin. The same small errors live inside every calculation a computer makes: in the weather model of BMKG, in the bank's instalment table, in the app that tells an ojek driver how far he has travelled, and in every program of this book. They are usually harmless. Sometimes they are not. During the Gulf War of 1991, a Patriot missile battery in Dhahran failed to intercept an incoming missile because its clock counted time in tenths of a

second with a number format that could not hold 0.1 exactly; after about 100 hours of operation the accumulated error was about 0.34 s, enough to look for the target in the wrong place, and 28 soldiers died [1]. The error per tick was about one ten-millionth of a second. What killed was the accumulation.

This chapter explains where such errors come from, how to measure them, and how to keep them small. Section 1.13 returns to Bu Wati's ledger and adds up her year exactly. First, though, we need a workbench.

1.2 What Numerical Methods Are

School mathematics solves problems with formulas. The equation $2x + 3 = 11$ has the solution $x = 4$, and the quadratic equation $ax^2 + bx + c = 0$ has the famous formula for its roots. Real problems rarely offer such formulas. Chapter 2 asks what monthly interest rate is hidden in an instalment plan; the equation is simple to write down, but no formula of algebra solves it for the rate. Chapter 3 marks the dipstick of a water truck whose tank lies on its side; the volume of water at a given depth has a formula, but the depth for a given volume does not. Chapter 4 finds how much of each fertilizer a durian tree needs from four equations with four unknowns, which has a formula in principle but not one anybody would want to use by hand for ten or a thousand unknowns.

A **numerical method** replaces such a problem with a finite sequence of additions, subtractions, multiplications and divisions that a computer can carry out, and that produces an answer as close to the true one as we need. Bisection, the first method of Chapter 2, is a good example: to find where a function changes sign, it halves an interval again and again; after 20 halvings the answer is known to one part in a million. The answer is never exact, but we know *how* inexact it is, and we can buy more accuracy with more work.

Every method in this book is therefore judged by the same three questions, which were already the questions of the classic textbooks [2], [3]:

1. **Does it converge?** As we spend more work (more iterations, a smaller step), does the error go to zero?
2. **How fast?** Does each iteration add a fixed number of correct digits, or double them? This is the *order of convergence*, and Chapter 3 measures it.
3. **Is it stable?** Do the small errors of the machine stay small, or does the method amplify them until the answer is useless? Section 1.11 shows a formula that is correct in algebra and useless on a computer.

The book moves from the basic tools to their modern uses. Chapters 2 and 3 find roots of equations; Chapters 4 to 6 solve systems of linear equations; Chapters 7 and 8 fit curves through data; Chapters 9 to 11 differentiate, integrate and solve differential equations; Chapters 12 and 13 find eigenvalues and optima; Chapter 14 shows how all of this sits inside machine learning; Chapter 15 makes it fast; and Chapter 16 simulates a real system from start to finish. Every chapter starts from a problem that ordinary people have, works the idea by hand with real numbers, derives the method with its error analysis, builds it in Java and in Python, runs it, and compares our answer with the answer of a professional library.

1.3 Your Workbench: What We Install and Why

A carpenter's first job in a new workshop is to set up the bench. Ours has the tools summarized in **Table 1.1**. The versions are the current ones as of September 2026; later patch releases (a JDK 25.0.5 or a Python 3.14.8) work exactly the same.

Table 1.1 The tools of the workbench and the versions used throughout the book.

Tool	What it is	Version used in this book
JDK (Java Development Kit)	The Java compiler `javac`, the Java runtime `java` and the standard library	JDK 25 LTS, update 25.0.4 (July 2026) [4], [5]
Python	The Python interpreter `python` with its standard library, `pip` and `venv`	3.14.7 (August 5, 2026) [6]
NumPy and SciPy	Python libraries for arrays, linear algebra, root finding and much more	NumPy 2.5.3, SciPy 1.18.1 [7], [8]
Visual Studio Code	A free code editor for Windows, macOS and Linux	1.139 (September 23, 2026) [9]
Extension Pack for Java	Adds Java editing, running and debugging to VS Code	from the Extensions view [10]
Python extension	Adds Python editing (Pylance), running and debugging to VS Code	from the Extensions view [11]

Why JDK 25? Java has a new version every six months, and every two years one of them is a *long-term support* (LTS) release that receives security updates for many years. JDK 25 was released on September 16, 2025 as the current LTS [5]; the July 2026 update is 25.0.4 [4]. JDK 27 appeared on September 15, 2026 [12], but it is a short-term release, and for a book whose programs should keep working for years the LTS is the right choice. Every program in this book uses only features that exist since Java 17, so JDK 17 and 21 also run them. There is one small difference worth knowing: since JDK 19, Java prints a `double` with the shortest decimal string that identifies it, exactly as Python does, while older versions sometimes printed one digit too many [13]. With JDK 19 or newer, the Java and Python programs of this book print the same output byte for byte.

There are several builds of the same JDK, all made from the open-source OpenJDK code: Oracle's own, Eclipse Temurin from the Adoptium project, Microsoft's, Amazon Corretto, and the packages of Linux distributions. They behave identically for our purposes. We use **Eclipse Temurin** because it is free for any use, has a simple installer for every system, and appears in the package managers of all three systems; the September 2026 security release is Temurin 25.0.4.1 [14].

Why Python 3.14? Python 3.14 was first released on October 7, 2025 and brought, among other things, template strings, deferred evaluation of annotations, an officially supported build without the global interpreter lock, and a friendlier interactive shell [15]. The programs of this book do not need those features; they use plain, long-established Python and also run on Python 3.11 or newer. We install 3.14 because it is current and because the libraries we compare against are built and tested for it.

Why NumPy and SciPy? Our own programs use only the standard libraries of Java and Python, because the point is to see every step of a method. But a numerical result deserves a second opinion. NumPy and SciPy wrap decades of carefully tested numerical code (LAPACK for linear algebra, Brent's method for roots, and much more), and at the end of every chapter we ask them the same question and compare. Java has no single equivalent, so the Java side stays with plain code and the standard library's own tools such as `BigDecimal` and `Math.ulp`.

Why VS Code? Because it is free, runs on all three systems, handles Java and Python equally well, and has a built-in terminal. Any other editor works too; IntelliJ IDEA and PyCharm are excellent.

Before installing anything, it helps to know what happens when you "run" a program in each language, because the difference explains most first-day error messages. **Figure 1.1** shows the two paths.

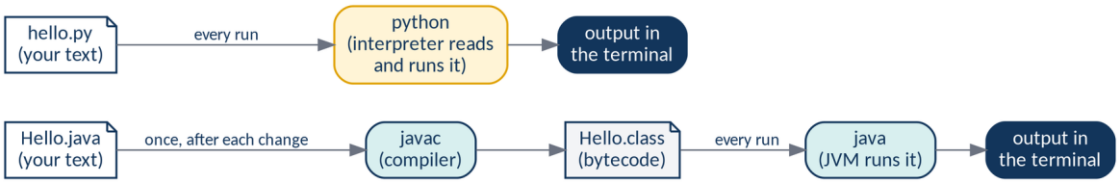


Figure 1.1 How a Java program and a Python program run. Java is compiled once by `javac` into bytecode, which the Java Virtual Machine runs; Python is read and run by the interpreter every time.

A Java program is text in a file such as `Hello.java`. The compiler `javac` checks it and translates it into *bytecode*, stored in `Hello.class`, and the command `java` starts the Java Virtual Machine (JVM), which runs the bytecode. A Python program is text in a file such as `hello.py`, and the command `python` reads and runs it directly each time; mistakes that Java finds while compiling, Python finds only when it reaches the faulty line. Since Java 11 you can also run a single Java file without compiling it first, with `java Hello.java` [16]. For programs made of several files, as most chapters of this book are, we compile with `javac`.

1.4 Installing on Windows 10 and 11

1.4.1 The JDK

The simplest way is **winget**, the package manager built into Windows 10 and 11. Open **PowerShell** (press the Windows key, type `powershell`, press Enter) and type:

```
PS C:\Users\you> winget install EclipseAdoptium.Temurin.25.JDK
```

If you prefer clicking, open `https://adoptium.net`, choose **Temurin 25 (LTS)**, **Windows** and **x64** (or **aarch64** on an ARM laptop), download the `.msi` installer and run it. On the page **Custom Setup**, set both **Add to PATH** and **Set JAVA_HOME variable** to "Will be installed on local hard drive"; the second is off by default, and some tools need it.

Close PowerShell, open a new one (a window opened before the installation does not see the new program), and check both commands:

```
PS C:\Users\you> java -version
openjdk version "25.0.4" 2026-07-21 LTS
PS C:\Users\you> javac -version
javac 25.0.4
```

The first line of `java -version` is what matters; two more lines follow that name the build. If Windows answers that `java` "is not recognized as the name of a cmdlet", the installer did not add Java to the `PATH`, the list of folders in which Windows looks for commands. Run the installer again and choose **Add to PATH**, or add `C:\Program Files\Eclipse Adoptium\jdk-25...\bin` yourself through **Settings › System › About › Advanced system settings › Environment Variables**.

1.4.2 Python

On Windows, Python is now installed with the **Python install manager**, a small program that downloads, updates and switches between Python versions [17]. Get it from the Microsoft Store (search for "Python Install Manager"), from `https://www.python.org/downloads/windows/`, or with winget:

```
PS C:\Users\you> winget install 9NQ7512CXL7T -e --accept-package-agreements
```

Then open a new PowerShell window, install Python 3.14 and check it:

```
PS C:\Users\you> py install 3.14
PS C:\Users\you> python --version
Python 3.14.7
```

After this, `python`, `py` and `python3` all start Python. If `python --version` opens the Microsoft Store instead, close every terminal, open a new one and try again; the manager can also repair its settings with `py install --configure` [17].

1.4.3 VS Code

Install VS Code with `winget install Microsoft.VisualStudioCode`, or download the **User Installer** from `https://code.visualstudio.com`. In the installer, tick **Add to PATH** and **Add "Open with Code" action to Windows Explorer file context menu**: the first lets you type `code .` in a terminal to open the current folder, and the second lets you right-click a folder and open it in VS Code.

1.5 Installing on macOS

1.5.1 The JDK

Open `https://adoptium.net`, choose **Temurin 25 (LTS)**, **macOS** and **aarch64** for a Mac with an Apple chip (M1 and later) or **x64** for an older Intel Mac, download the `.pkg` file and double-click it. With the Homebrew package manager, one command does the same:

```
you@mac ~ % brew install --cask temurin@25
```

Open **Terminal** (press Cmd+Space, type "Terminal", press Enter) and check:

```
you@mac ~ % java -version
openjdk version "25.0.4" 2026-07-21 LTS
you@mac ~ % javac -version
javac 25.0.4
```

If several JDKs are installed, macOS picks the newest; `/usr/libexec/java_home -V` lists them all.

1.5.2 Python

macOS may already contain a `python3` command, but it is an older version meant for Apple's developer tools; do not build on it. Download the macOS installer for Python 3.14.7 from `https://www.python.org/downloads/macos/` and double-click it; it is a universal package that runs natively on Apple and Intel chips. At the end the installer opens the folder **Applications › Python 3.14**: double-click **Install Certificates.command** there once, so that Python (and `pip`) can make secure internet connections. With Homebrew, `brew install python@3.14` works as well. Then check:

```
you@mac ~ % python3 --version
Python 3.14.7
```

On macOS and Linux the command is `python3`, not `python`. The book writes `python3` in commands for macOS and Linux and `python` for Windows.

1.5.3 VS Code

Download VS Code for macOS from `https://code.visualstudio.com`, open the downloaded file and drag **Visual Studio Code** into **Applications**. To be able to type `code .` in the terminal, start

VS Code, press Cmd+Shift+P to open the Command Palette, and run **Shell Command: Install 'code' command in PATH**.

1.6 Installing on Linux

Most Linux distributions carry recent versions of Java and Python in their own repositories, and those are the best choice because the system keeps them updated. Ubuntu 26.04 LTS uses OpenJDK 25 as its default Java [18] and Python 3.14 as its default `python3`, but it installs neither `javac` nor Python's `venv` module unless you ask:

```
$ sudo apt update
$ sudo apt install openjdk-25-jdk python3 python3-venv python3-pip
$ java -version
openjdk version "25.0.4" 2026-07-21
$ javac -version
javac 25.0.4
$ python3 --version
Python 3.14.7
```

On Fedora the equivalent is `sudo dnf install java-25-openjdk-devel python3 python3-pip`. On an older distribution whose repository has no JDK 25, install Temurin from Adoptium's own repository following `https://adoptium.net/installation`, or keep the distribution's JDK 21: every program in this book runs on JDK 17 or newer. The same holds for Python: Ubuntu 24.04 ships Python 3.12, which runs all the programs of this book. Never remove or replace the system Python on Linux; parts of the operating system depend on it.

For VS Code, download the `.deb` package (Ubuntu, Debian, Mint) or the `.rpm` package (Fedora, openSUSE) from `https://code.visualstudio.com` and install it with `sudo apt install ./code_*.deb` or `sudo dnf install ./code-*.rpm`. The package also adds Microsoft's repository, so VS Code is then updated with the rest of the system.

1.7 VS Code, a Virtual Environment, NumPy and SciPy

Start VS Code. Click the **Extensions** icon in the Activity Bar on the left (or press Ctrl+Shift+X, on macOS Cmd+Shift+X) and install two extensions, both published by Microsoft:

- **Extension Pack for Java** (identifier `vscjava.vscode-java-pack`). It bundles the Java language support, the debugger, the test runner and a project manager [10]. It finds the JDK you installed by itself; if you have several, the command **Java: Configure Java Runtime** in the Command Palette lets you choose.
- **Python** (identifier `ms-python.python`). It brings **Pylance**, which completes names and marks mistakes while you type, and the **Python Debugger** [11].

Figure 1.2 shows the parts of the window that the book refers to. The Activity Bar (1) switches between Explorer, Search, Source Control, Run and Debug, and Extensions. The Explorer (2) shows the files of the folder you opened. The editor (3) is where you type. The Run button (4), a triangle at the top right of the editor, runs the current file. The integrated terminal (5), opened with **Terminal › New Terminal** or Ctrl+`, is an ordinary PowerShell, zsh or bash window that already sits in your project folder. The Status Bar (6) shows which Python interpreter and which Java runtime VS Code is using; clicking the Python version lets you choose another.

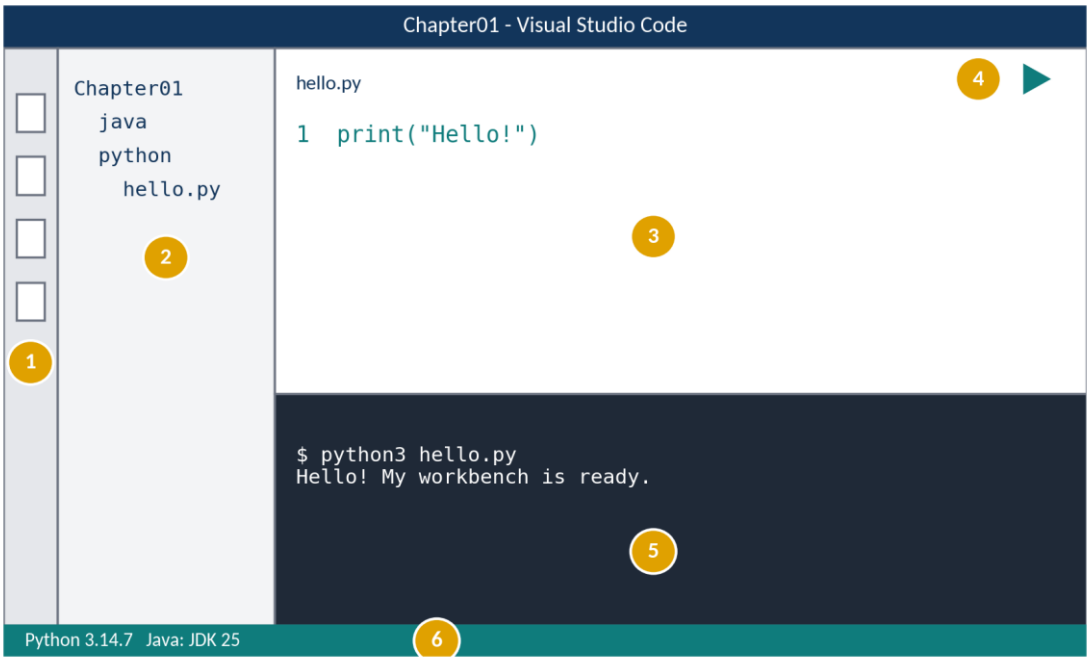


Figure 1.2 The parts of Visual Studio Code used in this book: (1) Activity Bar, (2) Explorer, (3) editor, (4) Run button, (5) integrated terminal, (6) Status Bar.

Always open a *folder* in VS Code (**File › Open Folder**), not a single file. The folder becomes your workspace: the Explorer shows it, the terminal starts in it, and the Java extension treats the `.java` files inside as one project. For this book, open the folder of the chapter you are reading, such as `Chapter01`. Each chapter folder has a `java` folder, a `python` folder, a `data` folder when the programs read data, a `README.md` that repeats every command for Windows, macOS and Linux, and a `requirements.txt`.

Unlike the early chapters of many programming books, this one needs two Python libraries from its first chapter, because every chapter compares our results with NumPy and SciPy. Libraries are installed with `pip`, and they should go into a **virtual environment**: a private folder, by convention `.venv`, with its own copy of `python` and its own place for packages, so that the libraries of one project never collide with those of another [19]. Create one environment for the whole book, in the folder that holds the chapter folders, and install the two libraries into it. On Windows:

```
PS C:\Users\you\MetodeNumerik> python -m venv .venv
PS C:\Users\you\MetodeNumerik> .venv\Scripts\Activate.ps1
(.venv) PS C:\Users\you\MetodeNumerik> python -m pip install numpy scipy
```

On macOS and Linux:

```
$ python3 -m venv .venv
$ source .venv/bin/activate
(.venv) $ python -m pip install numpy scipy
```

The prompt now starts with `(.venv)`, which tells you that the environment is active; inside it, `python` means the environment's Python on every system. If PowerShell refuses to run `Activate.ps1` with a message about execution policies, run `Set-ExecutionPolicy -Scope CurrentUser RemoteSigned` once and try again. In VS Code, press Ctrl+Shift+P, run **Python: Select Interpreter** and choose the interpreter inside `.venv`; from then on the Run button and the integrated terminal use it. Each chapter's `requirements.txt` lists what that chapter needs, so `python -m pip install -r requirements.txt` in a chapter folder is always enough.

1.8 Your First Programs

1.8.1 Hello, in Java

Open the `Chapter01` folder in VS Code, or create it with the subfolders `java` and `python`. Choose **File › New File**, type the program in **Listing 1.1** and save it as `java/Hello.java`. In Java the file name must be the class name followed by `.java`, with the same capital letters.

Listing 1.1 `Hello.java`: the first Java program, which also reports the Java version and the operating system.

```java
/** The first program: say hello and show which Java runs it. */
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello! My workbench for numerical methods is ready.");
        System.out.println("Java version: " + System.getProperty("java.version"));
        System.out.println("Running on:   " + System.getProperty("os.name"));
    }
}
```

`public class Hello` declares a *class*, the unit in which Java keeps code. The comment above it, between `/**` and `*/`, is for people and is ignored by the compiler. `public static void main(String[] args)` is the *main method*, where every Java application starts; `String[] args` receives the words typed after the program's name on the command line. `System.out.println(...)` prints a line, and `System.getProperty("java.version")` asks the running JVM for its version, which is the first thing to check when something behaves strangely. In the terminal of VS Code (Ctrl+`), go into the `java` folder, compile and run:

```
$ cd java
$ javac -d out Hello.java
$ java -cp out Hello
Hello! My workbench for numerical methods is ready.
Java version: 25.0.4
Running on:   Windows 11
```

The option `-d out` puts the compiled `Hello.class` in a folder called `out`, which keeps the source folder tidy, and `-cp out` (*class path*) tells `java` where to find it. On macOS the last line says `Mac OS X`, on Linux `Linux`. The Run button (4 in **Figure 1.2**) does the same in one click.

JDK 25 also allows a much shorter first program. Since JEP 512 became final in JDK 25, a file may contain just a method `void main()` without a class around it, and the helper class `IO` prints a line [20]:

```
void main() {
    IO.println("Hello!");
}
```

The programs in this book nevertheless use the classic form with a class and `public static void main`, because most of them are made of several classes, and because the classic form also runs on JDK 17 and 21.

1.8.2 Hello, in Python

Listing 1.2 is the same program in Python. Save it as `python/hello.py`.

Listing 1.2 `hello.py`: the first Python program.

```python
"""The first program: say hello and show which Python runs it."""
import platform

print("Hello! My workbench for numerical methods is ready.")
print("Python version: " + platform.python_version())
print("Running on:     " + platform.system())
```

The text between the triple quotes at the top is a *docstring*, a description of the file. `import platform` loads a standard module that knows about the computer and the interpreter. There is no class and no `main` method: Python simply runs the file from top to bottom.

```
$ cd ../python
$ python3 hello.py
Hello! My workbench for numerical methods is ready.
Python version: 3.14.7
Running on:     Linux
```

On Windows type `python hello.py`, and the last line says `Windows`; on macOS it says `Darwin`, the name of the core of macOS.

1.8.3 Checking the whole workbench

Rather than trusting that everything went well, we write a program that checks. **Listing 1.3** reports the Python version, whether a virtual environment is active, whether NumPy and SciPy are installed, and whether a Java runtime can be found; **Listing 1.4** does the same from the Java side.

Listing 1.3 `check_setup.py`: checks Python, the virtual environment, NumPy, SciPy and Java.

```
"""Check the workbench: Python, the virtual environment, NumPy, SciPy and Java."""
import importlib
import platform
import shutil
import subprocess
import sys

RECOMMENDED = (3, 14)      # the version this book is written for
MINIMUM = (3, 11)          # the oldest version the book's code runs on


def java_version():
    """Ask the 'java' command for its version; None when it is not found."""
    if shutil.which("java") is None:
        return None
    done = subprocess.run(["java", "-version"], capture_output=True, text=True)
    for line in (done.stderr + done.stdout).splitlines():  # java uses stderr
        if " version " in line and '"' in line:
            return line.split('"')[1]         # openjdk version "25.0.4" ...
    return "unknown"


def library_version(name):
    """The version of an installed library, or None when it is missing."""
    try:
        return importlib.import_module(name).__version__
    except ImportError:
        return None


def main():
    version = sys.version_info[:2]
    in_venv = sys.prefix != sys.base_prefix
    print(f"Python version : {platform.python_version()}")
    print(f"System         : {platform.system()} ({platform.machine()})")
    print(f"Virtual env.   : {'yes' if in_venv else 'no'}")
    for name in ("numpy", "scipy"):
        found = library_version(name)
        print(f"{name:<15}: {found or 'not installed (pip install ' + name + ')'}")
    java = java_version()
    print(f"Java version   : {java or 'NOT FOUND - install JDK 25'}")
    if version < MINIMUM:
        print("Result         : Python is too old, please install Python 3.14")
    elif java is None:
        print("Result         : Python is fine; now install the JDK")
    elif version < RECOMMENDED:
```

Listing 1.3 (Continued) *check_setup.py: checks Python, the virtual environment, NumPy, SciPy and Java.*

```
        print("Result         : works; Python 3.14 is recommended")
    else:
        print("Result         : your workbench is ready")


if __name__ == "__main__":
    main()
```

`sys.version_info[:2]` is the version as a pair of numbers such as `(3, 14)`, and Python compares pairs element by element, so `(3, 12) < (3, 14)` is true. `sys.prefix` and `sys.base_prefix` differ exactly when a virtual environment is active. `importlib.import_module(name)` loads a library by its name and raises `ImportError` when it is missing. `shutil.which("java")` looks for a `java` command on the `PATH`, and `subprocess.run` starts `java -version` as a separate program and captures what it prints; Java prints its version to the *error* stream, a historical quirk, which is why the function reads `done.stderr`.

Listing 1.4 `CheckSetup.java`: checks the Java version and looks for Python.

```java
/** Check the workbench: the Java version, and whether Python can be found. */
public class CheckSetup {
    static final int RECOMMENDED = 25;       // the JDK this book is written for
    static final int MINIMUM = 17;           // the oldest JDK the code runs on

    /** Ask a command for its version; null when the command is not found. */
    static String versionOf(String... command) {
        try {
            Process p = new ProcessBuilder(command).redirectErrorStream(true)
                                                   .start();
            String out = new String(p.getInputStream().readAllBytes()).trim();
            p.waitFor();
            return out.isEmpty() ? null : out.lines().findFirst().get();
        } catch (Exception e) {
            return null;                     // not installed, or not on PATH
        }
    }

    public static void main(String[] args) {
        int feature = Runtime.version().feature();
        System.out.println("Java version    : " + Runtime.version());
        System.out.println("Java vendor     : " + System.getProperty("java.vendor"));
        System.out.println("System          : " + System.getProperty("os.name")
                           + " (" + System.getProperty("os.arch") + ")");
        String python = versionOf("python3", "--version");
        if (python == null) {
            python = versionOf("python", "--version");  // the name on Windows
        }
        System.out.println("Python          : "
                           + (python != null ? python : "NOT FOUND"));
        if (feature < MINIMUM) {
            System.out.println("Result          : Java is too old, install JDK 25");
        } else if (python == null) {
            System.out.println("Result          : Java is fine; now install Python");
        } else if (feature < RECOMMENDED) {
            System.out.println("Result          : works; JDK 25 is recommended");
        } else {
            System.out.println("Result          : your workbench is ready");
        }
    }
}
```

`Runtime.version().feature()` is the main version number of the running Java, 25 for JDK 25. `ProcessBuilder` starts another program; `redirectErrorStream(true)` merges its error stream into its normal output, so the version is found whichever stream Python uses. Because Python on Windows is usually called `python` and on macOS and Linux `python3`, the program tries both. On a finished workbench, with the virtual environment active, the two programs print:

```
(.venv) $ python check_setup.py
Python version : 3.14.7
System         : Linux (x86_64)
Virtual env.   : yes
numpy          : 2.5.3
scipy          : 1.18.1
Java version   : 25.0.4
Result         : your workbench is ready
(.venv) $ java -cp out CheckSetup
Java version   : 25.0.4+8-LTS
Java vendor    : Eclipse Adoptium
System         : Linux (amd64)
Python         : Python 3.14.7
Result         : your workbench is ready
```

The exact build numbers differ from computer to computer. The last line is the one to look at: if it says anything other than “your workbench is ready”, it also says what to install, and the programs still run as long as it says “works”.

1.9 Measuring Error

With the bench ready, we turn to the subject of the book. Before we can make errors small we must be able to measure them, and there are several ways to do so.

1.9.1 True error and relative error

Suppose the true value of a quantity is p and we have an approximation $\tilde{p}$ (read “p tilde”). The **true error** is their difference,

$$E_t = p - \tilde{p} \tag{1.1}$$

and the **absolute error** is its size $|E_t|$. The absolute error alone can mislead. An error of Rp1,000 on a receipt of Rp30,750 is serious; an error of Rp1,000 on the price of a house is nothing. What matters is the error compared with the size of the thing measured, the **relative error**, usually given in percent:

$$\varepsilon_t = \frac{p - \tilde{p}}{p} \times 100\% \tag{1.2}$$

Here is a worked example. The number $\pi = 3.14159265\ldots$ is often approximated by $22/7 = 3.14285714\ldots$ and by $355/113 = 3.14159292\ldots$. For $22/7$, $E_t = 3.14159265 - 3.14285714 = -0.00126449$, so the absolute error is 0.00126 and the relative error is $0.00126449/3.14159265 \times 100\% = 0.0402\%$. For $355/113$, $E_t = -0.00000027$ and $\varepsilon_t = -0.0000085\%$, almost five thousand times smaller. Chinese astronomers knew $355/113$ fifteen centuries ago.

A related idea is the number of **significant digits**: the digits of $\tilde{p}$ that can be trusted. $\tilde{p}$ approximates p to n significant digits when its relative error is below 5×10^{-n}, that is, below $0.5 \times 10^{2-n}$ percent [3]. $22/7$ has a relative error of 4.0×10^{-4}, below 5×10^{-4}, so it gives π to 4 significant digits (3.142 against 3.14159); $355/113$ gives 7.

1.9.2 Approximate error and the stopping rule

The trouble with Equation (1.2) is that it needs the true value p, and if we knew p we would not be computing it. Most numerical methods, however, are *iterative*: they produce a sequence of approximations $\tilde{p}_1, \tilde{p}_2, \ldots$ that get better and better. Then the change from one approximation to the next is a good measure of how much we are still learning. The **approximate relative error** compares the current approximation with the previous one [2]:

$$\varepsilon_a = \frac{\tilde{p}_{\text{new}} - \tilde{p}_{\text{old}}}{\tilde{p}_{\text{new}}} \times 100\% \tag{1.3}$$

We stop when $|\varepsilon_a|$ falls below a tolerance ε_s that we choose in advance. If we want n correct significant digits, a convenient choice, known as the Scarborough criterion, is

$$\varepsilon_s = 0.5 \times 10^{2-n}\ \% \tag{1.4}$$

so $n = 3$ digits gives $\varepsilon_s = 0.05\%$ and $n = 6$ gives 0.00005%. The criterion is not a guarantee, because a slowly converging sequence can change little from step to step while still being far from its limit (Chapter 2 meets exactly this with regula falsi). But for the methods that converge well, it is a reliable rule, and when the approximate error is below ε_s, the result is usually correct to at least n significant digits [2].

1.9.3 Two kinds of error

Where do errors come from? Leaving aside mistakes (a wrong formula, a misread scale) and uncertain data (Bu Wati's scale shows only one decimal), two kinds of error belong to numerical computing itself, and this book meets them in every chapter.

- **Round-off error** comes from the machine. A computer stores every number with a fixed number of binary digits, so most numbers, such as 2.3 and 0.1, are stored slightly wrong, and every arithmetic operation rounds its result again. Section 1.10 shows exactly how.
- **Truncation error** comes from the method. When we replace an infinite process by a finite one, such as summing only the first terms of an infinite series, or replacing a derivative by a difference over a small step, we cut something off. That would happen even on a perfect machine with infinitely many digits.

Section 1.12 puts the two side by side and shows that fighting one of them often feeds the other.

1.10 How the Machine Stores a Number

1.10.1 Binary fractions

A computer stores numbers in binary, with the digits 0 and 1. Whole numbers are easy: $13 = 8 + 4 + 1 = 1101_2$. Fractions work the same way with negative powers of two: $0.625 = \frac{1}{2} + \frac{1}{8} = 0.101_2$. The trouble is that most decimal fractions have no finite binary form, exactly as $1/3 = 0.333\ldots$ has no finite decimal form. To find the binary digits of 0.1, double it again and again and write down the whole part each time:

$0.1 \times 2 = 0.2$ (digit 0), $0.2 \times 2 = 0.4$ (0), $0.4 \times 2 = 0.8$ (0), $0.8 \times 2 = 1.6$ (1, keep 0.6), $0.6 \times 2 = 1.2$ (1, keep 0.2), and now 0.2 comes back, so the digits repeat forever:

$$0.1 = 0.0\,0011\,0011\,0011\ldots_2 \tag{1.5}$$

A machine must stop somewhere, and wherever it stops, the stored number is not 0.1. The same holds for 0.2, 0.3, 4.1 and nearly every number with a decimal point that people type.

1.10.2 The IEEE 754 double

Since 1985, practically every computer stores real numbers according to the standard IEEE 754, revised most recently in 2019 [21]. Java's `double` and Python's `float` are both the standard's **binary64** format, called *double precision*. A binary64 number uses 64 bits, split into three fields, as **Figure 1.3** shows for the number 0.1: one sign bit s, eleven exponent bits holding a whole number e between 0 and 2047, and 52 fraction bits $f_1 f_2 \ldots f_{52}$. For the ordinary (*normal*) numbers, the value is

$$x = (-1)^s \times (1.f_1 f_2 \ldots f_{52})_2 \times 2^{e-1023} \tag{1.6}$$

The leading 1 before the binary point is not stored, because every normal number has it; this *hidden bit* gives 53 significant binary digits for the price of 52. The exponent is stored with a *bias* of 1023, so that $e = 1023$ means 2^0, and small numbers get exponents below 1023. The extreme values $e = 0$ and $e = 2047$ are reserved for zero, the very small *subnormal* numbers, infinity and NaN ("not a number", the result of $0/0$).

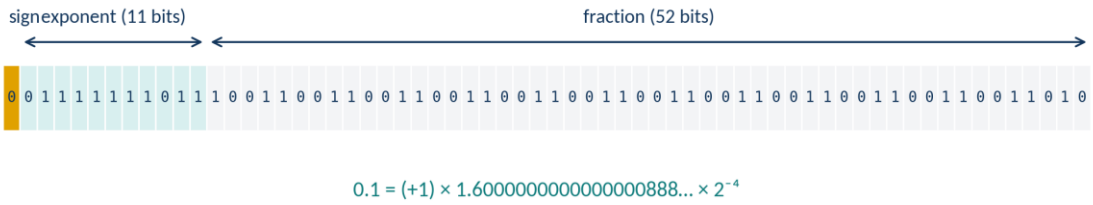


Figure 1.3 The 64 bits of the double nearest to 0.1: sign 0, exponent 01111111011 (1019, meaning 2^{-4}) and 52 fraction bits in which the pattern 1001 repeats until it is cut off and rounded.

Let us decode **Figure 1.3** by hand. The sign bit is 0, so the number is positive. The exponent bits 01111111011_2 are 1019, and $1019 - 1023 = -4$. The fraction starts 1001 1001 1001 ..., so the significand is $1.1001\,1001\,\ldots_2 = 1.6000\ldots$, and $1.6 \times 2^{-4} = 0.1$. The last bits are ...1010 instead of the ...1001 that the pattern would continue with, because the infinite tail 1001 1001 ... was rounded *up*. That is why the stored value is slightly *more* than 0.1, as the first program of Section 1.11 will print in full.

1.10.3 Machine epsilon and the rounding bound

How far apart are two neighboring doubles? Between 1 and 2 the exponent is fixed and the 52 fraction bits count in steps of 2^{-52}. This gap between 1 and the next larger double is the **machine epsilon**,

$$\varepsilon_{\text{mach}} = 2^{-52} \approx 2.22 \times 10^{-16} \tag{1.7}$$

Between 2 and 4 the gap is $2\varepsilon_{\text{mach}}$, between 1024 and 2048 it is $1024\,\varepsilon_{\text{mach}}$: the doubles are spread like a ruler whose marks get farther apart as the numbers grow, but always with the same *relative* spacing. When a real number x is stored, it is rounded to the nearest double $\text{fl}(x)$, which is at most half a gap away. In relative terms,

$$\text{fl}(x) = x\,(1+\delta), \qquad |\delta| \le u = \frac{1}{2}\varepsilon_{\text{mach}} = 2^{-53} \approx 1.11 \times 10^{-16} \tag{1.8}$$

The number u is the **unit round-off**. IEEE 754 also requires that each of the operations $+, -, \times, \div$ and the square root gives the exactly rounded result: the computer behaves as if it computed the exact answer and then rounded it once [21]. So every single operation has a relative error of at most u, about 16 significant decimal digits. Equation (1.8) is the most important fact in this chapter, and the error analyses of the whole book start from it [22].

The standard defines other formats too. **Table 1.2** compares the two that Java and Python offer directly; Java's `float` is binary32, and in Python it is available through NumPy's `float32`.

Table 1.2 The two IEEE 754 formats used in this book.

	Binary32 (`float`)	Binary64 (`double`)
Bits: sign, exponent, fraction	1, 8, 23	1, 11, 52
Significant binary digits	24	53
Machine epsilon	$2^{-23} \approx 1.19 \times 10^{-7}$	$2^{-52} \approx 2.22 \times 10^{-16}$
Decimal digits, about	7	16
Largest number, about	3.4×10^{38}	1.8×10^{308}
Whole numbers stored exactly up to	$2^{24} = 16{,}777{,}216$	$2^{53} \approx 9.0 \times 10^{15}$

The last row explains the lost Rp826 of Section 1.1. A `float` holds whole numbers exactly only up to about 16.8 million. Bu Wati's year of receipts adds up to more than 700 million rupiah, where the gap between neighboring floats is 64 rupiah: the tablet program could not even *write down* the correct total.

1.11 Five Experiments with FloatLab

Theory is convincing when the machine confirms it. **Listing 1.5** runs five small experiments; **Listing 1.6** is the same program in Java.

Listing 1.5 `float_lab.py`: five experiments with double-precision numbers.

```python
"""Five small experiments that show how a computer stores real numbers."""
from decimal import Decimal
import math


def main():
    print("1. Adding 0.1 and 0.2")
    s = 0.1 + 0.2
    print(f"   0.1 + 0.2        = {s!r}")
    print(f"   0.1 + 0.2 == 0.3 : {s == 0.3}")

    print("2. What is really stored for 0.1")
    print(f"   {Decimal(0.1)}")        # the exact value of the double

    print("3. Machine epsilon, found by halving")
    eps = 1.0
    while 1.0 + eps / 2 > 1.0:     # stop when half of eps is no longer seen
        eps /= 2
    print(f"   eps              = {eps:.6e}")
    print(f"   1 + eps/2 == 1   : {1.0 + eps / 2 == 1.0}")

    print("4. Adding 0.1 again and again")
    for n in (10, 1000, 1000000):
        total = 0.0
        for _ in range(n):
            total += 0.1
        exact = n / 10
        print(f"   {n:>7} times: {total!r:<22} error {abs(total - exact):.3e}")

    print("5. Subtracting nearly equal numbers: x^2 - 1e8 x + 1 = 0")
    a, b, c = 1.0, -1.0e8, 1.0
    root = math.sqrt(b * b - 4 * a * c)
    naive = (-b - root) / (2 * a)          # small root, textbook formula
    big = (-b + root) / (2 * a)            # large root: no cancellation
    stable = c / (a * big)                 # small root from x1 * x2 = c / a
    print(f"   textbook formula : {naive:.10e}")
    print(f"   stable formula   : {stable:.10e}")
    print(f"   relative error of the textbook formula: "
          f"{abs(naive - stable) / stable:.3e}")


if __name__ == "__main__":
    main()
```

Listing 1.6 `FloatLab.java`: the same experiments in Java.

```
import java.math.BigDecimal;
import java.util.Locale;

/** Five small experiments that show how a computer stores real numbers. */
public class FloatLab {
    public static void main(String[] args) {
        Locale.setDefault(Locale.ROOT);         // a dot as decimal point
        System.out.println("1. Adding 0.1 and 0.2");
        double s = 0.1 + 0.2;
        System.out.println("   0.1 + 0.2        = " + s);
        System.out.println("   0.1 + 0.2 == 0.3 : "
                           + (s == 0.3 ? "True" : "False"));

        System.out.println("2. What is really stored for 0.1");
        System.out.println("   " + new BigDecimal(0.1));  // exact value

        System.out.println("3. Machine epsilon, found by halving");
        double eps = 1.0;
        while (1.0 + eps / 2 > 1.0) {   // stop when half of eps is not seen
            eps /= 2;
        }
        System.out.printf("   eps              = %.6e%n", eps);
        System.out.println("   1 + eps/2 == 1   : "
                           + (1.0 + eps / 2 == 1.0 ? "True" : "False"));

        System.out.println("4. Adding 0.1 again and again");
        for (int n : new int[] {10, 1000, 1000000}) {
            double total = 0.0;
            for (int i = 0; i < n; i++) {
                total += 0.1;
            }
            double exact = n / 10.0;
            System.out.printf("   %7d times: %-22s error %.3e%n", n,
                              Double.toString(total), Math.abs(total - exact));
        }
```

Listing 1.6 (Continued) *FloatLab.java: the same experiments in Java.*

```
        System.out.println("5. Subtracting nearly equal numbers: "
                           + "x^2 - 1e8 x + 1 = 0");
        double a = 1.0, b = -1.0e8, c = 1.0;
        double root = Math.sqrt(b * b - 4 * a * c);
        double naive = (-b - root) / (2 * a);    // small root, textbook formula
        double big = (-b + root) / (2 * a);      // large root: no cancellation
        double stable = c / (a * big);           // small root from x1 x2 = c/a
        System.out.printf("   textbook formula : %.10e%n", naive);
        System.out.printf("   stable formula   : %.10e%n", stable);
        System.out.printf("   relative error of the textbook formula: %.3e%n",
                          Math.abs(naive - stable) / stable);
    }
}
```

Two details of the Java version matter. `Locale.setDefault(Locale.ROOT)` makes `printf` write a decimal point instead of the decimal comma that a computer set to Indonesian would use; every Java program of the book starts with this line, so that its output is the same everywhere. And `new BigDecimal(0.1)` converts the double *exactly* into a decimal number with as many digits as needed, which is what Python's `Decimal(0.1)` does too. Run either version:

```
$ python3 float_lab.py          (or: java -cp out FloatLab)
```

```
1. Adding 0.1 and 0.2
   0.1 + 0.2        = 0.30000000000000004
   0.1 + 0.2 == 0.3 : False
2. What is really stored for 0.1
   0.1000000000000000055511151231257827021181583404541015625
3. Machine epsilon, found by halving
   eps              = 2.220446e-16
   1 + eps/2 == 1   : True
4. Adding 0.1 again and again
        10 times: 0.9999999999999999     error 1.110e-16
      1000 times: 99.9999999999986       error 1.407e-12
   1000000 times: 100000.00000133288     error 1.333e-06
5. Subtracting nearly equal numbers: x^2 - 1e8 x + 1 = 0
   textbook formula : 7.4505805969e-09
   stable formula   : 1.0000000000e-08
   relative error of the textbook formula: 2.549e-01
```

Both programs print exactly these lines. Let us read them one by one.

Experiment 1. $0.1 + 0.2$ is `0.30000000000000004`, and it is not equal to `0.3`. Both 0.1 and 0.2 are stored slightly too large, the sum is rounded once more, and the result lands on the double just above the double nearest to 0.3. Neither answer is "wrong" by IEEE 754; they are simply different doubles. The lesson is that `==` between computed doubles is almost always a mistake. Compare with a tolerance instead: $|a - b| \le \tau \max(|a|, |b|)$ for a relative tolerance τ such as 10^{-12}.

Experiment 2. The double nearest to 0.1 is exactly `0.1000000000000000055511151231257827021181583404541015625`. The error is 5.55×10^{-18}, a relative error of 5.55×10^{-17}, below the bound $u = 1.11 \times 10^{-16}$ of Equation (1.8), as it must be.

Experiment 3. Halving a number `eps` until $1 + \text{eps}/2$ is no longer larger than 1 finds 2.220446×10^{-16}, the machine epsilon of Equation (1.7). The program discovers a property of the hardware by experiment.

Experiment 4. Adding 0.1 ten times gives `0.9999999999999999`, one gap below 1. After a thousand additions the error is 1.4×10^{-12} and after a million it is 1.3×10^{-6}. Every addition rounds, and the rounding errors do not cancel: the error grows with the number of operations. This is the Patriot clock of Section 1.1 in miniature. A careful bound: n additions of numbers of size up to S can make an error up to about $n\,u\,S$; for $n = 10^6$ and a running sum up to 10^5, that is $10^6 \times 1.1 \times 10^{-16} \times 10^5 \approx 10^{-5}$, and the measured error is well inside it [22].

Experiment 5 is the most important. The equation $x^2 - 10^8 x + 1 = 0$ has two roots, one near 10^8 and one near 10^{-8}. The school formula for the small root is

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{1.9}$$

with $b = -10^8$, so it computes $10^8 - \sqrt{10^{16} - 4}$: the difference of two numbers that agree in their first 16 digits. The square root is correct to about 16 digits, its error is about $10^8 \times 1.1 \times 10^{-16} \approx 10^{-8}$, and that error is as large as the answer. The subtraction itself is exact; it merely reveals the garbage that the rounding had put in the last digits. This is called **catastrophic cancellation**, and the result `7.45e-09` has a relative error of 25%. The cure is to avoid the subtraction. The large root x_1 is computed by adding numbers of the same sign, which is harmless, and the product of the two roots of $ax^2 + bx + c$ is c/a (Vieta's formula), so

$$x_2 = \frac{c}{a\,x_1} \tag{1.10}$$

needs only a multiplication and a division, each accurate to u. The stable formula gives `1.0000000000e-08`, correct to all printed digits. Equation (1.9) and Equation (1.10) are equal in algebra and very different on a computer. That is the meaning of *numerical stability*.

1.12 Round-off and Truncation Together

1.12.1 Taylor series and truncation

Many functions are computed from their **Taylor series**. If a function f has derivatives of all orders near a point a, then for x near a,

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n + R_n \tag{1.11}$$

where the remainder $R_n = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-a)^{n+1}$ for some ξ between a and x [3]. Keeping the terms up to $(x-a)^n$ and dropping R_n is a *truncation*, and R_n is the truncation error. For $f(x) = e^x$ at $a = 0$, every derivative is $e^0 = 1$, so

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \tag{1.12}$$

Let us compute $e^{0.5}$ to three significant digits by hand, stopping with Equation (1.4): $\varepsilon_s = 0.5 \times 10^{2-3} = 0.05\%$. The true value is $e^{0.5} = 1.648721\ldots$ One term gives 1. Two terms give $1 + 0.5 = 1.5$, with $\varepsilon_a = (1.5-1)/1.5 = 33.3\%$. Three terms add $0.5^2/2 = 0.125$: 1.625, $\varepsilon_a = 0.125/1.625 = 7.69\%$. Four terms add $0.5^3/6 = 0.0208333$: 1.6458333, $\varepsilon_a = 1.27\%$. Five terms add $0.5^4/24 = 0.0026042$: 1.6484375, $\varepsilon_a = 0.158\%$. Six terms add $0.5^5/120 = 0.00026042$: 1.6486979, $\varepsilon_a = 0.0158\%$, below 0.05%, so we stop. The true relative error is $(1.648721 - 1.648698)/1.648721 = 0.0014\%$: better than promised. **Listing 1.7** and **Listing 1.8** do the same and also try something harder.

Listing 1.7 `exp_series.py`: e^x from its Taylor series with the stopping rule of Equation (1.4).

```
"""e^x from its Taylor series, stopped by the tolerance for n significant digits."""
import math


def exp_series(x, digits, show=False):
    """Sum 1 + x + x^2/2! + ... until the approximate relative error is small."""
    tol = 0.5 * 10 ** (2 - digits)          # percent (Scarborough criterion)
    total, term, k = 1.0, 1.0, 0
    if show:
        print(f"{'terms':>5} {'sum':>14} {'true err %':>12} {'approx err %':>13}")
        true = math.exp(x)
        print(f"{1:>5} {total:>14.9f} {abs(true - total) / true * 100:>12.6f}")
    while True:
        k += 1
        term *= x / k                        # x^k / k! from the previous term
        old, total = total, total + term
        approx = abs((total - old) / total) * 100
        if show:
            err = abs(true - total) / true * 100
            print(f"{k + 1:>5} {total:>14.9f} {err:>12.6f} {approx:>13.6f}")
        if approx < tol:
            return total, k + 1


def main():
    print("e^0.5 to 3 significant digits (tolerance 0.05 %):")
    exp_series(0.5, 3, show=True)
    print()
    print("e^-20 in three ways:")
    direct, n = exp_series(-20.0, 15)
    inverse, _ = exp_series(20.0, 15)
    print(f"  series at x = -20 ({n} terms): {direct:.10e}")
    print(f"  1 / series at x = 20         : {1 / inverse:.10e}")
    print(f"  library exp(-20)             : {math.exp(-20.0):.10e}")


if __name__ == "__main__":
    main()
```

Listing 1.8 `ExpSeries.java`: the same series in Java.

```
import java.util.Locale;

/** e^x from its Taylor series, stopped by the tolerance for n digits. */
public class ExpSeries {
    static int terms;                       // terms used by the last call

    /** Sum 1 + x + x^2/2! + ... until the approximate error is small. */
    static double expSeries(double x, int digits, boolean show) {
        double tol = 0.5 * Math.pow(10, 2 - digits);  // percent (Scarborough)
        double total = 1.0, term = 1.0, t = Math.exp(x);
        int k = 0;
        if (show) {
            System.out.printf("%5s %14s %12s %13s%n", "terms", "sum",
                              "true err %", "approx err %");
            System.out.printf("%5d %14.9f %12.6f%n", 1, total,
                              Math.abs(t - total) / t * 100);
        }
        while (true) {
            k++;
            term *= x / k;                  // x^k / k! from the previous term
            double old = total;
            total = total + term;
            double approx = Math.abs((total - old) / total) * 100;
            if (show) {
                double err = Math.abs(t - total) / t * 100;
                System.out.printf("%5d %14.9f %12.6f %13.6f%n", k + 1, total,
                                  err, approx);
            }
            if (approx < tol) {
                terms = k + 1;
                return total;
            }
        }
    }

    public static void main(String[] args) {
        Locale.setDefault(Locale.ROOT);
        System.out.println("e^0.5 to 3 significant digits (tolerance 0.05 %):");
        expSeries(0.5, 3, true);
        System.out.println();
        System.out.println("e^-20 in three ways:");
        double direct = expSeries(-20.0, 15, false);
        int n = terms;
        double inverse = expSeries(20.0, 15, false);
        System.out.printf("  series at x = -20 (%d terms): %.10e%n", n, direct);
        System.out.printf("  1 / series at x = 20         : %.10e%n", 1 / inverse);
```

Listing 1.8 (Continued) *ExpSeries.java: the same series in Java.*

```
        System.out.printf("  library exp(-20)             : %.10e%n",
                          Math.exp(-20.0));
    }
}
```

The loop computes each term from the previous one, $x^k/k! = \frac{x^{k-1}}{(k-1)!} \cdot \frac{x}{k}$, instead of computing powers and factorials separately, which would be slow and, for large k, would overflow. The output:

```
e^0.5 to 3 significant digits (tolerance 0.05 %):
terms            sum   true err %  approx err %
    1    1.000000000    39.346934
    2    1.500000000     9.020401     33.333333
    3    1.625000000     1.438768      7.692308
    4    1.645833333     0.175162      1.265823
    5    1.648437500     0.017212      0.157978
    6    1.648697917     0.001416      0.015795

e^-20 in three ways:
  series at x = -20 (95 terms): 6.1475618289e-09
  1 / series at x = 20        : 2.0611536224e-09
  library exp(-20)            : 2.0611536224e-09
```

The table agrees with our hand computation line by line, and it is the classic example of Chapra and Canale [2]. The second part is a warning. At $x = -20$ the series has terms as large as $20^{20}/20! \approx 4.3 \times 10^{7}$ with alternating signs, which must cancel down to $e^{-20} \approx 2.06 \times 10^{-9}$. Each large term carries a rounding error of about $4.3 \times 10^{7} \times 1.1 \times 10^{-16} \approx 5 \times 10^{-9}$, larger than the answer, and the sum `6.1e-09` is wrong by a factor of three although no term was truncated too early. The stable way is to sum the series at $+20$, where all terms are positive and nothing cancels, and take the reciprocal; it agrees with the library to all ten digits. Truncation was not the problem here; round-off, magnified by cancellation, was.

1.12.2 The step-size dilemma

The derivative $f'(x)$ is the slope of f at x. Taylor's formula Equation (1.11) with $a = x$ and $x + h$ in place of x gives $f(x+h) = f(x) + f'(x)h + \frac{f''(\xi)}{2}h^2$, and solving for $f'(x)$,

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{f''(\xi)}{2}\,h \tag{1.13}$$

The fraction is the **forward difference**, and the last term is its truncation error, proportional to h: halve the step and the truncation error halves. So a tiny h should give a perfect derivative. The program of **Listing 1.9** tries it for $f(x) = \sqrt{x}$ at $x = 2$, where the exact derivative is $1/(2\sqrt{2}) = 0.35355339\ldots$, with $h = 10^{-1}, 10^{-2}, \ldots, 10^{-16}$.

Listing 1.9 `diff_step.py`: the forward difference of $\sqrt{x}$ with ever smaller steps.

```python
"""Truncation against round-off: the forward difference of sqrt(x) at x = 2."""
import math


def main():
    x = 2.0
    true = 0.5 / math.sqrt(x)              # the exact derivative 1 / (2 sqrt x)
    print(f"{'h':>8} {'estimate':>20} {'error':>11}")
    for p in range(1, 17):
        h = 1.0 / 10 ** p                  # 10^p is exact for p <= 22
        estimate = (math.sqrt(x + h) - math.sqrt(x)) / h
        print(f"{h:>8.0e} {estimate:>20.15f} {abs(estimate - true):>11.3e}")


if __name__ == "__main__":
    main()
```

Listing 1.10 `DiffStep.java`: the same experiment in Java.

```java
import java.util.Locale;

/** Truncation against round-off: the forward difference of sqrt(x) at x = 2. */
public class DiffStep {
    public static void main(String[] args) {
        Locale.setDefault(Locale.ROOT);
        double x = 2.0;
        double exact = 0.5 / Math.sqrt(x);   // the derivative 1 / (2 sqrt x)
        System.out.printf("%8s %20s %11s%n", "h", "estimate", "error");
        for (int p = 1; p <= 16; p++) {
            double h = 1.0 / Math.pow(10, p);    // 10^p is exact for p <= 22
            double estimate = (Math.sqrt(x + h) - Math.sqrt(x)) / h;
            System.out.printf("%8.0e %20.15f %11.3e%n", h, estimate,
                              Math.abs(estimate - exact));
        }
    }
}
```

In both **Listing 1.9** and **Listing 1.10**, the step is computed as `1.0 / 10 ** p` in Python and `1.0 / Math.pow(10, p)` in Java rather than as 10^{-p} directly: 10^{p} is a whole number that a double holds exactly (up to 10^{22}), so the division is rounded once in both languages and the two programs use exactly the same steps.

```
       h             estimate       error
   1e-01    0.349241122458488   4.312e-03
   1e-02    0.353112550268730   4.408e-04
   1e-03    0.353509207464464   4.418e-05
   1e-04    0.353548971285811   4.419e-06
   1e-05    0.353552948650915   4.419e-07
   1e-06    0.353553346377211   4.422e-08
   1e-07    0.353553384346839   6.246e-09
   1e-08    0.353553386567285   4.026e-09
   1e-09    0.353553408771745   1.818e-08
   1e-10    0.353552742637930   6.480e-07
   1e-11    0.353539419961635   1.397e-05
   1e-12    0.353495011040650   5.838e-05
   1e-13    0.353050921830800   5.025e-04
   1e-14    0.355271367880050   1.718e-03
   1e-15    0.222044604925031   1.315e-01
   1e-16    0.000000000000000   3.536e-01
```

Down to $h = 10^{-7}$ the error falls by ten with each step, as Equation (1.13) promises. Then it turns around, grows, and at $h = 10^{-16}$ the estimate is exactly 0: $2 + 10^{-16}$ rounds to 2, because 10^{-16} is less than half the gap between doubles near 2. **Figure 1.4** shows the V-shaped curve on logarithmic axes.

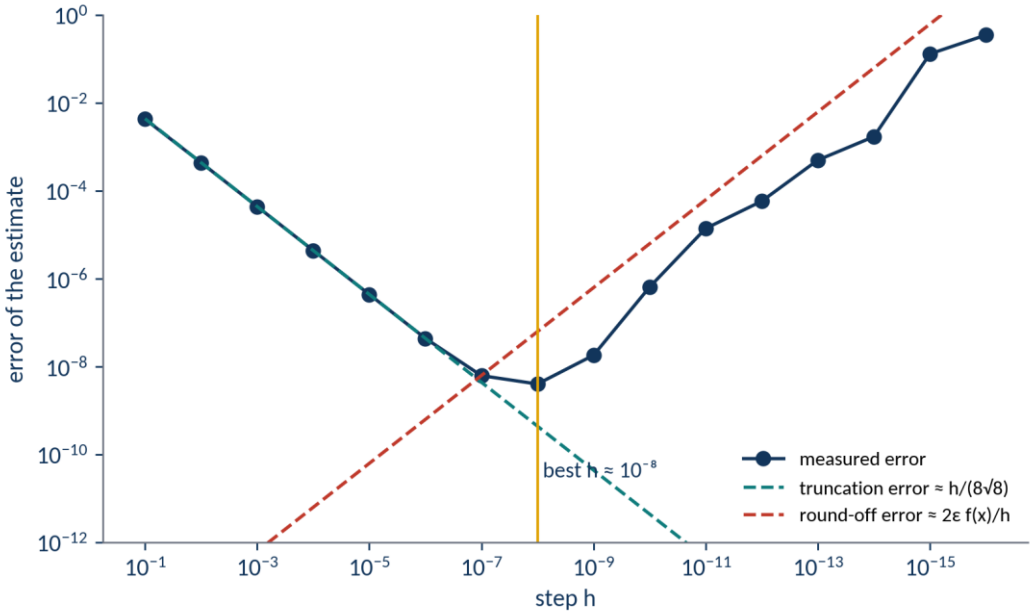


Figure 1.4 The error of the forward difference of $\sqrt{x}$ at $x = 2$ against the step h: truncation error falls with h, round-off error grows as $1/h$, and the total is smallest near $h = 10^{-8}$.

The explanation combines the two kinds of error. Each computed value of f carries a rounding error of about $\varepsilon_{\text{mach}}|f(x)|$ (counting the rounding of $x + h$ too), so their difference may be wrong by up to $2\varepsilon_{\text{mach}}|f(x)|$, and dividing by h multiplies that by $1/h$. With $M = |f''(x)|$, the total error is about

$$E(h) \approx \frac{M}{2}\,h + \frac{2\,\varepsilon_{\text{mach}}\,|f(x)|}{h} \tag{1.14}$$

The first term falls and the second grows as h shrinks. Setting the derivative $E'(h) = M/2 - 2\varepsilon_{\text{mach}}|f|/h^2$ to zero gives the best step,

$$h_{\text{opt}} = 2\sqrt{\frac{\varepsilon_{\text{mach}}\,|f(x)|}{M}} \tag{1.15}$$

For $\sqrt{x}$ at 2, $f = \sqrt{2} = 1.414$ and $M = \frac{1}{4}x^{-3/2} = 0.0884$, so $h_{\text{opt}} = 2\sqrt{2.22 \times 10^{-16} \times 1.414/0.0884} \approx 1.2 \times 10^{-7}$, and the smallest error is about $Mh_{\text{opt}} \approx 10^{-8}$. The measurement agrees in size: the best steps are 10^{-7} and 10^{-8}, with errors of 6 and 4 billionths. Equation (1.14) is a bound, and real rounding errors partly cancel, which is why the measured curve lies below the red line. The practical rule is worth remembering: for a forward difference in double precision, take h near $\sqrt{\varepsilon_{\text{mach}}} \approx 10^{-8}$ times the size of x, never "as small as possible". Chapter 9 builds better difference formulas whose truncation error falls like h^2 or h^4 and whose best step is therefore larger.

1.13 The Laundry Ledger: Adding Up Money Safely

Back to Bu Wati. We can now simulate her year and see exactly where the tablet's Rp826 went. **Listing 1.11** and its Java twin **Listing 1.12** generate 365 days of 60 orders each, with weights from 1.0 kg to 8.0 kg in steps of 100 g, and add the prices up in four ways.

Listing 1.11 `laundry_ledger.py`: a year of laundry receipts added up as integers, as doubles, as floats, and as floats with Kahan's compensation.

```
"""A year of laundry receipts added up in four ways.

The receipts are generated by a small random-number generator that gives the
same numbers in Java and Python, so both programs print the same report."""

PRICE_PER_KG = 7500           # rupiah per kilogram
DAYS = 365
ORDERS_PER_DAY = 60


def lcg(seed):
    """A linear congruential generator: the same sequence in every language."""
    while True:
        seed = (1103515245 * seed + 12345) % 2147483648
        yield seed


def to_float32(x):
    """Round a Python float (a double) to the nearest single-precision value."""
    import struct
    return struct.unpack("f", struct.pack("f", x))[0]


def main():
    rnd = lcg(2026)
    exact = 0                  # whole rupiah, as an integer: no error at all
    naive64 = 0.0              # double precision, simple running sum
    naive32 = 0.0              # single precision, simple running sum
    kahan32, carry = 0.0, 0.0  # single precision with Kahan's compensation
    for _ in range(DAYS * ORDERS_PER_DAY):
        grams = 1000 + (next(rnd) % 71) * 100     # 1.0 kg to 8.0 kg
        rupiah = grams * PRICE_PER_KG // 1000     # exact: grams is 100s
        kg = grams / 1000.0
        exact += rupiah
        naive64 += kg * PRICE_PER_KG
        naive32 = to_float32(naive32 + to_float32(kg * PRICE_PER_KG))
        y = to_float32(to_float32(kg * PRICE_PER_KG) - carry)
        t = to_float32(kahan32 + y)
        carry = to_float32(to_float32(t - kahan32) - y)   # what was lost
        kahan32 = t
    print(f"Orders in the year     : {DAYS * ORDERS_PER_DAY:,}")
    print(f"Exact total (integer)  : Rp {exact:,}")
    for name, value in (("double, simple sum", naive64),
                        ("float, simple sum", naive32),
                        ("float, Kahan sum", kahan32)):
        print(f"{name:<23}: Rp {value:,.2f}  (off by {value - exact:+,.2f})")
```

Listing 1.11 (Continued) *laundry_ledger.py: a year of laundry receipts added up as integers, as doubles, as floats, and as floats with Kahan's compensation.*

```


if __name__ == "__main__":
    main()
```

Listing 1.12 `LaundryLedger.java`: the same ledger in Java.

```
import java.util.Locale;

/**
 * A year of laundry receipts added up in four ways. The receipts come from a
 * small random-number generator that gives the same numbers in Java and
 * Python, so both programs print the same report.
 */
public class LaundryLedger {
    static final long PRICE_PER_KG = 7500;    // rupiah per kilogram
    static final int DAYS = 365;
    static final int ORDERS_PER_DAY = 60;
    static long seed = 2026;

    /** A linear congruential generator: the same sequence in every language. */
    static long next() {
        seed = (1103515245L * seed + 12345L) % 2147483648L;
        return seed;
    }

    public static void main(String[] args) {
        Locale.setDefault(Locale.ROOT);
        long exact = 0;             // whole rupiah, as an integer: no error
        double naive64 = 0.0;       // double precision, simple running sum
        float naive32 = 0.0f;       // single precision, simple running sum
        float kahan32 = 0.0f;       // single precision with Kahan's method
        float carry = 0.0f;
        for (int i = 0; i < DAYS * ORDERS_PER_DAY; i++) {
            long grams = 1000 + (next() % 71) * 100;     // 1.0 kg to 8.0 kg
            long rupiah = grams * PRICE_PER_KG / 1000;   // exact: 100s of g
            double kg = grams / 1000.0;
            exact += rupiah;
            naive64 += kg * PRICE_PER_KG;
            naive32 = naive32 + (float) (kg * PRICE_PER_KG);
            float y = (float) (kg * PRICE_PER_KG) - carry;
            float t = kahan32 + y;
            carry = (t - kahan32) - y;                   // what was lost
            kahan32 = t;
        }
        System.out.printf("Orders in the year     : %,d%n", DAYS * ORDERS_PER_DAY);
        System.out.printf("Exact total (integer)  : Rp %,d%n", exact);
        String[] names = {"double, simple sum", "float, simple sum",
                          "float, Kahan sum"};
        double[] values = {naive64, naive32, kahan32};
        for (int k = 0; k < names.length; k++) {
            System.out.printf("%-23s: Rp %,.2f  (off by %+,.2f)%n", names[k],
                              values[k], values[k] - exact);
```

Listing 1.12 (Continued) *LaundryLedger.java: the same ledger in Java.*

```
        }
    }
}
```

The receipts come from a **linear congruential generator**, the rule $s_{k+1} = (1103515245\, s_k + 12345) \bmod 2^{31}$, instead of the language's own random numbers, because Java's `Random` and Python's `random` produce different sequences and we want both programs to see the same year. Python has no single-precision type in its standard library, so `to_float32` rounds a double to the nearest float by packing it into four bytes with `struct` and unpacking it again. In Java, `(float)` does the same. Because IEEE 754 rounds every operation exactly, a float sum computed in double and then rounded to float is the same as the float sum computed directly, and the two programs agree:

```
Orders in the year     : 21,900
Exact total (integer)  : Rp 739,304,250
double, simple sum     : Rp 739,304,250.00  (off by +0.00)
float, simple sum      : Rp 739,303,424.00  (off by -826.00)
float, Kahan sum       : Rp 739,304,256.00  (off by +6.00)
```

The integer total is exact by construction. A weight in grams is a whole number, grams × 7500/1000 is a whole number of rupiah because the grams are multiples of 100, and whole numbers add without any error until they exceed 2^{63}, about 9.2×10^{18} rupiah.

The double total is correct to the printed sen, although some single prices, such as 4.1×7500, were slightly off. With 16 significant digits, the double has plenty of room for a total of 7.4×10^{8} rupiah; the accumulated error is far below one sen.

The float total is Rp826 short. Near 7×10^{8} neighboring floats are 64 apart (**Table 1.2**), so each addition of a receipt of about Rp30,000 is rounded to a multiple of 64, losing up to 32 rupiah each time. Over 21,900 additions these losses partly cancel and partly pile up.

The Kahan total is also computed in float, yet it is off by only Rp6, and Rp6 is the best a float can do: 739,304,256 is the float nearest to the exact total 739,304,250. The trick, published by William Kahan in 1965 [23], is to remember what each addition lost and give it back in the next one. When the running sum t is large and the new value y is small, the computed $t + y$ loses the low-order digits of y. But $(t + y) - t$ can be computed *exactly* in floating point (under mild conditions), and subtracting y from it gives the lost part with its sign:

$$c = ((t + y) - t) - y \tag{1.16}$$

The variable `carry` holds c, and the next value to add is corrected as $y - c$. Kahan's method bounds the total error by about $2u$ times the sum of the absolute values, independent of the number of terms, instead of growing with it [22].

The lesson for money is simple, and it is the rule that banks and payment systems follow: **count money in integers of its smallest unit** (whole rupiah, or sen if you need them), and use floating point only for quantities that are measured, such as weights and rates. When floating point must be summed at length, use double precision and a compensated or pairwise sum.

1.14 Comparing with the Libraries

Our programs found machine epsilon by experiment and summed with our own code. The libraries offer the same knowledge ready-made, and it is instructive to ask them. **Listing 1.13** uses Python's `math` module and NumPy; **Listing 1.14** uses Java's `Math` and `BigDecimal`.

Listing 1.13 `compare_library.py`: machine epsilon, the next double, float32 and three ways to sum, with NumPy 2.x.

```python
"""What the libraries already know about floating point (needs NumPy)."""
from decimal import Decimal
import math

import numpy as np


def main():
    print(f"float64 eps (np.finfo)  : {np.finfo(np.float64).eps:.6e}")
    print(f"float32 eps (np.finfo)  : {np.finfo(np.float32).eps:.6e}")
    print(f"math.ulp(1.0)           : {math.ulp(1.0):.6e}")
    print(f"next double after 1.0   : {float(np.nextafter(1.0, 2.0))!r}")
    print(f"0.1 stored as float32   : {Decimal(float(np.float32(0.1)))}")
    tenths = [0.1] * 1000000
    loop = 0.0
    for v in tenths:
        loop += v
    print(f"loop sum of 10^6 x 0.1  : {loop!r}")
    print(f"np.sum (pairwise)       : {float(np.sum(np.array(tenths)))!r}")
    print(f"math.fsum (exact round) : {math.fsum(tenths)!r}")


if __name__ == "__main__":
    main()
```

Listing 1.14 `CompareLibrary.java`: the same facts from the Java standard library.

```java
import java.math.BigDecimal;
import java.util.Locale;

/** What the Java standard library already knows about floating point. */
public class CompareLibrary {
    public static void main(String[] args) {
        Locale.setDefault(Locale.ROOT);
        System.out.printf("double eps (Math.ulp)   : %.6e%n", Math.ulp(1.0));
        System.out.printf("float eps (Math.ulp)    : %.6e%n", Math.ulp(1.0f));
        System.out.println("next double after 1.0   : " + Math.nextUp(1.0));
        System.out.println("0.1 stored as float     : " + new BigDecimal(0.1f));
        double loop = 0.0;
        BigDecimal exact = BigDecimal.ZERO;
        BigDecimal tenth = new BigDecimal("0.1");   // decimal, not binary
        for (int i = 0; i < 1000000; i++) {
            loop += 0.1;
            exact = exact.add(tenth);
        }
        System.out.println("loop sum of 10^6 x 0.1  : " + loop);
        System.out.println("BigDecimal sum          : " + exact);
    }
}
```

```
float64 eps (np.finfo)  : 2.220446e-16
float32 eps (np.finfo)  : 1.192093e-07
math.ulp(1.0)           : 2.220446e-16
next double after 1.0   : 1.0000000000000002
0.1 stored as float32   : 0.100000001490116119384765625
loop sum of 10^6 x 0.1  : 100000.00000133288
np.sum (pairwise)       : 100000.00000000003
math.fsum (exact round) : 100000.0
double eps (Math.ulp)   : 2.220446e-16
float eps (Math.ulp)    : 1.192093e-07
next double after 1.0   : 1.0000000000000002
0.1 stored as float     : 0.100000001490116119384765625
loop sum of 10^6 x 0.1  : 100000.00000133288
BigDecimal sum          : 100000.0
```

The libraries confirm our experiments: `np.finfo` and `math.ulp` report the same epsilon we found by halving, and single precision stores 0.1 as `0.100000001490116119384765625`, correct to only about 8 digits. The three sums of a million tenths are worth a closer look. The simple loop is off by 1.3×10^{-6}. NumPy's `np.sum` uses **pairwise summation**, which adds neighbors in a tree and makes the error grow like $\log_2 n$ instead of n [24]; it is off by only 3×10^{-11}. `math.fsum` tracks the lost parts exactly and returns the correctly rounded sum of the stored values, `100000.0`. In Java, `BigDecimal` with the decimal string `"0.1"` avoids binary fractions altogether and is exact, at the price of being far slower; it is the right tool for money when whole units are not enough. Python's documentation has a good tutorial chapter on these matters [25].

1.15 Numerical Methods in Practice: Profitina

Profitina is an AI visibility platform, live at app.profitina.com, that measures how ready a brand's website is to be cited by AI answer engines and helps fix what it finds. Several of its features rest on exactly the ideas of this chapter.

Money in whole units. A Pro subscription costs Rp50,000 for thirty days, paid via QRIS or with PROFIT, the platform's utility coin on the Solana blockchain. The coin has 6 decimals, which on a blockchain means that every balance is stored as a whole number of the smallest unit, a millionth of a coin. That is Section 1.13's rule applied at scale: amounts that must add up exactly are integers, and a balance never drifts by a fraction nobody can explain. Conversions between PROFIT and rupiah go through a price oracle, and a conversion is the one place where a rate, a real number, meets an amount; the result must be rounded to a whole unit once, at a clear point, by a stated rule.

Scores that add up. The AI Visibility Engine scores a site from many signals and explains each one. Combining many measured values into one number is a sum of products, and Section 1.13 showed that the order and method of summation matter when there are many terms. Normalizing a score to a 0 to 100 scale divides by a range; if the range is tiny, the division magnifies whatever error the inputs carry, the same effect as the $1/h$ in Equation (1.14).

Trends over time. The historical dashboard shows a weekly score trend per brand and exports it as CSV. A difference of two nearly equal scores, such as "up 0.3 points this week", is a subtraction of nearly equal numbers, and it is only meaningful when the scores themselves are known to more digits than the difference shows: catastrophic cancellation in the language of business. The general lesson is the same as in Bu Wati's laundry: a reported number must be stored and computed with enough precision that its last shown digit can be trusted.

1.16 Using It for Real

Everything in this chapter applies directly to the spreadsheets and small programs that shops, farms and clinics use every day.

- **Money columns.** In a spreadsheet, keep prices and totals in whole rupiah and round each receipt once, when it is made, with the spreadsheet's `ROUND` function; never compare two computed totals with "equals" but check that their difference is below half a rupiah. In a program, keep money in a `long` (Java) or an `int` (Python), or in `BigDecimal` and `Decimal` when sen are needed.

- **Measured values.** Weights, lengths, temperatures and rates are measured with few digits anyway; doubles are more than accurate enough for them, and floats usually are too, as long as you do not sum thousands of them into a large total.
- **Long sums.** A daily log of sensor readings, a year of transactions, a long simulation: use double precision, and for very long sums use `math.fsum` or NumPy's `sum` in Python, or a Kahan sum in Java.
- **Differences of nearly equal numbers.** Before subtracting two large, nearly equal numbers, ask whether the formula can be rearranged, as the quadratic formula could. Often it can.

1.17 Common Mistakes, and How to Avoid Them

- **Comparing doubles with `==`.** `0.1 + 0.2 == 0.3` is false. Compare with a tolerance, relative for large numbers, absolute near zero.
- **Keeping money in floating point.** Rp826 disappeared from a year of float totals. Count money in integers of its smallest unit.
- **Believing all printed digits.** A double prints 17 significant digits, but after a cancellation only a few of them may be right. The number of trustworthy digits is decided by the error analysis, not by the printout.
- **Making the step "as small as possible".** A forward difference with $h = 10^{-16}$ gave 0 instead of 0.35. The best step balances truncation against round-off (Equation (1.15)).
- **Summing an alternating series with huge terms.** e^{-20} from its series was wrong by a factor of three. Rearrange (here, $1/e^{20}$) so that large terms of opposite sign do not cancel.
- **Stopping on the true error.** A program cannot know the true error. Use the approximate error Equation (1.3) with a tolerance from Equation (1.4), and remember that a slowly converging method may stop too early.
- **Printing with the wrong locale in Java.** On a computer set to Indonesian, `printf("%.2f", 1.5)` prints `1,50`. Set `Locale.setDefault(Locale.ROOT)` when the output is data, not text for people.
- **Forgetting to activate the virtual environment.** `ModuleNotFoundError: No module named 'numpy'` usually means that the terminal runs the system Python instead of `.venv`. Activate it, or choose the `.venv` interpreter in VS Code.

1.18 Chapter Summary

- A numerical method replaces a problem with a finite sequence of arithmetic operations whose answer is close to the true one by an amount we can estimate. Methods are judged by convergence, speed of convergence and stability.
- The workbench is JDK 25 LTS, Python 3.14 with a virtual environment holding NumPy and SciPy, and VS Code with the Java and Python extensions; `check_setup.py` and `CheckSetup.java` confirm it.
- Errors are measured as true error Equation (1.1), relative error Equation (1.2) and, when the truth is unknown, approximate relative error Equation (1.3); iterations stop when it falls below $\varepsilon_s = 0.5 \times 10^{2-n}\%$ for n digits (Equation (1.4)).
- A double is stored as a sign, an 11-bit exponent and a 52-bit fraction (Equation (1.6)); most decimal fractions, such as 0.1 (Equation (1.5)), are stored slightly wrong. Every stored number and every operation has a relative error of at most $u = 2^{-53}$ (Equation (1.8)); machine epsilon is 2^{-52} (Equation (1.7)).

- Round-off errors accumulate over many operations, and subtracting nearly equal numbers exposes them (catastrophic cancellation); rearranging a formula, as in Equation (1.10), cures it.
- Truncation error comes from cutting an infinite process short, as in the Taylor series Equation (1.11) and the forward difference Equation (1.13). Shrinking the step lowers truncation error but raises round-off error; the best step balances them (Equation (1.14), Equation (1.15)).
- Money is counted in integers; long sums of measured values use double precision and compensated (Equation (1.16)) or pairwise summation. Our programs agree with NumPy's and Java's library answers.

1.19 Exercises

1. Compute by hand the true error, the relative error and the number of correct significant digits of the approximations 3.14 and 3.1416 to π.
2. Write 0.2 and 0.375 in binary. Which of them is stored exactly as a double, and why?
3. Decode the double whose 64 bits, written in hexadecimal, are `3FF8000000000000`. (Hexadecimal digit 3 is 0011, F is 1111, 8 is 1000.)
4. How many terms of the series Equation (1.12) does the stopping rule need to give e^1 to 6 significant digits? Work out the first terms by hand and then change `exp_series.py` or `ExpSeries.java` to check.
5. The formula $\sqrt{x+1}-\sqrt{x}$ loses accuracy for large x. Explain why, rewrite it without the subtraction, and compare both forms at $x = 10^{12}$ with a few lines of Python or Java.
6. Sum $1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{10^6}$ in single precision once forward (from 1 up) and once backward (from $\frac{1}{10^6}$ down), and compare both with the double-precision sum. Which order is better, and why?
7. Using Equation (1.15), predict the best step for the forward difference of $f(x) = e^x$ at $x = 1$ and the smallest error you can expect.
8. Bu Wati raises her price to Rp8,000 per kilogram and her scale now shows weights to 10 g. Is the integer total of `laundry_ledger.py` still exact? What changes if the price becomes Rp7,750?

1.20 Solutions to the Exercises

1. With $\pi = 3.14159265$: for 3.14, $E_t = 0.00159265$ and $\varepsilon_t = 0.00159265/3.14159265 = 5.07 \times 10^{-4} = 0.0507\%$. That is below 5×10^{-3} but not below 5×10^{-4}, so 3.14 has 3 correct significant digits. For 3.1416, $E_t = -0.00000735$ and $\varepsilon_t = -2.34 \times 10^{-6}$, below 5×10^{-6}: 5 significant digits.

2. Doubling 0.2: 0.4 (0), 0.8 (0), 1.6 (1, keep 0.6), 1.2 (1, keep 0.2), and 0.2 returns, so $0.2 = 0.0011\,0011\,\ldots_2$, infinitely long and not stored exactly. Doubling 0.375: 0.75 (0), 1.5 (1, keep 0.5), 1.0 (1, keep 0), so $0.375 = 0.011_2$, exact. A fraction is exact in binary exactly when its denominator in lowest terms is a power of two: $0.375 = 3/8$, but $0.2 = 1/5$.

3. The bits are 0011 1111 1111 1000 0000 … The sign is 0. The exponent is the next 11 bits, $01111111111_2 = 1023$, so the power is 2^0. The fraction starts with 1000 …, so the significand is $1.1_2 = 1.5$. The number is 1.5.

4. With $n = 6$, $\varepsilon_s = 0.5 \times 10^{-4}\% = 0.00005\%$. The partial sums of e^1 are 1, 2, 2.5, 2.666667, 2.708333, 2.716667, 2.718056, 2.718254, 2.718279, 2.718282. The added term divided by the new sum is $(1/9!)/2.718282 = 1.01 \times 10^{-6}$, that is 0.000101%, after 10 terms, still above the

tolerance, and $(1/10!)/2.718282 = 1.01 \times 10^{-7}$, that is 0.0000101%, after 11 terms, the first below 0.00005%. So 11 terms are needed. Calling `exp_series(1.0, 6, show=True)` prints the table and stops at 11 terms, with a true relative error of about 0.000001%.

5. For large x both square roots are almost equal, so the subtraction cancels their leading digits and leaves round-off. Multiply and divide by the sum: $\sqrt{x+1} - \sqrt{x} = \frac{(x+1)-x}{\sqrt{x+1}+\sqrt{x}} = \frac{1}{\sqrt{x+1}+\sqrt{x}}$. At $x = 10^{12}$ the direct form gives `5.000038072466850e-07` and the rewritten form `4.999999999998749e-07`; the true value is $4.99999999999875 \times 10^{-7}$, so the direct form has only 5 correct digits and the rewritten form 15. In Python: `x = 1e12; print(math.sqrt(x + 1) - math.sqrt(x), 1 / (math.sqrt(x + 1) + math.sqrt(x)))`.

6. Listing 1.15 and **Listing 1.16** do it.

Listing 1.15 `harmonic32.py`: the harmonic sum in single precision, forward and backward.

```
"""Exercise 1.6: the harmonic sum in single precision, forward and backward."""
import struct

N = 1000000


def f32(x):
    """Round to the nearest single-precision number."""
    return struct.unpack("f", struct.pack("f", x))[0]


def main():
    forward = 0.0
    for k in range(1, N + 1):
        forward = f32(forward + f32(1.0 / k))
    backward = 0.0
    for k in range(N, 0, -1):
        backward = f32(backward + f32(1.0 / k))
    double = 0.0
    for k in range(N, 0, -1):
        double += 1.0 / k
    print(f"float, 1 up to N   : {forward:.7f}")
    print(f"float, N down to 1 : {backward:.7f}")
    print(f"double, N down to 1: {double:.7f}")


if __name__ == "__main__":
    main()
```

Listing 1.16 `Harmonic32.java`: the same in Java.

```java
import java.util.Locale;

/** Exercise 1.6: the harmonic sum in single precision, forward and backward. */
public class Harmonic32 {
    static final int N = 1000000;

    public static void main(String[] args) {
        Locale.setDefault(Locale.ROOT);
        float forward = 0.0f;
        for (int k = 1; k <= N; k++) {
            forward = forward + (float) (1.0 / k);
        }
        float backward = 0.0f;
        for (int k = N; k >= 1; k--) {
            backward = backward + (float) (1.0 / k);
        }
        double dbl = 0.0;
        for (int k = N; k >= 1; k--) {
            dbl += 1.0 / k;
        }
        System.out.printf("float, 1 up to N   : %.7f%n", forward);
        System.out.printf("float, N down to 1 : %.7f%n", backward);
        System.out.printf("double, N down to 1: %.7f%n", dbl);
    }
}
```

```
float, 1 up to N   : 14.3573580
float, N down to 1 : 14.3926516
double, N down to 1: 14.3927267
```

The backward sum is right to 5 digits (14.39265 against 14.39273); the forward sum is wrong in the third digit. Going forward, the running sum is soon about 10, where floats are about 10^{-6} apart, and the small terms $1/k$ for large k are rounded away almost completely. Going backward, the small terms are added to each other first, while the running sum is still small, and the large terms come last. Adding numbers from the smallest to the largest keeps the running sum as small as possible for as long as possible.

7. For e^x at 1, $f = f'' = e = 2.718$, so $h_{\text{opt}} = 2\sqrt{\varepsilon_{\text{mach}}\, e/e} = 2\sqrt{2.22 \times 10^{-16}} = 3.0 \times 10^{-8}$, and the smallest error is about $Mh_{\text{opt}} = 2.718 \times 3.0 \times 10^{-8} \approx 8 \times 10^{-8}$, a relative error of about 3×10^{-8}: only half of the 16 digits of the machine survive a forward difference.

8. At Rp8,000 per kilogram, grams $\times\, 8000/1000 = 8 \times$ grams, a whole number for any whole number of grams: still exact. At Rp7,750 and weights in steps of 10 g, grams $\times\, 7750/1000 = 7.75 \times$ grams, which for 1,010 g gives Rp7,827.5. The price is no longer a whole number of rupiah, so the shop must decide how to round each receipt (for example, up to the next Rp100, as many shops do), and the program must round each receipt once, with integer arithmetic such as `(grams * 7750 + 999) // 1000` for rounding up to whole rupiah, before adding. Integer division `//` in Python and `/` on `long` values in Java both drop the fraction, which is why a rounding term is added first.

References

[1] U.S. General Accounting Office, “Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia,” GAO/IMTEC-92-26, Feb. 1992. [Online]. Available: https://www.gao.gov/products/imtec-92-26

[2] S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers*, 8th ed. New York, NY, USA: McGraw-Hill Education, 2021.

[3] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical Analysis*, 10th ed. Boston, MA, USA: Cengage Learning, 2016.

[4] Oracle, "Java Downloads: JDK 25 (LTS) and JDK 27," accessed Sep. 2026. [Online]. Available: https://www.oracle.com/java/technologies/downloads/

[5] OpenJDK, "JDK 25," General Availability Sep. 16, 2025. [Online]. Available: https://openjdk.org/projects/jdk/25/

[6] Python Software Foundation, "Python 3.14.7," Aug. 5, 2026. [Online]. Available: https://www.python.org/downloads/release/python-3147/

[7] NumPy Developers, "numpy 2.5.3," Python Package Index, Sep. 6, 2026. [Online]. Available: https://pypi.org/project/numpy/

[8] SciPy Developers, "scipy 1.18.1," Python Package Index, Aug. 21, 2026. [Online]. Available: https://pypi.org/project/scipy/

[9] Microsoft, "Visual Studio Code 1.139," release notes, Sep. 23, 2026. [Online]. Available: https://code.visualstudio.com/updates

[10] Microsoft, "Java in Visual Studio Code," *VS Code Documentation*. [Online]. Available: https://code.visualstudio.com/docs/languages/java

[11] Microsoft, "Python in Visual Studio Code," *VS Code Documentation*. [Online]. Available: https://code.visualstudio.com/docs/languages/python

[12] Inside.java, "The Arrival of Java 27!," Sep. 15, 2026. [Online]. Available: https://inside.java/2026/09/15/jdk-27-available/

[13] Oracle, "JDK 19 Release Notes: Double.toString(double) and Float.toString(float) May Return Slightly Different Results," 2022. [Online]. Available: https://www.oracle.com/java/technologies/javase/19-relnote-issues.html

[14] Adoptium, "Eclipse Temurin 8u504, 11.0.32.1, 17.0.20.1, 21.0.12.1, 25.0.4.1 and 26.0.2.1 Available," Sep. 2, 2026. [Online]. Available: https://adoptium.net/news/2026/09/eclipse-temurin-8u504-110321-170201-210121-25041-26021-available

[15] Python Software Foundation, "What's New in Python 3.14," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3.14/whatsnew/3.14.html

[16] OpenJDK, "JEP 330: Launch Single-File Source-Code Programs," JDK 11. [Online]. Available: https://openjdk.org/jeps/330

[17] Python Software Foundation, "Using Python on Windows," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3.14/using/windows.html

[18] Phoronix, "Ubuntu 26.04 LTS Moving To OpenJDK 25 By Default," 2026. [Online]. Available: https://www.phoronix.com/news/Ubuntu-26.04-OpenJDK-25

[19] Python Software Foundation, "venv — Creation of virtual environments," *The Python Standard Library*. [Online]. Available: https://docs.python.org/3/library/venv.html

[20] OpenJDK, "JEP 512: Compact Source Files and Instance Main Methods," JDK 25. [Online]. Available: https://openjdk.org/jeps/512

[21] *IEEE Standard for Floating-Point Arithmetic*, IEEE Std 754-2019, Jul. 2019.

[22] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: SIAM, 2002.

[23] W. Kahan, "Further remarks on reducing truncation errors," *Communications of the ACM*, vol. 8, no. 1, p. 40, 1965.

[24] NumPy Developers, "numpy.sum," *NumPy Reference*. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.sum.html

[25] Python Software Foundation, "Floating-Point Arithmetic: Issues and Limitations," *The Python Tutorial*. [Online]. Available: https://docs.python.org/3/tutorial/floatingpoint.html

Chapter 2

Root Finding I: Bisection and Regula Falsi

Many everyday questions end in an equation that no formula can solve: what interest rate is hidden in an instalment plan, how deep the water stands in a round tank that holds 3,000 litres, when a falling price reaches a given level. Such questions ask for a *root*, a number that makes some function zero. This chapter builds the two oldest and safest ways to find one. Both start from an interval in which the function changes sign and shrink it until the root is pinned down. By the end of the chapter you will be able to turn a practical question into a root-finding problem; explain and prove why a sign change guarantees a root; carry out bisection and regula falsi by hand; predict exactly how many bisection steps a given accuracy needs; recognize when regula falsi crawls and repair it with the Illinois modification; implement all three methods in Java and Python with iteration tables that agree to the last digit; and check the results against SciPy's professional root finders.

2.1 Mas Dimas and the "1.67% a Month" Phone

Mas Dimas drives an online ojek in Yogyakarta. His phone is his office: the orders arrive on it, the map runs on it, and the payments go through it. When the old phone's screen cracked for the third time, he went to a phone counter in a shopping center to buy a new one on instalments. The phone costs Rp6,000,000. The sales assistant offered twelve monthly instalments of Rp600,000, "with interest of only 1.67% a month".

Dimas did the arithmetic on the back of the brochure. Twelve times Rp600,000 is Rp7,200,000, which is Rp1,200,000 more than the price. Rp1,200,000 spread over 12 months is Rp100,000 a month, and Rp100,000 is 1.67% of Rp6,000,000. So the assistant was not lying. But something bothered him. In the last months of the plan he would owe the shop very little, yet he would still pay the same Rp100,000 of "interest" each month. Surely the interest on a small debt should be small?

He was right to be bothered. The "1.67%" is a **flat rate**: it is computed on the original price for the whole term, although the debt shrinks with every instalment. The rate that a bank would compute, the rate at which the shrinking debt actually grows each month, is called the **effective rate**, and it is considerably higher. Indonesian consumer-protection rules require financial service providers to explain the costs of a product clearly to their customers [1], and the effective rate is exactly what allows two offers to be compared fairly. But the brochure does not print it, and Dimas has no formula to compute it, because none exists. The rate is the root of an equation.

This chapter finds that root. It is arithmetic, not financial advice: the program tells Dimas what rate a given plan implies, so that he can compare it with a bank loan, a cooperative (*koperasi*), or saving for three months first; the decision stays his. The same program works for a motorbike, a refrigerator or a laptop, and the same methods find roots in every chapter that follows.

2.2 The Idea in Plain Words

2.2.1 Money today and money later

A rupiah paid a month from now is worth less to the shop than a rupiah paid today, because a rupiah today could be lent out and earn interest for a month. If the monthly interest rate is r (for example $r = 0.02$ for 2%), then Rp1 today grows to Rp$(1+r)$ after one month and to Rp$(1+r)^k$ after k months. Turned around, an instalment A paid after k months is worth $A/(1+r)^k$ today; this is its **present value**.

A fair instalment plan is one in which the present values of all instalments add up to the price P:

$$P = \frac{A}{1+r} + \frac{A}{(1+r)^2} + \cdots + \frac{A}{(1+r)^n} \tag{2.1}$$

The right-hand side of Equation (2.1) is a geometric series with first term $a = A/(1+r)$ and ratio $q = 1/(1+r)$. The usual formula for such a sum is $a(1-q^n)/(1-q)$; here $1-q = r/(1+r)$, so $a/(1-q) = \frac{A}{1+r}\cdot\frac{1+r}{r} = A/r$, and $q^n = (1+r)^{-n}$. The sum therefore simplifies to

$$P = A\,\frac{1-(1+r)^{-n}}{r} \tag{2.2}$$

This is the **annuity formula**, and banks use it in both directions. Given the rate r, it tells you the instalment A; for example, at $r = 2\%$ a price of Rp6,000,000 over 12 months needs $A = 6{,}000{,}000/10.5753 =$ Rp567,358. But Dimas knows P, A and n and wants r, and r appears in a power and in a denominator at once. No sequence of algebraic steps isolates it.

2.2.2 Turning the question into a root

Every such question can be put in the same shape: move everything to one side, and ask where the result is zero. Define

$$f(r) = A\,\frac{1-(1+r)^{-n}}{r} - P \tag{2.3}$$

For Dimas, $P = 6{,}000{,}000$, $A = 600{,}000$ and $n = 12$. The rate we are looking for is the number r^* with $f(r^*) = 0$, a **root** (or *zero*) of f. In words, $f(r)$ is "how much the instalments are worth today at rate r, minus the price": positive when the rate r is too low (the instalments are worth more than the phone), negative when r is too high.

2.2.3 A table of guesses

Before any method, let us simply try some rates, as Dimas might do with a calculator. The fraction in Equation (2.2) is called the annuity factor; multiplied by A and reduced by P it gives f.

Table 2.1 The loan function Equation (2.3) at five monthly rates: it changes sign between 2% and 3%.

Rate r	Annuity factor	$f(r)$ (rupiah)
1%	11.2551	+753,046
2%	10.5753	+345,205
3%	9.9540	−27,598
4%	9.3851	−368,956
5%	8.8633	−682,049

Table 2.1 already tells us a great deal. At 2% the instalments are worth Rp345,205 more than the phone, at 3% they are worth Rp27,598 less, so the true rate lies between 2% and 3%, and since $-27{,}598$ is much closer to zero than $+345{,}205$, it probably lies close to 3%. The sign change is the key. The two methods of this chapter are nothing but disciplined ways of continuing this table: bisection always tries the middle of the interval where the sign changes, and regula falsi uses the *sizes* of the two values, as we just did by eye, to guess where the zero is.

2.3 Bisection

2.3.1 Why a sign change guarantees a root

The whole method rests on one theorem about continuous functions, functions whose graph can be drawn without lifting the pencil.

> **DEFINITION.** *Intermediate Value Theorem.* If f is continuous on the closed interval $[a, b]$ and K is any number between $f(a)$ and $f(b)$, then there is at least one c in $[a, b]$ with $f(c) = K$. In particular, if $f(a)$ and $f(b)$ have opposite signs, that is $f(a)\,f(b) < 0$, then f has at least one root in (a, b) [2].

The picture behind it is plain: a continuous path that starts below the axis and ends above it must cross the axis somewhere. Our loan function Equation (2.3) is continuous for every $r > 0$ (it is built from additions, powers and a division by r, which is not zero there), and $f(0.01) > 0 > f(0.05)$, so the theorem promises a root between 1% and 5%. An interval $[a, b]$ with $f(a)f(b) < 0$ is called a **bracket**, and methods that keep a bracket at every step are called *bracketing methods*.

Two warnings belong here. The theorem promises *at least* one root; if the interval contains three, bisection finds one of them, and which one depends on the interval. And continuity matters: $f(x) = 1/x$ changes sign on $[-1,1]$ but has no root there, and bisection would happily converge to the pole at 0. For the loan function neither problem arises, because f is decreasing: a higher rate always makes the instalments worth less today, so there is exactly one root.

2.3.2 The method

Bisection halves the bracket again and again. Take the midpoint c of $[a, b]$. If $f(c)$ has the same sign as $f(a)$, the sign change, and so the root, is in $[c, b]$; otherwise it is in $[a, c]$. Either way we have a new bracket of half the width. **Pseudocode 2.1** writes the procedure out.

Pseudocode 2.1 The bisection method with a tolerance on the change of the midpoint.

```
BISECTION(f, a, b, tol, maxIter)
 1  fa = f(a);  fb = f(b)
 2  if fa * fb > 0: error "no sign change: [a, b] is not a bracket"
 3  c_old = a
 4  for k = 1 to maxIter
 5      c = a + (b - a) / 2              // the midpoint
 6      fc = f(c)
 7      if fc == 0 or |c - c_old| < tol: return c
 8      if fa * fc < 0: b = c; fb = fc   // root in [a, c]
 9      else:           a = c; fa = fc   // root in [c, b]
10      c_old = c
11  return c                             // tolerance not reached
```

Let us run it by hand on Dimas's function with the bracket [0.01,0.05], using the values of **Table 2.1**.

Step 1. $c = 0.01 + 0.04/2 = 0.03$ and $f(0.03) = -27{,}598$. Since $f(0.01) > 0$ and $f(0.03) < 0$, the signs differ: the root is in [0.01,0.03], so b becomes 0.03.

Step 2. $c = 0.01 + 0.02/2 = 0.02$ and $f(0.02) = +345{,}205$, the same sign as $f(a)$: the root is in [0.02,0.03], so a becomes 0.02.

Step 3. $c = 0.025$, and the annuity factor at 2.5% is 10.25776, so $f(0.025) = 600{,}000 \times 10.25776 - 6{,}000{,}000 = +154{,}659$: positive, $a = 0.025$, bracket [0.025,0.03].

Step 4. $c = 0.0275$, $f = +62{,}522$: positive, bracket [0.0275,0.03].

After four steps the rate is known to lie between 2.75% and 3%, and it is already clear that the seller's "1.67%" is not the effective rate. **Figure 2.1** draws the function and the first brackets.

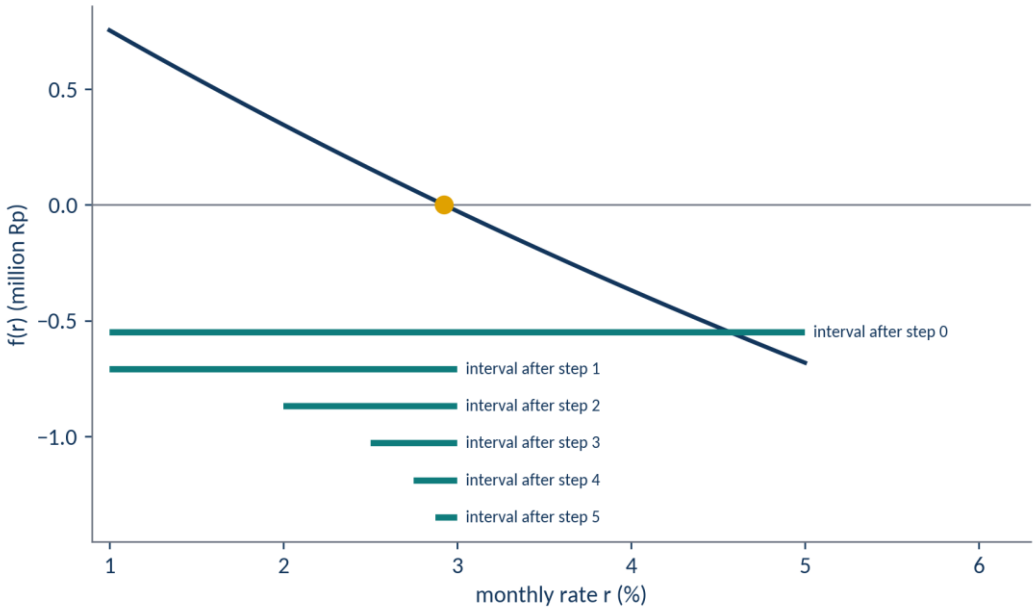


Figure 2.1 The loan function $f(r)$ of Equation (2.3) and the bracket after each bisection step, starting from $[1\%, 5\%]$: every step keeps the half that contains the sign change (amber dot).

2.3.3 How many steps: the error bound

Bisection has a property that no other method in this book has: we know *in advance* how many steps it needs. Write $[a_0, b_0]$ for the first bracket and c_k for the k-th midpoint. The bracket that c_k halves has width $(b_0 - a_0)/2^{k-1}$, the root lies somewhere in it, and c_k is its middle, so

$$|c_k - r^*| \le \frac{b_0 - a_0}{2^k} \tag{2.4}$$

Each step halves the bound: bisection gains exactly one binary digit of the answer per step, whatever the function looks like [2]. To be sure of an error below a tolerance τ it suffices that $(b_0 - a_0)/2^k \le \tau$, which after taking logarithms gives

$$k \ge \log_2 \frac{b_0 - a_0}{\tau} \tag{2.5}$$

For Dimas, $b_0 - a_0 = 0.04$ and $\tau = 10^{-7}$ (a rate correct to 0.00001%, far more than needed), so $k \ge \log_2 400{,}000 = 18.6$: 19 steps. Since $\log_2 10 \approx 3.32$, each additional correct *decimal* digit costs a little more than three steps, and double precision's 16 digits are exhausted after about 53 halvings of an interval of length 1. The program's stopping test $|c_k - c_{k-1}| < \tau$ fits this analysis exactly, because two successive midpoints are always half a bracket apart: $|c_k - c_{k-1}| = (b_0 - a_0)/2^k$, the right-hand side of Equation (2.4).

A method whose error shrinks by a constant factor $\lambda < 1$ at each step, $|e_{k+1}| \approx \lambda |e_k|$, is said to **converge linearly** with rate λ. Bisection's bound shrinks with $\lambda = 1/2$. (The actual error $|c_k - r^*|$ jumps up and down, because a midpoint may by chance land very near the root and the next one farther away; only the bound falls smoothly.) Chapter 3 meets methods that converge much faster, but none of them comes with a guarantee like Equation (2.5).

2.3.4 Finding a bracket in the first place

Everything so far assumed that we already had a bracket. For the loan, common sense gave one: no instalment plan in a shop charges less than 1% or more than 5% a month, and **Table 2.1** confirmed the signs. When common sense is silent, a bracket can be *searched for*. The simplest search walks from a starting point x_0 in steps of size Δ, computing $f(x_0), f(x_0 + \Delta), f(x_0 + 2\Delta), \ldots$, until two neighboring values differ in sign; those two points are a bracket. The step must be small enough not to jump over two roots at once (two sign changes cancel and the walk sees none) and large enough not to waste thousands of evaluations. A second strategy doubles the step each time, $x_0 + \Delta, x_0 + 2\Delta, x_0 + 4\Delta, \ldots$, which finds a far-away sign change in few steps; it is the natural choice for a quantity such as a rate or a depth that is known to be positive but whose size is unknown.

A plot is even better. One look at the graph of f over a generous range shows where it crosses the axis, how many times, and whether it has poles or flat stretches. **Figure 2.1** is such a plot for the loan. Plotting first and bracketing second is a habit worth forming: it catches most of the mistakes of Section 2.12 before a single line of the root finder runs.

> **KEY INSIGHT:** A bracketing method turns a question that has no formula into a sequence of yes-or-no questions ("is the root left or right of this point?"), each answered by the sign of one function value. That is why it is so robust: the sign of $f(c)$ is almost always computed correctly, even when the value itself is inaccurate.

2.3.5 Small details that matter

Three details in **Pseudocode 2.1** look fussy and are not.

The midpoint is computed as $a + (b - a)/2$, not as $(a + b)/2$. In exact arithmetic they are equal. In floating point, $(a + b)/2$ can fall *outside* $[a, b]$ when a and b are close together and the

rounding of $a + b$ goes the wrong way, and for very large a and b the sum can overflow. The form $a + (b - a)/2$ adds a small correction to a and stays inside the bracket [3].

The sign test uses the stored $f(a)$, updated when a moves, so that each step evaluates f only once. For the loan, one evaluation is cheap; for a function that runs a simulation, the number of evaluations is the whole cost of the method. (A careful implementation compares signs rather than multiplying, because the product of two tiny values can underflow to zero and of two huge ones overflow; for functions of ordinary size, as here, the product is fine.)

The stopping test is on the change in c, with a maximum number of iterations as a safety net. Other tests are possible: a small $|f(c)|$, or a small relative change $|c_k - c_{k-1}|/|c_k|$ as in the approximate relative error of Section 1.9.2. A test on $|f(c)|$ alone is dangerous, because "small" depends on the scale of f: our loan function is measured in rupiah, and $|f| < 1$ means one rupiah, while the same test on a function measured in millions of rupiah would mean something quite different.

2.4 Regula Falsi

2.4.1 The chord

Bisection ignores useful information. In **Table 2.1**, $f(0.02) = +345{,}205$ and $f(0.03) = -27{,}598$; a person would guess that the root is much closer to 3% than to 2%, but bisection tries 2.5%. **Regula falsi**, the "rule of false position", uses the sizes of the two values. It draws the straight line, the *chord*, through $(a, f(a))$ and $(b, f(b))$, and takes the point c where the chord crosses the axis as the next guess. The chord has slope $(f(b) - f(a))/(b - a)$; the line through $(b, f(b))$ with that slope is zero where

$$c = b - f(b)\,\frac{a - b}{f(a) - f(b)} \tag{2.6}$$

The method is very old: Egyptian, Babylonian and Chinese scribes used "false position" to solve linear problems, and medieval Arabic and European arithmetic books taught it as a standard rule [4]. The bracket is then updated exactly as in bisection, keeping the part where the sign changes.

Its first step for Dimas, with $a = 0.01$, $f(a) = 753{,}046.5$, $b = 0.05$, $f(b) = -682{,}049.0$, is

$$c = 0.05 - (-682{,}049.0)\,\frac{0.01 - 0.05}{753{,}046.5 + 682{,}049.0} = 0.05 - \frac{27{,}282.0}{1{,}435{,}095.5} = 0.030989$$

already within 0.2 percentage points of the root, where bisection's first midpoint 0.03 was within 0.08 points by luck but its second, 0.02, fell far back. **Pseudocode 2.2** shows the whole method.

Pseudocode 2.2 Regula falsi, with the Illinois modification switched on by a flag (lines 10 and 14).

```
REGULA-FALSI(f, a, b, tol, maxIter, illinois)
 1  fa = f(a);  fb = f(b)
 2  if fa * fb > 0: error "no sign change"
 3  c_old = a;  side = 0
 4  for k = 1 to maxIter
 5      c = b - fb * (a - b) / (fa - fb)     // where the chord meets zero
 6      fc = f(c)
 7      if fc == 0 or |c - c_old| < tol: return c
 8      if fa * fc < 0:                      // root in [a, c]: b moves
 9          b = c; fb = fc
10          if illinois and side == -1: fa = fa / 2
11          side = -1
12      else:                                // root in [c, b]: a moves
13          a = c; fa = fc
14          if illinois and side == +1: fb = fb / 2
15          side = +1
16      c_old = c
17  return c
```

2.4.2 Why regula falsi can crawl

Regula falsi keeps the bracket, so it can never lose the root, and when the function is nearly straight it converges quickly. But it has a weakness that bisection does not. If f is curved in the same direction over the whole bracket (convex or concave), the chord always lies on the same side of the graph, the new point c always falls on the same side of the root, and one end of the bracket *never moves*. Our loan function is convex (its graph bends upward), f is negative to the right of the root, and every chord crosses the axis to the right of the root; so $a = 0.01$ stays fixed for the whole computation, and only b creeps toward the root.

With one end fixed at a, the method still converges, but only linearly, and we can compute its rate. Write the moving end as $b = r^* + e$ with a small error e, so that $f(b) \approx f'(r^*)\,e$ by Taylor's formula. Putting this into Equation (2.6) and keeping only the terms of first order in e gives $c - r^* \approx \lambda\, e$ with

$$\lambda = 1 - \frac{f'(r^*)\,(a - r^*)}{f(a)} \tag{2.7}$$

The fraction compares two slopes: the slope $f'(r^*)$ of the tangent at the root with the slope $f(a)/(a - r^*)$ of the chord from the fixed end to the root. When the two are nearly equal, the function is nearly straight between a and the root, λ is near 0, and the method is fast. When the chord is much steeper or much flatter than the tangent, λ approaches 1 and the method crawls. For Dimas's function the tangent slope at the root is about -3.59×10^7 rupiah per unit of rate and the chord slope from $a = 0.01$ is -3.92×10^7, so $\lambda = 1 - 3.59/3.92 = 0.083$: each step should cut the error by a factor of about 12, and Section 2.6 will measure exactly that. The classic example is $f(x) = x^{10} - 1$ on [0,1.3] [5]: the function is almost flat near 0 and very steep near 1.3, so every chord from the fixed right end $b = 1.3$ is nearly the same steep line, and each step moves a only a little toward the root 1. With $f'(1) = 10$ and $f(1.3) = 12.79$, Equation (2.7) (with b in the role of the fixed end) gives $\lambda = 1 - 10 \times 0.3/12.79 = 0.77$: each step removes less than a quarter of the error. Section 2.7 measures it. A second problem follows from the first: with one end fixed, the width of the bracket does not shrink to zero, so a test on the width can never succeed, and a test on the change $|c_k - c_{k-1}|$ can succeed while the error is still larger than the tolerance, because a slowly crawling sequence changes little from step to step.

2.4.3 The Illinois modification

The cure is simple and elegant. When the same end of the bracket has stayed twice in a row, halve the function value stored for it:

$$f(a) \leftarrow \frac{1}{2} f(a) \tag{2.8}$$

(or the same for b). The halved value pulls the next chord toward the stuck end, so the next point c jumps to the *other* side of the root, and the stuck end finally moves. This is the **Illinois method**, long known under the name of the University of Illinois computing laboratory and analyzed by Dowell and Jarratt in 1971 [6]. They proved that its order of convergence is about 1.442: roughly, the number of correct digits is multiplied by 1.44 at each step, which is far better than a constant gain. Lines 10 and 14 of **Pseudocode 2.2** implement it with a variable `side` that remembers which end moved last.

2.5 Building It in Java and Python

The three methods go into one module, **Listing 2.1** in Python and **Listing 2.2** in Java, so that the loan program and every later program can use them. Each function receives the function f as a parameter. In Python a function is an ordinary value that can be passed around; in Java the parameter has the type `DoubleUnaryOperator`, a standard interface for "a function from `double` to `double`", and the caller can pass a lambda expression such as `x -> x * x - 2`.

Listing 2.1 `roots.py`: bisection, regula falsi and the Illinois variant, with an optional iteration table.

```
"""Bracketing root finders: bisection, regula falsi and the Illinois variant."""

HEADER = f"{'k':>3} {'a':>11} {'b':>11} {'c':>11} {'f(c)':>15} {'|c - c_old|':>12}"


def _check(fa, fb):
    if fa * fb > 0:
        raise ValueError("f(a) and f(b) have the same sign: no bracket")


def bisection(f, a, b, tol=1e-7, max_iter=100, trace=False):
    """Halve [a, b] until two midpoints differ by less than tol.
    Returns (root, number of iterations)."""
    fa, fb = f(a), f(b)
    _check(fa, fb)
    c_old = a
    for k in range(1, max_iter + 1):
        c = a + (b - a) / 2                 # safer than (a + b) / 2
        fc = f(c)
        if trace:
            print(f"{k:>3} {a:>11.8f} {b:>11.8f} {c:>11.8f} {fc:>15.4f} "
                  f"{abs(c - c_old):>12.3e}")
        if fc == 0 or abs(c - c_old) < tol:
            return c, k
        if fa * fc < 0:                     # the root is in [a, c]
            b, fb = c, fc
        else:                               # the root is in [c, b]
            a, fa = c, fc
        c_old = c
    return c, max_iter


def regula_falsi(f, a, b, tol=1e-7, max_iter=100, trace=False,
                 illinois=False):
    """False position: cut [a, b] where the chord through (a, f(a)) and
    (b, f(b)) crosses zero. With illinois=True, an end point that stays
    twice in a row has its function value halved."""
    fa, fb = f(a), f(b)
    _check(fa, fb)
    c_old = a
    side = 0                                # which end stayed last time
    for k in range(1, max_iter + 1):
        c = b - fb * (a - b) / (fa - fb)    # where the chord meets zero
        fc = f(c)
        if trace:
            print(f"{k:>3} {a:>11.8f} {b:>11.8f} {c:>11.8f} {fc:>15.4f} "
```

Listing 2.1 (Continued) *roots.py: bisection, regula falsi and the Illinois variant, with an optional iteration table.*

```
                  f"{abs(c - c_old):>12.3e}")
        if fc == 0 or abs(c - c_old) < tol:
            return c, k
        if fa * fc < 0:                     # root in [a, c]: b moves
            b, fb = c, fc
            if illinois and side == -1:     # a stayed twice: halve f(a)
                fa /= 2
            side = -1
        else:                               # root in [c, b]: a moves
            a, fa = c, fc
            if illinois and side == 1:      # b stayed twice: halve f(b)
                fb /= 2
            side = 1
        c_old = c
    return c, max_iter
```

Listing 2.2 `Roots.java`: the same three methods in Java, returning a `Result` record with the root and the number of iterations.

```java
import java.util.function.DoubleUnaryOperator;

/** Bracketing root finders: bisection, regula falsi and the Illinois variant. */
public class Roots {
    /** A root and the number of iterations that found it. */
    public record Result(double root, int iterations) { }

    public static final String HEADER = String.format(
            "%3s %11s %11s %11s %15s %12s", "k", "a", "b", "c", "f(c)",
            "|c - c_old|");

    static void check(double fa, double fb) {
        if (fa * fb > 0) {
            throw new IllegalArgumentException(
                    "f(a) and f(b) have the same sign: no bracket");
        }
    }

    static void row(int k, double a, double b, double c, double fc, double d) {
        System.out.printf("%3d %11.8f %11.8f %11.8f %15.4f %12.3e%n",
                          k, a, b, c, fc, d);
    }

    /** Halve [a, b] until two midpoints differ by less than tol. */
    public static Result bisection(DoubleUnaryOperator f, double a, double b,
                                   double tol, int maxIter, boolean trace) {
        double fa = f.applyAsDouble(a), fb = f.applyAsDouble(b);
        check(fa, fb);
        double c = a, cOld = a;
        for (int k = 1; k <= maxIter; k++) {
            c = a + (b - a) / 2;                 // safer than (a + b) / 2
            double fc = f.applyAsDouble(c);
            if (trace) {
                row(k, a, b, c, fc, Math.abs(c - cOld));
            }
            if (fc == 0 || Math.abs(c - cOld) < tol) {
                return new Result(c, k);
            }
            if (fa * fc < 0) {                   // the root is in [a, c]
                b = c;
                fb = fc;
            } else {                             // the root is in [c, b]
                a = c;
                fa = fc;
            }
            cOld = c;
```

Listing 2.2 (Continued) *Roots.java: the same three methods in Java, returning a Result record with the root and the number of iterations.*

```
        }
        return new Result(c, maxIter);
    }

    /**
     * False position: cut [a, b] where the chord through (a, f(a)) and
     * (b, f(b)) crosses zero. With illinois = true, an end point that stays
     * twice in a row has its function value halved.
     */
    public static Result regulaFalsi(DoubleUnaryOperator f, double a, double b,
                                     double tol, int maxIter, boolean trace,
                                     boolean illinois) {
        double fa = f.applyAsDouble(a), fb = f.applyAsDouble(b);
        check(fa, fb);
        double c = a, cOld = a;
        int side = 0;                               // which end stayed last time
        for (int k = 1; k <= maxIter; k++) {
            c = b - fb * (a - b) / (fa - fb);       // where the chord meets zero
            double fc = f.applyAsDouble(c);
            if (trace) {
                row(k, a, b, c, fc, Math.abs(c - cOld));
            }
            if (fc == 0 || Math.abs(c - cOld) < tol) {
                return new Result(c, k);
            }
            if (fa * fc < 0) {                      // root in [a, c]: b moves
                b = c;
                fb = fc;
                if (illinois && side == -1) {       // a stayed twice: halve f(a)
                    fa /= 2;
                }
                side = -1;
            } else {                                // root in [c, b]: a moves
                a = c;
                fa = fc;
                if (illinois && side == 1) {        // b stayed twice: halve f(b)
                    fb /= 2;
                }
                side = 1;
            }
            cOld = c;
        }
        return new Result(c, maxIter);
    }
}
```

The code follows the pseudocode line by line. When `trace` is on, each iteration prints one row: the iteration number k, the bracket $[a, b]$ before the step, the new point c, $f(c)$, and $|c - c_{\text{old}}|$. In the first row c_{old} is the start a, so the last column there shows how far the first point lies from a. Python returns the pair `(root, k)`; Java returns a `record Result(double root, int iterations)`, a compact class that Java has offered since version 16. A missing bracket is refused with an exception in both: `ValueError` in Python and `IllegalArgumentException` in Java.

The loan program, **Listing 2.3** and **Listing 2.4**, reads the price, the instalment and the number of months from the command line (with Dimas's plan as default), defines f of Equation (2.3), and runs the three methods on [0.01,0.05] with the tolerance 10^{-7}.

Listing 2.3 `loan_rate.py`: the hidden monthly rate of an instalment plan by three methods.

```python
"""The hidden monthly interest rate of an instalment plan.
Usage: python loan_rate.py [price] [instalment] [months]"""
import sys
from roots import HEADER, bisection, regula_falsi


def power(x, n):
    """x to the whole power n by repeated multiplication."""
    result = 1.0
    for _ in range(n):
        result *= x
    return result


def main():
    price = float(sys.argv[1]) if len(sys.argv) > 1 else 6000000.0
    pay = float(sys.argv[2]) if len(sys.argv) > 2 else 600000.0
    months = int(sys.argv[3]) if len(sys.argv) > 3 else 12

    def f(r):
        """Present value of all instalments at monthly rate r, minus price."""
        return pay * (1 - 1 / power(1 + r, months)) / r - price

    print(f"Price Rp {price:,.0f}, {months} instalments of Rp {pay:,.0f}")
    flat = (pay * months - price) / price / months
    print(f"Flat rate the seller quotes: {flat * 100:.4f} % per month")
    print("\nBisection on [0.01, 0.05]")
    print(HEADER)
    r, _ = bisection(f, 0.01, 0.05, trace=True)
    print("\nRegula falsi on [0.01, 0.05]")
    print(HEADER)
    regula_falsi(f, 0.01, 0.05, trace=True)
    print("\nIllinois on [0.01, 0.05]")
    print(HEADER)
    regula_falsi(f, 0.01, 0.05, trace=True, illinois=True)
    print(f"\nTrue monthly rate : {r * 100:.4f} %")
    print(f"Per year, compound: {(power(1 + r, 12) - 1) * 100:.2f} %")


if __name__ == "__main__":
    main()
```

Listing 2.4 `LoanRate.java`: the same program in Java.

```java
import java.util.Locale;
import java.util.function.DoubleUnaryOperator;

/**
 * The hidden monthly interest rate of an instalment plan.
 * Usage: java LoanRate [price] [instalment] [months]
 */
public class LoanRate {
    /** x to the whole power n by repeated multiplication. */
    static double power(double x, int n) {
        double result = 1.0;
        for (int i = 0; i < n; i++) {
            result *= x;
        }
        return result;
    }

    public static void main(String[] args) {
        Locale.setDefault(Locale.ROOT);
        double price = args.length > 0 ? Double.parseDouble(args[0]) : 6000000.0;
        double pay = args.length > 1 ? Double.parseDouble(args[1]) : 600000.0;
        int months = args.length > 2 ? Integer.parseInt(args[2]) : 12;

        // present value of all instalments at monthly rate r, minus the price
        DoubleUnaryOperator f =
                r -> pay * (1 - 1 / power(1 + r, months)) / r - price;

        System.out.printf("Price Rp %,.0f, %d instalments of Rp %,.0f%n",
                          price, months, pay);
        double flat = (pay * months - price) / price / months;
        System.out.printf("Flat rate the seller quotes: %.4f %% per month%n",
                          flat * 100);
        System.out.println("\nBisection on [0.01, 0.05]");
        System.out.println(Roots.HEADER);
        double r = Roots.bisection(f, 0.01, 0.05, 1e-7, 100, true).root();
        System.out.println("\nRegula falsi on [0.01, 0.05]");
        System.out.println(Roots.HEADER);
        Roots.regulaFalsi(f, 0.01, 0.05, 1e-7, 100, true, false);
        System.out.println("\nIllinois on [0.01, 0.05]");
        System.out.println(Roots.HEADER);
        Roots.regulaFalsi(f, 0.01, 0.05, 1e-7, 100, true, true);
        System.out.printf("%nTrue monthly rate : %.4f %%%n", r * 100);
        System.out.printf("Per year, compound: %.2f %%%n",
                          (power(1 + r, 12) - 1) * 100);
    }
}
```

Two small decisions keep the Java and Python outputs identical. The power $(1+r)^n$ is computed by repeated multiplication in a helper `power`, not with `Math.pow` or `**`: IEEE 754 guarantees that one multiplication gives the same result on every machine, while the library power functions of the two languages may differ in the last bit. And in Java the function `f` is a lambda that uses the local variables `pay`, `price` and `months`; Java allows this because they are never changed after they are set (*effectively final*).

2.6 Running It

```
$ python3 loan_rate.py          (or: java -cp out LoanRate)
```

```
Price Rp 6,000,000, 12 instalments of Rp 600,000
Flat rate the seller quotes: 1.6667 % per month

Bisection on [0.01, 0.05]
  k           a           b           c            f(c)  |c - c_old|
  1  0.01000000  0.05000000  0.03000000     -27597.6039    2.000e-02
  2  0.01000000  0.03000000  0.02000000     345204.7326    1.000e-02
  3  0.02000000  0.03000000  0.02500000     154658.7589    5.000e-03
  4  0.02500000  0.03000000  0.02750000      62522.1949    2.500e-03
  5  0.02750000  0.03000000  0.02875000      17213.5840    1.250e-03
  6  0.02875000  0.03000000  0.02937500      -5253.7705    6.250e-04
  7  0.02875000  0.02937500  0.02906250       5964.4147    3.125e-04
  8  0.02906250  0.02937500  0.02921875        351.4556    1.563e-04
  9  0.02921875  0.02937500  0.02929688      -2452.1233    7.813e-05
 10  0.02921875  0.02929688  0.02925781      -1050.5754    3.906e-05
 11  0.02921875  0.02925781  0.02923828       -349.6203    1.953e-05
 12  0.02921875  0.02923828  0.02922852          0.9026    9.766e-06
 13  0.02922852  0.02923828  0.02923340       -174.3626    4.883e-06
 14  0.02922852  0.02923340  0.02923096        -86.7310    2.441e-06
 15  0.02922852  0.02923096  0.02922974        -42.9145    1.221e-06
 16  0.02922852  0.02922974  0.02922913        -21.0060    6.104e-07
 17  0.02922852  0.02922913  0.02922882        -10.0517    3.052e-07
 18  0.02922852  0.02922882  0.02922867         -4.5746    1.526e-07
 19  0.02922852  0.02922867  0.02922859         -1.8360    7.629e-08

Regula falsi on [0.01, 0.05]
  k           a           b           c            f(c)  |c - c_old|
  1  0.01000000  0.05000000  0.03098945     -62719.8812    2.099e-02
  2  0.01000000  0.03098945  0.02937568      -5278.1637    1.614e-03
  3  0.01000000  0.02937568  0.02924082       -440.7396    1.349e-04
  4  0.01000000  0.02924082  0.02922957        -36.7789    1.125e-05
  5  0.01000000  0.02922957  0.02922863         -3.0690    9.391e-07
  6  0.01000000  0.02922863  0.02922855         -0.2561    7.836e-08

Illinois on [0.01, 0.05]
  k           a           b           c            f(c)  |c - c_old|
  1  0.01000000  0.05000000  0.03098945     -62719.8812    2.099e-02
  2  0.01000000  0.03098945  0.02937568      -5278.1637    1.614e-03
  3  0.01000000  0.02937568  0.02910782       4335.4506    2.679e-04
  4  0.02910782  0.02937568  0.02922862         -2.8128    1.208e-04
  5  0.02910782  0.02922862  0.02922854         -0.0015    7.832e-08

True monthly rate : 2.9229 %
Per year, compound: 41.30 %
```

The three tables tell the story of this chapter.

Bisection reproduces our hand computation in its first four rows and then needs exactly the 19 steps predicted by Equation (2.5). The last column halves at every row, as Equation (2.4) says it must. The column $f(c)$ does not fall steadily: at step 12 the midpoint 0.02922852 happens to lie within two millionths of a percentage point of the root, and f is less than one rupiah there, but the method does not notice and continues until the *bound* is below the tolerance.

Regula falsi needs only 6 steps. As predicted in Section 2.4.2, the left end $a = 0.01$ never moves, because the function is convex; but the function is so nearly straight between 1% and the root that the crawl is mild: each step reduces the error by a factor of about 12, exactly the rate $\lambda = 0.083$ predicted by Equation (2.7).

Illinois needs 5 steps. Steps 1 and 2 are the same as in regula falsi; after step 2, a has stayed twice, its value is halved, and step 3 jumps to the left of the root ($f > 0$). Now a moves at last, and step 4 lands within 10^{-7} of the root, step 5 within 10^{-10}.

The answer to Dimas's question is at the bottom: the true monthly rate is **2.9229%**, not 1.67%, and compounded over a year it is **41.30%**. The seller's figure is not false, but it measures something else. **Figure 2.2** compares the three methods by the distance of c_k from the root (computed here to full precision), on a logarithmic scale where linear convergence looks like a straight line.

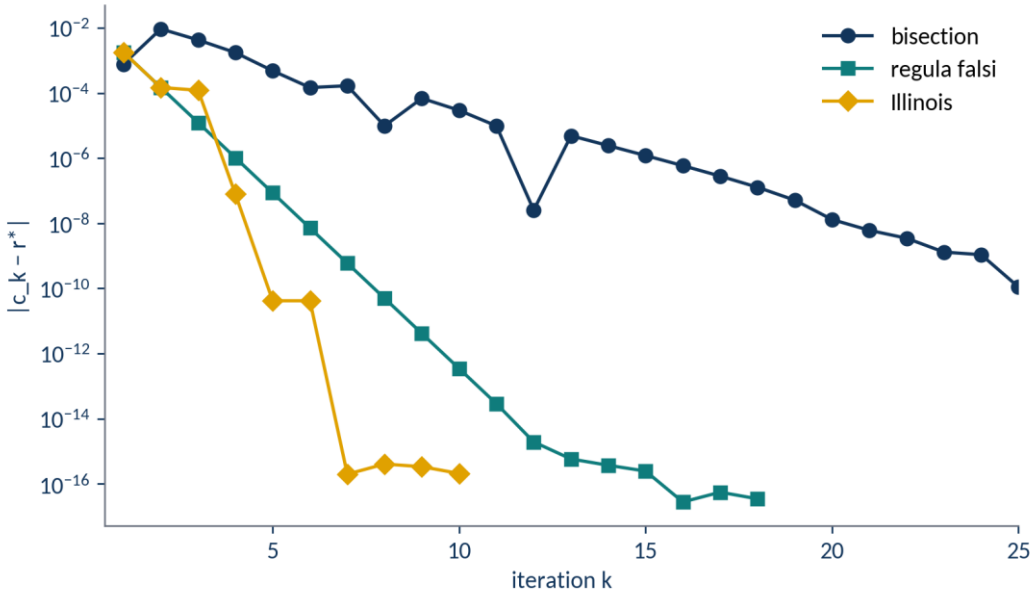


Figure 2.2 The error $|c_k - r^*|$ of the three bracketing methods on the loan function, continued past the tolerance until the error reaches the limit of double precision.

Bisection's error zigzags around a straight line of slope $\log_{10} 2$ per step. Regula falsi falls along a steeper straight line: it too is linear, with a better rate on this friendly function. Illinois bends downward, the sign of an order of convergence above one, and reaches the limit of double precision in 7 steps.

2.6.1 What 2.92% a month means in rupiah

A rate is abstract; a schedule is not. With the rate found, we can follow Dimas's debt month by month, exactly as a bank would. At the start of month 1 he owes Rp6,000,000. The interest for the month is $6{,}000{,}000 \times 0.029229 = \text{Rp}175{,}371$; of his instalment of Rp600,000, that much pays the interest and the rest, Rp424,629, pays off the debt, which falls to Rp5,575,371. In month 2 the interest is computed on the smaller debt, $5{,}575{,}371 \times 0.029229 = \text{Rp}162{,}960$, so more of the instalment, Rp437,040, reduces the debt. **Table 2.2** shows selected months.

Table 2.2 Dimas's debt month by month at the effective rate: the interest falls as the debt falls, and the twelfth instalment clears the debt exactly.

Month	Debt at start	Interest (2.9229%)	Principal repaid	Debt at end
1	6,000,000	175,371	424,629	5,575,371
2	5,575,371	162,960	437,040	5,138,331
3	5,138,331	150,186	449,814	4,688,517
6	3,749,062	109,580	490,420	3,258,642
9	2,234,380	65,308	534,692	1,699,687
12	582,961	17,039	582,961	0

Three things stand out. First, the debt reaches exactly zero with the twelfth instalment: that is what it *means* for 2.9229% to be the root of Equation (2.3), and it is a second, independent check of the answer. Second, the interest paid over the year adds up to Rp1,200,000, the same total as the flat computation, because the instalments are the same; only the *description* differs. Third, the interest in month 1 is Rp175,371, not Rp100,000: in the first months, when the debt is large, Dimas pays far more than the flat description suggests, and in the last months far less. The flat rate spreads the interest evenly over a debt that is not even, and that is why it understates the true cost of borrowing by almost half.

2.7 Where Regula Falsi Crawls

To see regula falsi at its worst, **Listing 2.5** and **Listing 2.6** apply all three methods to $f(x) = x^{10} - 1$ on [0,1.3], whose root is 1.

Listing 2.5 `stagnation.py`: the three methods on $x^{10} - 1$.

```
"""Where regula falsi crawls: f(x) = x^10 - 1 on [0, 1.3]."""
from roots import bisection, regula_falsi


def f(x):
    y = 1.0
    for _ in range(10):
        y *= x
    return y - 1


def main():
    for name, method in (("bisection", bisection),
                         ("regula falsi", regula_falsi),
                         ("Illinois", None)):
        if method is None:
            root, k = regula_falsi(f, 0.0, 1.3, illinois=True)
        else:
            root, k = method(f, 0.0, 1.3)
        print(f"{name:<13}: root {root:.9f} after {k:>3} iterations")


if __name__ == "__main__":
    main()
```

Listing 2.6 `Stagnation.java`: the same comparison in Java.

```java
import java.util.Locale;
import java.util.function.DoubleUnaryOperator;

/** Where regula falsi crawls: f(x) = x^10 - 1 on [0, 1.3]. */
public class Stagnation {
    public static void main(String[] args) {
        Locale.setDefault(Locale.ROOT);
        DoubleUnaryOperator f = x -> {
            double y = 1.0;
            for (int i = 0; i < 10; i++) {
                y *= x;
            }
            return y - 1;
        };
        Roots.Result[] results = {
            Roots.bisection(f, 0.0, 1.3, 1e-7, 100, false),
            Roots.regulaFalsi(f, 0.0, 1.3, 1e-7, 100, false, false),
            Roots.regulaFalsi(f, 0.0, 1.3, 1e-7, 100, false, true)};
        String[] names = {"bisection", "regula falsi", "Illinois"};
        for (int i = 0; i < 3; i++) {
            System.out.printf("%-13s: root %.9f after %3d iterations%n",
                              names[i], results[i].root(),
                              results[i].iterations());
        }
    }
}
```

```
bisection    : root 1.000000018 after  24 iterations
regula falsi : root 0.999999702 after  65 iterations
Illinois     : root 1.000000000 after  14 iterations
```

Bisection needs 24 steps, as Equation (2.5) predicts for a bracket of width 1.3 and $\tau = 10^{-7}$: $\log_2(1.3 \times 10^7) = 23.6$. Regula falsi needs 65, almost three times as many, and it *still* stops with an error of 3×10^{-7}, three times the tolerance: its steps had become so small that the change test was satisfied while the point was still crawling. This is the failure predicted in Section 2.4.2, and it is why regula falsi should not be used without the Illinois modification. Illinois needs 14 steps and is correct to all nine printed digits.

2.8 Testing It

Tests pin down what the code must do, so that a later change cannot break it unnoticed. **Listing 2.7** checks the three methods on $x^2 - 2$, whose root $\sqrt{2}$ we know, checks that bisection needs exactly the number of steps that Equation (2.5) predicts (for the bracket [0,2] and $\tau = 10^{-6}$, $\log_2(2 \times 10^6) = 20.9$, so 21), checks that Illinois beats plain regula falsi, and checks that a missing bracket is refused. **Listing 2.8** is the same in Java, as a plain program without a test framework.

Listing 2.7 `test_roots.py`: unit tests of the bracketing methods.

```python
"""Tests of the bracketing root finders: python -m unittest test_roots.py"""
import unittest
from roots import bisection, regula_falsi


def square_minus_two(x):
    return x * x - 2


class RootsTest(unittest.TestCase):
    def test_bisection_finds_sqrt2(self):
        root, _ = bisection(square_minus_two, 0.0, 2.0, tol=1e-10)
        self.assertAlmostEqual(root, 2 ** 0.5, places=9)

    def test_bisection_iteration_count(self):
        _, k = bisection(square_minus_two, 0.0, 2.0, tol=1e-6)
        self.assertEqual(k, 21)        # 2 / 2^k < 1e-6 first at k = 21

    def test_regula_falsi_and_illinois(self):
        r1, k1 = regula_falsi(square_minus_two, 0.0, 2.0, tol=1e-10)
        r2, k2 = regula_falsi(square_minus_two, 0.0, 2.0, tol=1e-10,
                              illinois=True)
        self.assertAlmostEqual(r1, 2 ** 0.5, places=9)
        self.assertAlmostEqual(r2, 2 ** 0.5, places=9)
        self.assertLess(k2, k1)

    def test_no_bracket(self):
        with self.assertRaises(ValueError):
            bisection(square_minus_two, 2.0, 3.0)


if __name__ == "__main__":
    unittest.main()
```

Listing 2.8 `RootsTest.java`: the same checks in Java.

```java
import java.util.function.DoubleUnaryOperator;

/** Tests of the bracketing root finders: java -cp out RootsTest */
public class RootsTest {
    static int failures = 0;

    static void check(boolean ok, String name) {
        System.out.println((ok ? "ok    " : "FAILED") + "  " + name);
        if (!ok) {
            failures++;
        }
    }

    public static void main(String[] args) {
        DoubleUnaryOperator f = x -> x * x - 2;
        double s2 = Math.sqrt(2);
        Roots.Result b = Roots.bisection(f, 0.0, 2.0, 1e-10, 100, false);
        check(Math.abs(b.root() - s2) < 1e-9, "bisection finds sqrt 2");
        check(Roots.bisection(f, 0.0, 2.0, 1e-6, 100, false).iterations() == 21,
              "bisection needs 21 iterations for tol 1e-6");
        Roots.Result r1 = Roots.regulaFalsi(f, 0.0, 2.0, 1e-10, 100, false, false);
        Roots.Result r2 = Roots.regulaFalsi(f, 0.0, 2.0, 1e-10, 100, false, true);
        check(Math.abs(r1.root() - s2) < 1e-9 && Math.abs(r2.root() - s2) < 1e-9,
              "regula falsi and Illinois find sqrt 2");
        check(r2.iterations() < r1.iterations(), "Illinois is faster");
        try {
            Roots.bisection(f, 2.0, 3.0, 1e-7, 100, false);
            check(false, "no bracket is refused");
        } catch (IllegalArgumentException e) {
            check(true, "no bracket is refused");
        }
        System.out.println(failures == 0 ? "all tests passed"
                                         : failures + " test(s) failed");
    }
}
```

```
$ python3 -m unittest test_roots.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
$ java -cp out RootsTest
```

```
ok      bisection finds sqrt 2
ok      bisection needs 21 iterations for tol 1e-6
ok      regula falsi and Illinois find sqrt 2
ok      Illinois is faster
ok      no bracket is refused
all tests passed
```

2.9 Comparing with SciPy

SciPy's module `scipy.optimize` offers several bracketing root finders. **Listing 2.9** asks three of them for Dimas's rate.

Listing 2.9 `compare_scipy.py`: the loan rate with SciPy's `bisect`, `brentq` and TOMS 748.

```python
"""The loan rate with SciPy's root finders (needs SciPy)."""
from scipy.optimize import bisect, brentq, root_scalar

PRICE, PAY, MONTHS = 6000000.0, 600000.0, 12


def f(r):
    return PAY * (1 - (1 + r) ** -MONTHS) / r - PRICE


def main():
    r1 = bisect(f, 0.01, 0.05, xtol=1e-7)
    r2, info = brentq(f, 0.01, 0.05, xtol=1e-12, full_output=True)
    sol = root_scalar(f, bracket=(0.01, 0.05), method="toms748")
    print(f"scipy bisect : {r1 * 100:.8f} %")
    print(f"scipy brentq : {r2 * 100:.8f} %  ({info.iterations} iterations)")
    print(f"toms748      : {sol.root * 100:.8f} %  ({sol.iterations} iterations)")


if __name__ == "__main__":
    main()
```

```
scipy bisect : 2.92285919 %
scipy brentq : 2.92285408 %  (7 iterations)
toms748      : 2.92285408 %  (3 iterations)
```

`bisect` returns 2.92285919%, exactly our bisection's 19th midpoint, because it is the same method with the same tolerance. `brentq` implements **Brent's method** [7], the standard bracketing root finder of numerical libraries since the 1970s: it keeps a bracket like bisection, but inside the bracket it tries fast steps (the secant step of Chapter 3 and inverse quadratic interpolation), and falls back to a bisection step whenever a fast step would leave the bracket or shrink it too slowly. It reached a tolerance of 10^{-12} in 7 iterations and agrees with our Illinois result to all printed digits. `toms748` is the algorithm of Alefeld, Potra and Shi [8], an even more refined bracketing method that needed only 3 iterations here. SciPy's documentation recommends `brentq` as the default choice for a bracketed scalar root [9]. The lesson for us is reassuring: our simple Illinois method reached the same answer in 5 steps, and our programs agree with the library.

2.10 Numerical Methods in Practice: Profitina

Profitina's PROFIT utility coin is designed to be deflationary: new coins are emitted on a *decaying* schedule, and coins used in the platform are burned. A decaying emission curve is a function $E(t)$ that gives the total number of coins emitted by time t; it rises quickly at first and then more and more slowly toward a ceiling. Questions about such a curve are naturally root-finding questions. *When will the cumulative emission reach half of the ceiling? At what month does the monthly emission fall below the monthly burn?* Each asks for the t at which some function $E(t) -$ level or $e(t) - b(t)$ is zero.

Such functions share the two properties that make bracketing methods the right tool. They are monotone, so a sign change brackets exactly one root, and the natural bracket is known in advance (from launch to a few decades later). A monotone function with a known bracket is precisely the case in which bisection's guarantee Equation (2.5) is most valuable: the computation cannot fail, and its cost is known before it starts. The same shape appears in the platform's pricing side, where a Pro subscription paid in PROFIT at a rate set by the PROFIT↔IDR oracle raises questions such as "at what coin price does a balance cover twelve renewals?": again a monotone function and a bracket.

2.11 Using It for Real

The program is ready for your own instalment plans; only the three numbers change. A motorbike priced Rp18,000,000 with 24 instalments of Rp950,000:

```
$ python3 loan_rate.py 18000000 950000 24
```

prints (among its tables) a true monthly rate of 1.9844% and a compound yearly rate of 26.59%, while its flat rate is only 1.1111% a month. Some practical advice for using it well:

- **Include every payment.** A down payment (*uang muka*) reduces the price financed: enter the price minus the down payment. An administration fee paid at the start does the same in the other direction: it is money you pay without receiving anything, so enter the price *minus* the fee as P, which raises the rate.
- **Check the bracket.** The program searches between 1% and 5% a month. If a plan has a lower or higher rate, both ends have the same sign and the program stops with "no bracket"; widen the bracket in the code, for example to [0.001,0.2].
- **Compare like with like.** Use the same program for every offer, including a bank's; the bank's advertised rate is usually already effective. A lower instalment over more months can still be the more expensive plan.

A quick check without a computer is also worth knowing. On average over the term, the debt is only about half of the price, so the true rate is roughly *twice* the flat rate; a slightly better rule of thumb is $r \approx \frac{2n}{n+1}$ times the flat rate. For Dimas, $\frac{24}{13} \times 1.667\% = 3.08\%$, a little above the true 2.92%; for the motorbike, $\frac{48}{25} \times 1.111\% = 2.13\%$ against the true 1.98%. The rule overestimates a little, because it ignores the fact that early instalments count more than late ones, but it is an excellent way to choose a bracket (from the flat rate to twice the flat rate) and to spot a result that cannot be right. If the program ever reports an effective rate *below* the flat rate, something in the input is wrong.

There is also a temptation to compare only the size of the monthly instalment. A smaller instalment feels lighter, and it is lighter for the daily cash flow of an ojek driver; but a smaller instalment over a longer term can mean much more interest in total. The program gives two numbers that complement each other: the effective rate, to compare the price of money between offers, and a schedule such as **Table 2.2**, to see how many rupiah of interest are paid each month. The final decision weighs both together with one's own finances, and it remains the user's.

The method is not limited to money. Any question of the form "what input gives this output?", where the output grows or falls steadily with the input, is a root: the fertilizer dose that brings a soil reading to a target, the temperature at which a battery reaches a given voltage, the depth of water in a tank of any shape (Exercise 2.6). Write $f(x) = \text{output}(x) - \text{target}$, find two inputs whose outputs lie on opposite sides of the target, and bisect.

2.12 Common Mistakes, and How to Avoid Them

- **Starting without a bracket.** If $f(a)$ and $f(b)$ have the same sign, bisection has nothing to work with. Tabulate f first, as in **Table 2.1**, and check the signs; the programs refuse a non-bracket rather than return nonsense.
- **Bracketing a pole, not a root.** A function that jumps from $+\infty$ to $-\infty$ changes sign without a root. Check that $|f(c)|$ becomes small at the end, not only that the bracket shrinks.

- **Evaluating f where it is undefined.** The loan function divides by r, so the bracket must not include $r = 0$ (Exercise 2.4 shows what happens if it does).
- **Stopping on a small $|f(c)|$ alone.** Whether $|f| < 1$ is small depends on the units of f. Stop on the change of c or on the bracket width, relative to the size of the root.
- **Trusting plain regula falsi's stopping test.** A crawling sequence changes little per step while still far from the root (Section 2.7). Use the Illinois modification, or bisection when a guarantee matters.
- **Computing the midpoint as $(a + b)/2$.** Use $a + (b - a)/2$, which cannot leave the bracket.
- **Reading the flat rate as the effective rate.** For an instalment plan they differ by almost a factor of two; compare offers by the effective rate.
- **Forgetting to check the answer.** A root finder returns a number even when the bracket, the function or the units were wrong. Put the root back into the problem, as the schedule of **Table 2.2** does, and see whether the story comes out right.

2.13 Chapter Summary

- A practical question "which input gives this output?" becomes a root-finding problem $f(x) = 0$; for an instalment plan, f is the present value of the instalments minus the price (Equation (2.2), Equation (2.3)).
- If f is continuous and $f(a)f(b) < 0$, the Intermediate Value Theorem guarantees a root in $[a, b]$: a bracket.
- Bisection halves the bracket at every step (**Pseudocode 2.1**). Its error is bounded by $(b_0 - a_0)/2^k$ (Equation (2.4)), so the number of steps for a tolerance is known in advance (Equation (2.5)): one binary digit, or about 0.3 decimal digits, per step.
- Regula falsi cuts the bracket where the chord crosses zero (Equation (2.6), **Pseudocode 2.2**); it is often faster, but when one end stays fixed it converges linearly with the rate Equation (2.7), which can be close to 1, and its stopping test can mislead.
- The Illinois modification halves the value at a stuck end (Equation (2.8)) and raises the order of convergence to about 1.44.
- For Dimas's phone, the three methods need 19, 6 and 5 steps and agree that the effective rate is 2.9229% a month (41.30% a year compounded), against a flat rate of 1.67%. SciPy's `bisect`, `brentq` and TOMS 748 give the same answer.

2.14 Exercises

1. By hand, carry out the first four bisection steps for $f(x) = x^2 - 2$ on $[1,2]$. Give each bracket and midpoint, and the error bound Equation (2.4) after step 4.
2. Prove Equation (2.4) by induction on k, and use it to show that bisection always converges for a continuous f and a bracket.
3. Carry out two steps of regula falsi for $f(x) = x^2 - 2$ on $[1,2]$ by hand. Which end stays fixed, and why?
4. What happens if the loan bracket is $[0,0.05]$ instead of $[0.01,0.05]$? Try it in Python and in Java and explain the two different behaviors. How can f be repaired at $r = 0$?
5. A shop offers a refrigerator of Rp6,000,000 in 24 instalments of Rp350,000. Before running the program, estimate from the flat rate and from Dimas's plan whether the effective rate is higher or lower than 2.92%. How many bisection steps does the bracket $[0.001,0.1]$ need for $\tau = 10^{-7}$? Then run the program.

6. A spherical water tank of radius $R = 1$ m holds, at depth h, the volume $V(h) = \pi h^2(3R - h)/3$. Write a program that finds the depth for 500, 1000, 2000, 3000 and 4000 litres with bisection, to be marked on a gauge.
7. Why does regula falsi keep the left end fixed on Dimas's function, but Illinois moves it? Answer by looking at the sign of $f(c)$ in the three tables of Section 2.6.

2.15 Solutions to the Exercises

1. $f(1) = -1$, $f(2) = 2$. Step 1: $c = 1.5$, $f = 0.25 > 0$, bracket [1,1.5]. Step 2: $c = 1.25$, $f = -0.4375$, bracket [1.25,1.5]. Step 3: $c = 1.375$, $f = -0.109375$, bracket [1.375,1.5]. Step 4: $c = 1.4375$, $f = 0.06640625$, bracket [1.375,1.4375]. The bound after step 4 is $(2-1)/2^4 = 0.0625$; indeed $|1.4375 - 1.41421| = 0.023$. The trace of `bisection(lambda x: x * x - 2, 1, 2, tol=1e-3, trace=True)` shows the same rows and stops after 10 steps at 1.41504, since $\log_2(1/10^{-3}) = 9.97$.

2. Let $[a_k, b_k]$ be the bracket after k steps, with $[a_0, b_0]$ the start. Claim: $b_k - a_k = (b_0 - a_0)/2^k$. For $k = 0$ this is true. If it holds for k, the next bracket is one half of $[a_k, b_k]$, so its width is $(b_0 - a_0)/2^{k+1}$. The midpoint $c_{k+1} = a_k + (b_k - a_k)/2$ and the root r^* both lie in $[a_k, b_k]$, and the midpoint is at most half the width from any point of it: $|c_{k+1} - r^*| \le (b_k - a_k)/2 = (b_0 - a_0)/2^{k+1}$. That is Equation (2.4) for $k + 1$. Each bracket contains a root by the Intermediate Value Theorem, because the method keeps the half with the sign change. Since the bound goes to zero, $c_k \to r^*$.

3. Step 1: $c = 2 - 2 \times (1-2)/(-1-2) = 2 - 2/3 = 1.33333$, $f(c) = -0.22222 < 0$, so a moves: bracket [1.33333,2]. Step 2: $c = 2 - 2 \times (1.33333 - 2)/(-0.22222 - 2) = 2 - 1.33333/2.22222 = 1.4$, $f = -0.04$, bracket [1.4,2]. The right end $b = 2$ stays fixed: $x^2 - 2$ is convex, and on a convex increasing function every chord crosses the axis to the left of the root, so $f(c) < 0$ every time.

4. At $r = 0$ the formula divides 0 by 0. In Python, `/ r` with `r = 0.0` raises `ZeroDivisionError` at once, when `bisection` evaluates $f(a)$. In Java, 0.0/0.0 is `NaN` without any error; the product `fa * fb` is then `NaN`, and `NaN > 0` is false, so the bracket check *passes*. The loop then compares signs with `NaN`: `fa * fc < 0` is always false, so a moves to c every time, and since `fa` becomes an ordinary number after the first step the method recovers by luck and returns the right root. Relying on such luck is a mistake. The repair is to use the limit at $r \to 0$: $(1 - (1+r)^{-n})/r \to n$, so define $f(0) = nA - P$ in the code (`if r == 0: return months * pay - price`).

5. The flat rate is $(24 \times 350{,}000 - 6{,}000{,}000)/6{,}000{,}000/24 = 1.667\%$, the same as Dimas's. But with a longer term the debt stays high for longer, so the same flat rate corresponds to a *lower* effective rate. The bracket needs $\log_2(0.099/10^{-7}) = 19.9$, so 20 steps. `python3 loan_rate.py 6000000 350000 24` gives 2.8874% a month and 40.72% a year: lower than 2.92%, as estimated.

6. **Listing 2.10** and **Listing 2.11** use the bracket $[0, 2R]$ (empty to full), where $V(h) - V_{\text{target}}$ changes sign for every target below the full volume $\frac{4}{3}\pi R^3 = 4{,}189$ litres, and a tolerance of 10^{-6} m.

Listing 2.10 `sphere_tank.py`: the depth of water in a spherical tank for given volumes.

```python
"""Exercise 2.6: how deep is the water in a spherical tank holding V litres?"""
import math
from roots import bisection

RADIUS = 1.0                 # metres


def volume(h):
    """Water volume in cubic metres when the depth is h metres."""
    return math.pi * h * h * (3 * RADIUS - h) / 3


def main():
    for litres in (500, 1000, 2000, 3000, 4000):
        target = litres / 1000.0
        h, k = bisection(lambda d: volume(d) - target, 0.0, 2 * RADIUS,
                         tol=1e-6)
        print(f"{litres:>5} L: depth {h * 100:6.2f} cm  ({k} iterations)")


if __name__ == "__main__":
    main()
```

Listing 2.11 `SphereTank.java`: the same in Java.

```java
import java.util.Locale;

/** Exercise 2.6: how deep is the water in a spherical tank holding V litres? */
public class SphereTank {
    static final double RADIUS = 1.0;            // metres

    /** Water volume in cubic metres when the depth is h metres. */
    static double volume(double h) {
        return Math.PI * h * h * (3 * RADIUS - h) / 3;
    }

    public static void main(String[] args) {
        Locale.setDefault(Locale.ROOT);
        for (int litres : new int[] {500, 1000, 2000, 3000, 4000}) {
            double target = litres / 1000.0;
            Roots.Result r = Roots.bisection(d -> volume(d) - target, 0.0,
                                             2 * RADIUS, 1e-6, 100, false);
            System.out.printf("%5d L: depth %6.2f cm  (%d iterations)%n",
                              litres, r.root() * 100, r.iterations());
        }
    }
}
```

```
  500 L: depth  43.11 cm  (21 iterations)
 1000 L: depth  63.55 cm  (21 iterations)
 2000 L: depth  96.99 cm  (21 iterations)
 3000 L: depth 129.70 cm  (21 iterations)
 4000 L: depth 174.37 cm  (21 iterations)
```

Every target needs 21 steps, because $\log_2(2/10^{-6}) = 20.9$ whatever the target: bisection's cost does not depend on the function. The marks are far from evenly spaced; the first 500 litres fill 43 cm, the last 1,000 litres (from 3,000 to 4,000) fill 45 cm, because the sphere is narrow at the bottom and top and wide in the middle. Chapter 3 marks the gauge of a lying cylinder with Newton's method.

7. In the regula falsi table every $f(c)$ is negative, so the root is always in $[a, c]$ and only b moves; the convex function puts every chord's zero to the right of the root. In the Illinois table, $f(c)$ is

negative in steps 1 and 2; after the second, $f(a)$ is halved, the next chord is flatter at the left end, its zero falls to the *left* of the root, $f(c) = +4{,}335 > 0$ in step 3, and a moves.

References

[1] Otoritas Jasa Keuangan, *Peraturan OJK Nomor 22 Tahun 2023 tentang Pelindungan Konsumen dan Masyarakat di Sektor Jasa Keuangan*, Dec. 2023. [Online]. Available: https://www.ojk.go.id/id/regulasi/Pages/POJK-22-Tahun-2023-Pelindungan-Konsumen-dan-Masyarakat-di-Sektor-Jasa-Keuangan.aspx

[2] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical Analysis*, 10th ed. Boston, MA, USA: Cengage Learning, 2016.

[3] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: SIAM, 2002.

[4] J.-L. Chabert, Ed., *A History of Algorithms: From the Pebble to the Microchip*. Berlin, Germany: Springer, 1999.

[5] S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers*, 8th ed. New York, NY, USA: McGraw-Hill Education, 2021.

[6] M. Dowell and P. Jarratt, "A modified regula falsi method for computing the root of an equation," *BIT Numerical Mathematics*, vol. 11, no. 2, pp. 168–174, 1971.

[7] R. P. Brent, *Algorithms for Minimization without Derivatives*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1973.

[8] G. E. Alefeld, F. A. Potra, and Y. Shi, "Algorithm 748: Enclosing zeros of continuous functions," *ACM Transactions on Mathematical Software*, vol. 21, no. 3, pp. 327–344, 1995.

[9] SciPy Developers, "Optimization and root finding (scipy.optimize)," *SciPy 1.18.1 Reference Guide*, 2026. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/optimize.html

Chapter 3

Root Finding II: Newton-Raphson, Secant and Fixed-Point Iteration

Bisection is safe but slow: one binary digit per step, whatever the function. When we know more about the function, above all its slope, we can do enormously better. This chapter builds the three classic *open* methods, which do not keep a bracket but follow the function's own shape toward the root: Newton-Raphson, which follows the tangent; the secant method, which follows a line through the last two points; and fixed-point iteration, which rewrites the equation as $x = g(x)$ and simply repeats. By the end of the chapter you will be able to derive Newton's method from Taylor's formula and prove that it doubles the number of correct digits at every step near a simple root; derive the secant method and explain why its order is the golden ratio 1.618; state when a fixed-point iteration converges and how fast; measure the order of convergence of any method from its errors; recognize the ways Newton fails (a flat tangent, a cycle, divergence) and guard against them; and use all three in Java and Python to mark the dipstick of a water truck, checking the result against SciPy.

3.1 Pak Harun's Water Truck

Every dry season, from July to October, the wells in many villages on the slopes of Gunungkidul run low, and households buy water by the tank. Pak Harun owns a small truck with a steel tank lying on its back: a cylinder 3.20 m long with a radius of 0.90 m, which holds a little over 8,000 litres when full. His customers do not buy the whole tank. A family fills its cistern with 1,000 or 2,000 litres, a food stall buys 500, a mosque before Friday prayers buys 3,000. At each stop Pak Harun must know how much water he has let out.

There is no meter on the truck. What there is, is a dipstick: a long wooden stick that he lowers vertically through the hatch on top of the tank until it touches the bottom, then pulls out to see how deep it is wet. If the tank stood upright, the depth would be proportional to the volume, and marking the stick would be trivial: 8,000 litres at 1.80 m, so 1,000 litres every 22.5 cm. But the tank lies on its side, and its cross-section is a circle. Near the bottom and the top, where the circle is narrow, a thousand litres make a deep layer; in the middle, where it is wide, the same thousand litres make a shallow one. The marks on the stick must be unevenly spaced, and nobody sold the truck with a marked stick.

Pak Harun's nephew, who studies at a vocational school in Wonosari, looked up the formula for the volume of water in a lying cylinder at a given depth (Section 3.2 derives it). That solved the wrong half of the problem. The formula gives the volume for a depth, but Pak Harun needs the depth for a volume: where to cut the notch for 1,000 litres, for 2,000, and so on. The formula contains the depth inside an inverse cosine *and* under a square root, and there is no way to solve it for the depth. The mark for 3,000 litres is the root of an equation.

Chapter 2 could find it by bisection in about 30 steps per mark. This chapter finds it in 4. The difference hardly matters for eight marks on a stick, but it matters enormously when a root must be found millions of times: in every step of a simulation, in every pixel of a rendered image, in

every prediction of a machine-learning model. Newton's method, the heart of this chapter, is one of the most used algorithms in all of computing.

3.2 The Idea in Plain Words

3.2.1 The volume of water in a lying tank

Look at the tank from its end. The water fills a *circular segment*: the part of a circle of radius R below a horizontal line at height h above the bottom. **Figure 3.1** shows it. The line is at distance $R - h$ below the center. Draw the two radii from the center to the points where the water surface meets the circle; each makes an angle θ with the downward vertical, where $\cos\theta = (R - h)/R$. The segment's area is the area of the "pie slice" between the two radii minus the triangle above the water:

- the slice spans the angle 2θ, so its area is $\frac{2\theta}{2\pi}\pi R^2 = R^2\theta$;
- the triangle has height $R - h$ and base equal to the surface width $w = 2\sqrt{R^2 - (R-h)^2} = 2\sqrt{2Rh - h^2}$, so its area is $(R - h)\sqrt{2Rh - h^2}$.

Multiplying the area by the length L of the tank gives the volume:

$$V(h) = L\left[R^2 \arccos\left(\frac{R-h}{R}\right) - (R-h)\sqrt{2Rh - h^2}\right] \tag{3.1}$$

Check the extremes. At $h = 0$, $\arccos 1 = 0$ and the square root is 0, so $V = 0$. At $h = R$ (half full), $\arccos 0 = \pi/2$ and $R - h = 0$, so $V = L\pi R^2/2$, half the full volume. At $h = 2R$, $\arccos(-1) = \pi$ and the square root is 0 again, so $V = L\pi R^2$. For Pak Harun, $L\pi R^2 = 3.20 \times \pi \times 0.81 = 8.143$ m^3, that is 8,143 litres.

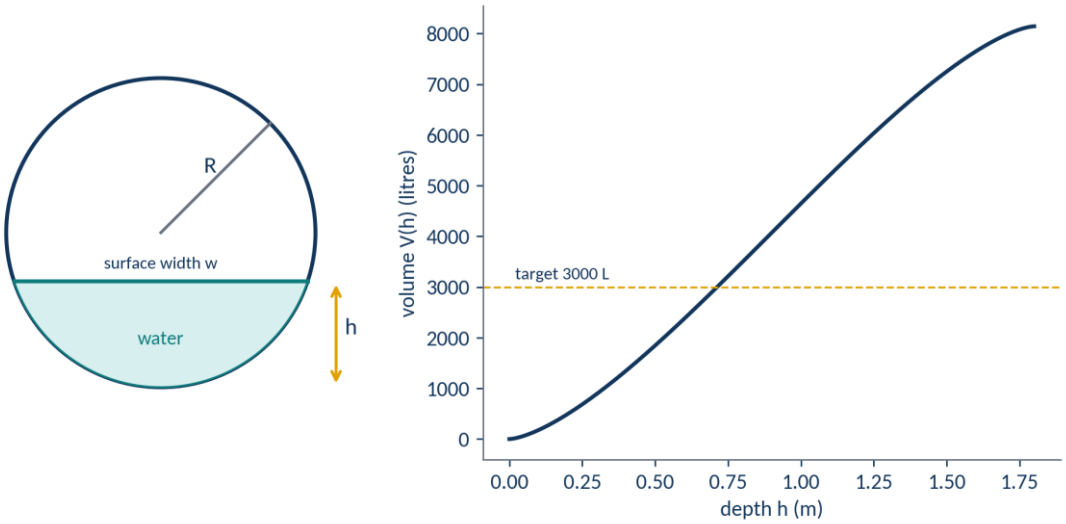


Figure 3.1 The water truck's tank seen from its end (left): the water fills a circular segment of depth h and surface width w. The volume Equation (3.1) as a function of the depth (right) is S-shaped, steepest at half depth.

We will also need the slope of this curve, the derivative $V'(h)$. Differentiating Equation (3.1) term by term takes a few lines of calculus, but there is a shortcut that explains the answer. If the depth rises by a tiny Δh, the extra water is a thin slab with the area of the water surface, $w \times L$, and thickness Δh. So $\Delta V \approx wL\,\Delta h$, and

$$V'(h) = L\,w = 2L\sqrt{2Rh - h^2} \tag{3.2}$$

The slope is largest at half depth, where the surface is widest ($w = 2R$), and zero at the bottom and the top, where the surface shrinks to a line. That is the S-shape in **Figure 3.1**.

3.2.2 Follow the tangent

To mark 3,000 litres, we need h with $V(h) = 3$ m^3, that is, the root of $f(h) = V(h) - 3$. Pak Harun's first guess is what the mark would be if the tank were a box: 3,000 of 8,143 litres is 36.8%, and 36.8% of the height 1.80 m is $h_0 = 0.6631$ m.

At this guess, Equation (3.1) gives $V(0.6631) = 2.7231$ m^3: 277 litres too few. How much deeper must the water be? The slope Equation (3.2) says that near h_0, each metre of depth adds $V'(0.6631) = 5.5570$ m^3. To add the missing 0.2769 m^3 we need about $0.2769/5.5570 = 0.0498$ m more. The new guess is $h_1 = 0.6631 + 0.0498 = 0.7130$ m.

That is the whole idea of Newton's method: at the current guess, pretend that the function is the straight line with the same value and the same slope, the *tangent*, and jump to where that line reaches the target. The next section shows that this simple idea is astonishingly powerful: the guess h_1 is already within 0.4 mm of the true mark, and the next step is within a hundred-thousandth of a millimetre.

3.3 Newton-Raphson

3.3.1 Derivation

Let x_k be the current guess for a root of f. Taylor's formula (Section 1.12) expands f around x_k:

$$f(x) = f(x_k) + f'(x_k)(x - x_k) + \frac{f''(\xi)}{2}(x - x_k)^2$$

Drop the last term and set the rest, the tangent line, to zero. Solving for x gives the next guess,

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{3.3}$$

the **Newton-Raphson** iteration. Isaac Newton described the idea in 1669 for polynomials, in a clumsier form; Joseph Raphson published the simpler form in 1690, and Thomas Simpson gave the general version for any function with a derivative in 1740 [1]. The method was known long before in special cases: the Babylonian method for square roots, attributed to Heron of Alexandria, is Newton's method for $f(x) = x^2 - a$, since $x - (x^2 - a)/(2x) = \frac{1}{2}(x + a/x)$.

Figure 3.2 draws two Newton steps for the 1,000-litre mark, where the curve bends more strongly than at 3,000 litres. Each step follows the tangent from the current point on the curve down to the axis.

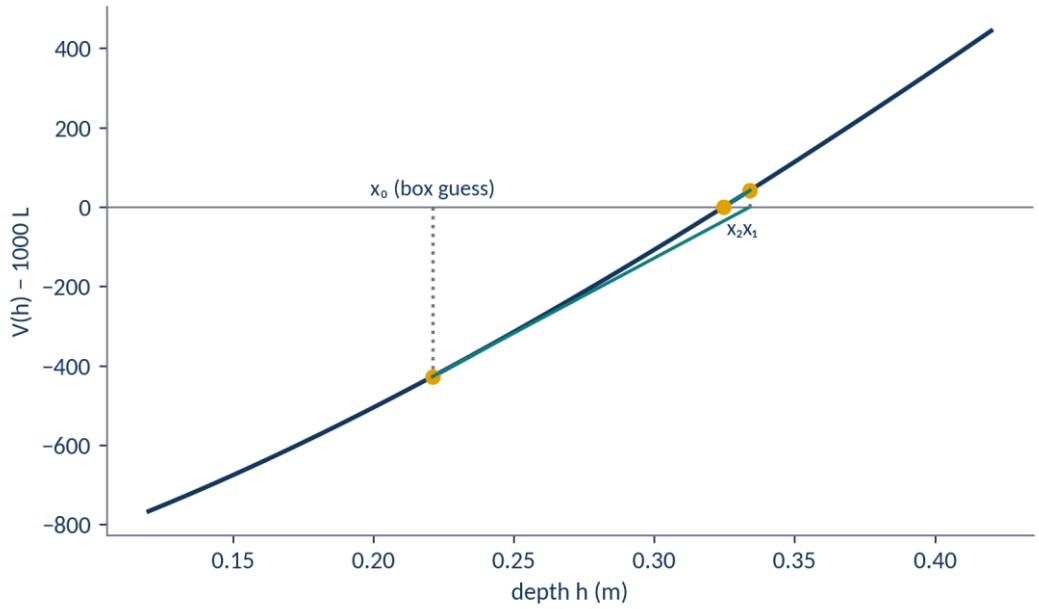


Figure 3.2 Newton's method for the 1,000-litre mark: from the box guess x_0 the tangent overshoots slightly to x_1, and the second tangent lands almost exactly on the root x_2.

Pseudocode 3.1 writes out the procedure with the safeguards that a real implementation needs: a test for a zero derivative, a tolerance on the change in x, and a maximum number of iterations.

Pseudocode 3.1 The Newton-Raphson method with its three safeguards.

```
NEWTON(f, df, x0, tol, maxIter)
 1  x = x0
 2  for k = 1 to maxIter
 3      d = df(x)
 4      if d == 0: error "f'(x) = 0: the tangent is flat"
 5      x_new = x - f(x) / d
 6      if |x_new - x| < tol: return x_new
 7      x = x_new
 8  error "no convergence in maxIter iterations"
```

The stopping test deserves a second look. For a quadratically convergent method, the change between two iterates is an excellent estimate of the error of the *older* one: $x_{k+1} - x_k = e_{k+1} - e_k \approx -e_k$, because e_{k+1} is much smaller than e_k. So when the test $|x_{k+1} - x_k| < \tau$ succeeds, the error of x_k is about τ, and the error of the returned x_{k+1} is about $C\tau^2$, far smaller. With $\tau = 10^{-9}$, as in the dipstick program, the returned depth is correct to the last digit of double precision. A method that converges linearly with rate λ has no such bonus: its change $|x_{k+1} - x_k| \approx (1 - \lambda)|e_k|$ can be much *smaller* than its error when λ is close to 1, which is how regula falsi fooled its stopping test in Section 2.7.

Unlike bisection, Newton's method returns an error rather than its last guess when it runs out of iterations. Bisection's last guess is always inside a bracket and therefore always useful; Newton's may be anywhere.

3.3.2 Why it is so fast: quadratic convergence

Let r be the root and $e_k = x_k - r$ the error of the k-th guess. Expand f around x_k again, this time evaluated *at the root*, where $f(r) = 0$:

$$0 = f(r) = f(x_k) + f'(x_k)(r - x_k) + \frac{f''(\xi_k)}{2}(r - x_k)^2$$

Divide by $f'(x_k)$ and rearrange: $x_k - \frac{f(x_k)}{f'(x_k)} - r = \frac{f''(\xi_k)}{2f'(x_k)}(x_k - r)^2$. The left side is exactly $x_{k+1} - r$, so

$$e_{k+1} = \frac{f''(\xi_k)}{2\,f'(x_k)}\,e_k^2 \approx C\,e_k^2, \qquad C = \frac{f''(r)}{2\,f'(r)} \tag{3.4}$$

The new error is proportional to the *square* of the old one [2]. If the error is 10^{-3}, the next is about $C \times 10^{-6}$, then $C^3 \times 10^{-12}$: the number of correct digits roughly *doubles* at every step. This is called **quadratic convergence**. Formally, a sequence converges with **order** p and asymptotic constant C if

$$\lim_{k\to\infty} \frac{|e_{k+1}|}{|\,e_k\,|^p} = |C| \tag{3.5}$$

Bisection's bound and regula falsi have $p = 1$ (linear), Illinois has $p \approx 1.44$, Newton has $p = 2$ at a *simple* root, one where $f'(r) \neq 0$.

Check it on the tank. At the 3,000-litre root $r = 0.712612$, the second derivative of V is $V''(r) = 2L(R - r)/\sqrt{2Rr - r^2} = 1.362$, and $V'(r) = 5.634$, so $C = 1.362/(2 \times 5.634) = 0.121$. The first error was $e_0 = 0.6631 - 0.7126 = -0.0495$; Equation (3.4) predicts $e_1 \approx 0.121 \times 0.0495^2 = 2.96 \times 10^{-4}$, and the program will find 3.56×10^{-4} (the prediction is only approximate while the error is still large). Then it predicts $e_2 \approx 0.121 \times (3.56 \times 10^{-4})^2 = 1.53 \times 10^{-8}$, and the program finds exactly 1.53×10^{-8}. The formula also says that the errors after the first step are *positive* (the curve bends upward here, $C > 0$): Newton approaches this root from above.

The price of this speed is the derivative. Every step needs $f'(x_k)$, which must be derived by hand (as we did in Equation (3.2)), approximated, or computed by *automatic differentiation*, a technique that Chapter 14 meets inside machine-learning libraries. And the guarantee is only *local*: Equation (3.4) holds when x_k is already close to a simple root. From a poor start, anything can happen.

3.3.3 When Newton fails

Four things can go wrong, and every user of Newton's method should know them [2], [3].

1. **A flat tangent.** If $f'(x_k) = 0$, the tangent is horizontal and never meets the axis; Equation (3.3) divides by zero. If $f'(x_k)$ is merely small, the step is huge and throws the guess far away. For the tank, the derivative is zero at the bottom and the top, so a guess of $h = 0$ or $h = 2R$ is fatal. The box guess avoids both.
2. **Divergence.** For $f(x) = x^{1/3}$, whose only root is 0, Newton's step is $x - x^{1/3}/\left(\frac{1}{3}x^{-2/3}\right) = x - 3x = -2x$: every step *doubles* the distance from the root, on alternating sides. The function is too steep at its root ($f'(0)$ is infinite) for the tangent to be a useful guide.
3. **A cycle.** For $f(x) = x^3 - 2x + 2$ starting at $x_0 = 0$: $f(0) = 2$, $f'(0) = -2$, so $x_1 = 1$; $f(1) = 1$, $f'(1) = 1$, so $x_2 = 0$, and the method bounces between 0 and 1 forever. The root, near -1.77, is never approached.

4. **A multiple root.** At a root where $f'(r) = 0$ too, such as $x = 1$ for $f(x) = (x-1)^2$, Equation (3.4) does not apply. Newton still converges, but only linearly: for a double root, each step halves the error, no better than bisection.

Section 3.9 runs the first three failures. The practical defenses are a good starting guess (from a table, a plot, the physics, or a few bisection steps), a maximum number of iterations, and, in a robust library, a *safeguard* that falls back to bisection when a Newton step leaves a known bracket, which is the idea behind Brent's method of Chapter 2.

3.4 The Secant Method

When the derivative is unavailable or expensive, it can be replaced by the slope of the line through the last two guesses, a *secant* of the curve:

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$$

Substituting into Equation (3.3) gives the **secant method**:

$$x_{k+1} = x_k - f(x_k)\,\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} \tag{3.6}$$

The formula is the same as the regula falsi formula of Section 2.4.1, with one crucial difference: the secant method always uses the *two most recent* points, whether or not they bracket the root, while regula falsi keeps a bracket. Giving up the bracket gives up the guarantee but removes the stuck end, and the result is much faster. A careful analysis, similar to the one for Newton but with two previous errors, shows that $e_{k+1} \approx C\, e_k\, e_{k-1}$ with $C = f''(r)/(2f'(r))$ [2]. If the order is p, then $e_{k+1} \sim e_k^{\,p}$ and $e_k \sim e_{k-1}^{\,p}$; substituting, $e_k^{\,p} \sim e_k \cdot e_k^{\,1/p}$, so $p = 1 + 1/p$, that is $p^2 - p - 1 = 0$, whose positive solution is the golden ratio:

$$p = \frac{1 + \sqrt{5}}{2} \approx 1.618 \tag{3.7}$$

The secant method is therefore *superlinear*: slower per step than Newton, but each step needs only one new function value, while a Newton step needs a value and a derivative. When the derivative costs as much as the function, two secant steps (order $1.618^2 = 2.618$) cost the same as one Newton step (order 2), and the secant method wins. It needs two starting points; a common choice is the guess x_0 and a point a little away from it, such as $1.01\,x_0$.

3.5 Fixed-Point Iteration

3.5.1 Rewriting $f(x) = 0$ as $x = g(x)$

A third way to look at root finding is to rewrite the equation $f(x) = 0$ as $x = g(x)$ for some function g, and then simply repeat:

$$x_{k+1} = g(x_k) \tag{3.8}$$

A number r with $g(r) = r$ is a **fixed point** of g: the iteration, once there, stays there. Newton's method is itself a fixed-point iteration with $g(x) = x - f(x)/f'(x)$, and many physical and

economic models are naturally written in this form ("next year's value is a function of this year's"). But any equation can be rewritten in many ways, and they behave very differently. Take $x^2 = 2$. Dividing by x gives $x = 2/x$; from $x_0 = 1$ this produces 2,1,2,1, … forever. Adding x to both sides of $x^2 - 2 = 0$ gives $x = x^2 + x - 2$, which runs off to infinity. But writing $x^2 - 1 = 1$ as $(x-1)(x+1) = 1$ gives $x = 1 + \frac{1}{1+x}$, and from $x_0 = 1$ this produces 1.5,1.4,1.41667,1.41379,1.41429, …, closing in on $\sqrt{2} = 1.41421$. Why?

3.5.2 When and how fast it converges

Subtract $r = g(r)$ from $x_{k+1} = g(x_k)$ and use the mean value theorem, which says that $g(x_k) - g(r) = g'(\xi_k)(x_k - r)$ for some ξ_k between x_k and r:

$$e_{k+1} = g'(\xi_k)\, e_k$$

So each step multiplies the error by about $g'(r)$. If $|g'(x)| \le K < 1$ near the root, the errors shrink at least as fast as K^k, and the iteration converges linearly with rate $|g'(r)|$; if $|g'(r)| > 1$ they grow, and it diverges. This is the heart of the **fixed-point theorem** [3]:

> **DEFINITION.** *Fixed-point theorem.* Let g be continuous on $[a,b]$ with $g(x) \in [a,b]$ for every x in $[a,b]$, and let g' exist on (a,b) with $|g'(x)| \le K < 1$ there. Then g has exactly one fixed point r in $[a,b]$, and for any x_0 in $[a,b]$ the iteration Equation (3.8) converges to r with

$$|x_k - r| \le \frac{K^k}{1-K}\,|x_1 - x_0|$$

> (3.9)

For $g(x) = 1 + 1/(1+x)$, $g'(x) = -1/(1+x)^2$, and at the root $g'(\sqrt{2}) = -1/(1+\sqrt{2})^2 = -0.1716$. Each step multiplies the error by -0.17: the sign alternates (the iterates hop from one side of the root to the other) and the error shrinks by a factor of almost 6 per step, about 0.77 decimal digits. For $g(x) = 2/x$, $g'(\sqrt{2}) = -2/2 = -1$, exactly on the border: the errors neither shrink nor grow, which is the endless 2,1,2,1. For $g(x) = x^2 + x - 2$, $g'(\sqrt{2}) = 2\sqrt{2} + 1 = 3.83 > 1$: divergence.

The theorem is useful before any computing is done, because it predicts the number of steps. For $g(x) = 1 + 1/(1+x)$ on $[1,2]$: g is decreasing, $g(1) = 1.5$ and $g(2) = 1.3333$, so g maps $[1,2]$ into $[1.3333,1.5]$, well inside $[1,2]$. On $[1,2]$, $|g'(x)| = 1/(1+x)^2 \le 1/4$, so $K = 0.25$. From $x_0 = 1$, $x_1 = 1.5$ and $|x_1 - x_0| = 0.5$. By Equation (3.9) the error after k steps is at most $0.25^k \times 0.5/0.75$, which is below 10^{-6} once $0.25^k < 1.5 \times 10^{-6}$, that is $k \ge \log(1.5 \times 10^{-6})/\log 0.25 = 9.7$: ten steps are guaranteed to be enough. The bound is pessimistic, because it uses the worst slope on the whole interval (0.25) instead of the slope at the root (0.17); the actual iteration reaches 10^{-6} after 8 steps, as Section 3.8 will show.

The same analysis explains Newton's speed. For $g(x) = x - f(x)/f'(x)$, the quotient rule gives $g'(x) = f(x)f''(x)/f'(x)^2$, which is *zero* at a simple root, where $f(r) = 0$. A fixed-point iteration whose g' vanishes at the fixed point converges faster than linearly: the error term $g'(\xi_k)e_k$ becomes $\frac{1}{2}g''(\xi_k)e_k^2$, which is Equation (3.4) again.

3.6 Building It in Java and Python

The three methods go into one module, **Listing 3.1** and **Listing 3.2**, next to the bracketing methods of Chapter 2. With `trace` on, each iteration prints the new iterate, the function value there, and the change from the previous iterate.

Listing 3.1 `roots2.py`: Newton-Raphson, secant and fixed-point iteration.

```
"""Open root finders: Newton-Raphson, secant and fixed-point iteration."""

HEADER = f"{'k':>3} {'x':>18} {'f(x)':>14} {'|x - x_old|':>12}"


def _row(k, x, fx, dx):
    print(f"{k:>3} {x:>18.12f} {fx:>14.6e} {dx:>12.3e}")


def newton(f, df, x0, tol=1e-10, max_iter=50, trace=False):
    """x_new = x - f(x) / f'(x) until two iterates differ by less than tol.
    Returns (root, iterations); raises ArithmeticError if f'(x) is 0."""
    x = x0
    for k in range(1, max_iter + 1):
        d = df(x)
        if d == 0:
            raise ArithmeticError(f"f'(x) = 0 at x = {x}")
        x_new = x - f(x) / d
        if trace:
            _row(k, x_new, f(x_new), abs(x_new - x))
        if abs(x_new - x) < tol:
            return x_new, k
        x = x_new
    raise ArithmeticError(f"no convergence in {max_iter} iterations")


def secant(f, x0, x1, tol=1e-10, max_iter=50, trace=False):
    """Newton with the derivative replaced by the slope through the last
    two iterates."""
    f0, f1 = f(x0), f(x1)
    for k in range(1, max_iter + 1):
        if f1 == f0:
            raise ArithmeticError("flat secant: f(x0) = f(x1)")
        x2 = x1 - f1 * (x1 - x0) / (f1 - f0)
        if trace:
            _row(k, x2, f(x2), abs(x2 - x1))
        if abs(x2 - x1) < tol:
            return x2, k
        x0, f0 = x1, f1
        x1, f1 = x2, f(x2)
    raise ArithmeticError(f"no convergence in {max_iter} iterations")
```

Listing 3.1 (Continued) *roots2.py: Newton-Raphson, secant and fixed-point iteration.*

```
def fixed_point(g, x0, tol=1e-10, max_iter=200, trace=False):
    """x_new = g(x) until two iterates differ by less than tol."""
    x = x0
    for k in range(1, max_iter + 1):
        x_new = g(x)
        if trace:
            _row(k, x_new, x_new - g(x_new), abs(x_new - x))
        if abs(x_new - x) < tol:
            return x_new, k
        x = x_new
    raise ArithmeticError(f"no convergence in {max_iter} iterations")
```

Listing 3.2 `Roots2.java`: the same three methods in Java.

```java
import java.util.function.DoubleUnaryOperator;

/** Open root finders: Newton-Raphson, secant and fixed-point iteration. */
public class Roots2 {
    /** A root and the number of iterations that found it. */
    public record Result(double root, int iterations) { }

    public static final String HEADER = String.format("%3s %18s %14s %12s",
            "k", "x", "f(x)", "|x - x_old|");

    static void row(int k, double x, double fx, double dx) {
        System.out.printf("%3d %18.12f %14.6e %12.3e%n", k, x, fx, dx);
    }

    /** x_new = x - f(x) / f'(x) until two iterates differ by less than tol. */
    public static Result newton(DoubleUnaryOperator f, DoubleUnaryOperator df,
                                double x0, double tol, int maxIter,
                                boolean trace) {
        double x = x0;
        for (int k = 1; k <= maxIter; k++) {
            double d = df.applyAsDouble(x);
            if (d == 0) {
                throw new ArithmeticException("f'(x) = 0 at x = " + x);
            }
            double xNew = x - f.applyAsDouble(x) / d;
            if (trace) {
                row(k, xNew, f.applyAsDouble(xNew), Math.abs(xNew - x));
            }
            if (Math.abs(xNew - x) < tol) {
                return new Result(xNew, k);
            }
            x = xNew;
        }
        throw new ArithmeticException("no convergence in " + maxIter
                                      + " iterations");
    }
```

Listing 3.2 (Continued) *Roots2.java: the same three methods in Java.*

```
    /** Newton with the derivative replaced by the slope of the secant. */
    public static Result secant(DoubleUnaryOperator f, double x0, double x1,
                                double tol, int maxIter, boolean trace) {
        double f0 = f.applyAsDouble(x0), f1 = f.applyAsDouble(x1);
        for (int k = 1; k <= maxIter; k++) {
            if (f1 == f0) {
                throw new ArithmeticException("flat secant: f(x0) = f(x1)");
            }
            double x2 = x1 - f1 * (x1 - x0) / (f1 - f0);
            if (trace) {
                row(k, x2, f.applyAsDouble(x2), Math.abs(x2 - x1));
            }
            if (Math.abs(x2 - x1) < tol) {
                return new Result(x2, k);
            }
            x0 = x1;
            f0 = f1;
            x1 = x2;
            f1 = f.applyAsDouble(x2);
        }
        throw new ArithmeticException("no convergence in " + maxIter
                                      + " iterations");
    }

    /** x_new = g(x) until two iterates differ by less than tol. */
    public static Result fixedPoint(DoubleUnaryOperator g, double x0,
                                    double tol, int maxIter, boolean trace) {
        double x = x0;
        for (int k = 1; k <= maxIter; k++) {
            double xNew = g.applyAsDouble(x);
            if (trace) {
                row(k, xNew, xNew - g.applyAsDouble(xNew), Math.abs(xNew - x));
            }
            if (Math.abs(xNew - x) < tol) {
                return new Result(xNew, k);
            }
            x = xNew;
        }
        throw new ArithmeticException("no convergence in " + maxIter
                                      + " iterations");
    }
}
```

The failures of Section 3.3.3 are handled by exceptions rather than by returning a doubtful number: `ArithmeticError` in Python (the parent of `ZeroDivisionError`) and `ArithmeticException` in Java, with a message that says whether the tangent was flat, the secant was flat, or the iteration ran out of steps. The fixed-point trace prints $x_{\text{new}} - g(x_{\text{new}})$ in the $f(x)$ column, which is the natural residual of the equation $x = g(x)$.

The dipstick program, **Listing 3.3** and **Listing 3.4**, defines the volume Equation (3.1) and its derivative Equation (3.2), prints the Newton iterations for 3,000 litres from the box guess, and then computes all eight marks from 1,000 to 8,000 litres with both Newton and secant, counting the iterations of each.

Listing 3.3 `dipstick.py`: the depth marks of a lying cylindrical tank by Newton-Raphson and secant.

```python
"""Marks for the dipstick of a water truck: a lying cylindrical tank.
For every volume V we solve V(h) = V for the depth h with Newton-Raphson."""
import math
from roots2 import HEADER, newton, secant

R = 0.90                      # radius of the tank, metres
L = 3.20                      # length of the tank, metres
FULL = math.pi * R * R * L    # cubic metres


def volume(h):
    """Water volume (m^3) at depth h: the circular segment times the length."""
    return L * (R * R * math.acos((R - h) / R)
                - (R - h) * math.sqrt(2 * R * h - h * h))


def dvolume(h):
    """dV/dh: the width of the water surface times the length."""
    return 2 * L * math.sqrt(2 * R * h - h * h)


def depth(litres, trace=False):
    target = litres / 1000.0
    guess = 2 * R * target / FULL           # as if the tank were a box
    return newton(lambda h: volume(h) - target, dvolume, guess, tol=1e-9,
                  trace=trace)


def main():
    print(f"Tank: radius {R} m, length {L} m, full {FULL * 1000:.1f} L")
    print("\nNewton-Raphson for 3000 L, starting from the box guess")
    print(HEADER)
    depth(3000, trace=True)
    print(f"\n{'volume':>7} {'depth':>9} {'Newton':>7} {'secant':>7}")
    for litres in range(1000, 8001, 1000):
        h, k1 = depth(litres)
        target = litres / 1000.0
        g = 2 * R * target / FULL
        _, k2 = secant(lambda d: volume(d) - target, g, g * 1.01, tol=1e-9)
        print(f"{litres:>5} L {h * 100:>6.2f} cm {k1:>7} {k2:>7}")


if __name__ == "__main__":
    main()
```

Listing 3.4 `Dipstick.java`: the same program in Java.

```
import java.util.Locale;

/**
 * Marks for the dipstick of a water truck: a lying cylindrical tank. For
 * every volume V we solve V(h) = V for the depth h with Newton-Raphson.
 */
public class Dipstick {
    static final double R = 0.90;                 // radius of the tank, metres
    static final double L = 3.20;                 // length of the tank, metres
    static final double FULL = Math.PI * R * R * L;   // cubic metres

    /** Water volume (m^3) at depth h: the circular segment times the length. */
    static double volume(double h) {
        return L * (R * R * Math.acos((R - h) / R)
                    - (R - h) * Math.sqrt(2 * R * h - h * h));
    }

    /** dV/dh: the width of the water surface times the length. */
    static double dvolume(double h) {
        return 2 * L * Math.sqrt(2 * R * h - h * h);
    }

    static Roots2.Result depth(double litres, boolean trace) {
        double target = litres / 1000.0;
        double guess = 2 * R * target / FULL;     // as if the tank were a box
        return Roots2.newton(h -> volume(h) - target, Dipstick::dvolume, guess,
                             1e-9, 50, trace);
    }

    public static void main(String[] args) {
        Locale.setDefault(Locale.ROOT);
        System.out.printf("Tank: radius %s m, length %s m, full %.1f L%n",
                          R, L, FULL * 1000);
        System.out.println("\nNewton-Raphson for 3000 L, starting from the box "
                           + "guess");
        System.out.println(Roots2.HEADER);
        depth(3000, true);
        System.out.printf("%n%7s %9s %7s %7s%n", "volume", "depth", "Newton",
                          "secant");
        for (int litres = 1000; litres <= 8000; litres += 1000) {
            Roots2.Result r = depth(litres, false);
            double target = litres / 1000.0;
            double g = 2 * R * target / FULL;
            Roots2.Result s = Roots2.secant(d -> volume(d) - target, g,
                                            g * 1.01, 1e-9, 50, false);
            System.out.printf("%5d L %6.2f cm %7d %7d%n", litres,
```

Listing 3.4 (Continued) *Dipstick.java: the same program in Java.*

```
                              r.root() * 100, r.iterations(), s.iterations());
        }
    }
}
```

The function `depth` builds $f(h) = V(h) - V_{\text{target}}$ as a small anonymous function (a `lambda` in Python, a lambda expression in Java) and hands it, with the derivative, to `newton`. The derivative of f is the derivative of V, because subtracting a constant does not change a slope. The secant method starts from the box guess and a point 1% above it. The tolerance 10^{-9} m is a millionth of a millimetre, far beyond what a wooden stick needs, but it costs almost nothing, as we are about to see.

3.7 Running It

```
$ python3 dipstick.py          (or: java -cp out Dipstick)
```

```
Tank: radius 0.9 m, length 3.2 m, full 8143.0 L

Newton-Raphson for 3000 L, starting from the box guess
  k                  x           f(x)  |x - x_old|
  1     0.712968084725   2.005405e-03    4.982e-02
  2     0.712612153675   8.618449e-08    3.559e-04
  3     0.712612138378   0.000000e+00    1.530e-08
  4     0.712612138378   0.000000e+00    0.000e+00

 volume     depth  Newton  secant
 1000 L  32.47 cm       5       6
 2000 L  52.96 cm       4       5
 3000 L  71.26 cm       4       4
 4000 L  88.76 cm       3       3
 5000 L 106.21 cm       4       4
 6000 L 124.33 cm       4       5
 7000 L 144.36 cm       4       6
 8000 L 171.37 cm       5       7
```

The trace for 3,000 litres shows quadratic convergence at work. The first step moves 4.98 cm from the box guess to 71.2968 cm, exactly as we computed by hand in Section 3.2.2. The second step corrects by 0.36 mm, the third by 0.0000153 mm, and the fourth by nothing at all: the iterate no longer changes in double precision, and f is exactly zero there. The changes 4.98×10^{-2}, 3.56×10^{-4}, 1.53×10^{-8} show the digits doubling: 1, 3, 8 correct digits, and then all 16.

The table is Pak Harun's dipstick. The marks are clearly uneven: the first 1,000 litres fill 32.47 cm, the next 1,000 only 20.49 cm (to 52.96 cm), the middle 1,000 litres (from 4,000 to 5,000) only 17.45 cm, and the last 1,000 before 8,000 litres again 27.01 cm. Newton needs 3 to 5 iterations for every mark, the secant method 3 to 7. Both need the most steps near the bottom and near the top, where the box guess is poorest and the curve bends most, and the fewest near 4,000 litres, where the tank is almost half full and the box guess is almost exact. Bisection with the bracket [0,1.8] and the same tolerance would need $\log_2(1.8/10^{-9}) = 30.7$, so 31 iterations for every mark.

In practice Pak Harun needs the marks to the millimetre: 32.5, 53.0, 71.3, 88.8, 106.2, 124.3, 144.4 and 171.4 cm, measured from the bottom of the tank. His nephew printed the table, taped it to a new stick, and cut a notch at each mark. The first time they checked it, by filling a 1,000-litre cistern whose volume they knew, the water level on the stick fell by exactly one mark.

3.8 Measuring the Order of Convergence

The definition Equation (3.5) can be turned into a measurement. If $|e_{k+1}| \approx |C| \; | \; e_k \; |^p$ and $|e_k| \approx |C| \; | \; e_{k-1} \; |^p$, dividing and taking logarithms eliminates C:

$$p \approx \frac{\ln(|e_{k+1}|/|e_k|)}{\ln(|e_k|/|e_{k-1}|)} \tag{3.10}$$

Listing 3.5 and **Listing 3.6** compute $\sqrt{2}$ with three methods (Newton on $x^2 - 2$, secant on $x^2 - 2$, and the fixed-point iteration $x = 1 + 1/(1 + x)$), print the error of every iterate against the library's $\sqrt{2}$, the ratio $|e_k|/|e_{k-1}|$, and the estimate Equation (3.10).

Listing 3.5 `convergence.py`: $\sqrt{2}$ by three methods, with the measured ratio and order of convergence.

```python
"""How fast the error falls: sqrt(2) by three methods, with the order p
estimated from three consecutive errors."""
import math

ROOT = math.sqrt(2.0)


def report(name, xs):
    print(name)
    print(f"{'k':>3} {'x_k':>18} {'error':>11} {'ratio':>8} {'order p':>8}")
    errs = [abs(x - ROOT) for x in xs]
    for k, (x, e) in enumerate(zip(xs, errs), 1):
        ratio, p = "", ""
        if k >= 2 and e > 0 and errs[k - 2] > 0:
            ratio = f"{e / errs[k - 2]:.4f}"          # e_k / e_(k-1)
        if k >= 3 and e > 0 and errs[k - 2] > 0 and errs[k - 3] > 0:
            top = math.log(e / errs[k - 2])
            p = f"{top / math.log(errs[k - 2] / errs[k - 3]):.3f}"
        print(f"{k:>3} {x:>18.14f} {e:>11.3e} {ratio:>8} {p:>8}")
    print()


def main():
    xs = []
    x = 1.0
    for _ in range(5):                    # Newton for x^2 - 2
        x = x - (x * x - 2) / (2 * x)
        xs.append(x)
    report("Newton-Raphson, x0 = 1", xs)
    xs, a, b = [], 1.0, 2.0
    for _ in range(7):                    # secant for x^2 - 2
        c = b - (b * b - 2) * (b - a) / ((b * b - 2) - (a * a - 2))
        xs.append(c)
        a, b = b, c
        if a == b:
            break
    report("Secant, x0 = 1, x1 = 2", xs)
    xs, x = [], 1.0
    for _ in range(8):                    # fixed point x = 1 + 1 / (1 + x)
        x = 1 + 1 / (1 + x)
        xs.append(x)
    report("Fixed point x = 1 + 1/(1 + x), x0 = 1", xs)


if __name__ == "__main__":
    main()
```

Listing 3.6 `Convergence.java`: the same measurement in Java.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;

/**
 * How fast the error falls: sqrt(2) by three methods, with the order p
 * estimated from three consecutive errors.
 */
public class Convergence {
    static final double ROOT = Math.sqrt(2.0);

    static void report(String name, List<Double> xs) {
        System.out.println(name);
        System.out.printf("%3s %18s %11s %8s %8s%n", "k", "x_k", "error",
                          "ratio", "order p");
        double[] e = new double[xs.size()];
        for (int i = 0; i < e.length; i++) {
            e[i] = Math.abs(xs.get(i) - ROOT);
        }
        for (int i = 0; i < e.length; i++) {
            String ratio = "", p = "";
            if (i >= 1 && e[i] > 0 && e[i - 1] > 0) {
                ratio = String.format("%.4f", e[i] / e[i - 1]);  // e_k / e_(k-1)
            }
            if (i >= 2 && e[i] > 0 && e[i - 1] > 0 && e[i - 2] > 0) {
                double top = Math.log(e[i] / e[i - 1]);
                p = String.format("%.3f", top / Math.log(e[i - 1] / e[i - 2]));
            }
            System.out.printf("%3d %18.14f %11.3e %8s %8s%n", i + 1, xs.get(i),
                              e[i], ratio, p);
        }
        System.out.println();
    }

    public static void main(String[] args) {
        Locale.setDefault(Locale.ROOT);
        List<Double> xs = new ArrayList<>();
        double x = 1.0;
        for (int i = 0; i < 5; i++) {            // Newton for x^2 - 2
            x = x - (x * x - 2) / (2 * x);
            xs.add(x);
        }
        report("Newton-Raphson, x0 = 1", xs);
        xs = new ArrayList<>();
        double a = 1.0, b = 2.0;
        for (int i = 0; i < 7; i++) {            // secant for x^2 - 2
```

Listing 3.6 (Continued) *Convergence.java: the same measurement in Java.*

```
            double c = b - (b * b - 2) * (b - a) / ((b * b - 2) - (a * a - 2));
            xs.add(c);
            a = b;
            b = c;
            if (a == b) {
                break;
            }
        }
        report("Secant, x0 = 1, x1 = 2", xs);
        xs = new ArrayList<>();
        x = 1.0;
        for (int i = 0; i < 8; i++) {            // fixed point x = 1 + 1/(1 + x)
            x = 1 + 1 / (1 + x);
            xs.add(x);
        }
        report("Fixed point x = 1 + 1/(1 + x), x0 = 1", xs);
    }
}
```

```
Newton-Raphson, x0 = 1
  k                x_k        error     ratio  order p
  1   1.50000000000000   8.579e-02
  2   1.41666666666667   2.453e-03   0.0286
  3   1.41421568627451   2.124e-06   0.0009    1.984
  4   1.41421356237469   1.595e-12   0.0000    2.000
  5   1.41421356237310   0.000e+00

Secant, x0 = 1, x1 = 2
  k                x_k        error     ratio  order p
  1   1.33333333333333   8.088e-02
  2   1.40000000000000   1.421e-02   0.1757
  3   1.41463414634146   4.206e-04   0.0296    2.025
  4   1.41421143847487   2.124e-06   0.0050    1.502
  5   1.41421356205732   3.158e-10   0.0001    1.667
  6   1.41421356237310   2.220e-16   0.0000    1.607
  7   1.41421356237310   0.000e+00

Fixed point x = 1 + 1/(1 + x), x0 = 1
  k                x_k        error     ratio  order p
  1   1.50000000000000   8.579e-02
  2   1.40000000000000   1.421e-02   0.1657
  3   1.41666666666667   2.453e-03   0.1726    0.977
  4   1.41379310344828   4.205e-04   0.1714    1.004
  5   1.41428571428571   7.215e-05   0.1716    0.999
  6   1.41420118343195   1.238e-05   0.1716    1.000
  7   1.41421568627451   2.124e-06   0.1716    1.000
  8   1.41421319796954   3.644e-07   0.1716    1.000
```

The measured orders match the theory. Newton gives 1.984 and 2.000: quadratic, Equation (3.4). Its ratio column falls toward zero, the mark of any method faster than linear. The secant method gives 2.025, 1.502, 1.667 and 1.607: noisy at first, then settling toward the golden ratio 1.618 of Equation (3.7). The fixed-point iteration gives an order of exactly 1.000 and a ratio that settles at 0.1716, precisely the $|g'(\sqrt{2})|$ of Section 3.5.2. Its error falls by the same factor at each step, forever, while Newton's error collapses: Newton reaches full precision in 5 steps, the secant method in 6, and the fixed-point iteration would need about 20. **Figure 3.3** shows the three together with bisection on a logarithmic scale.

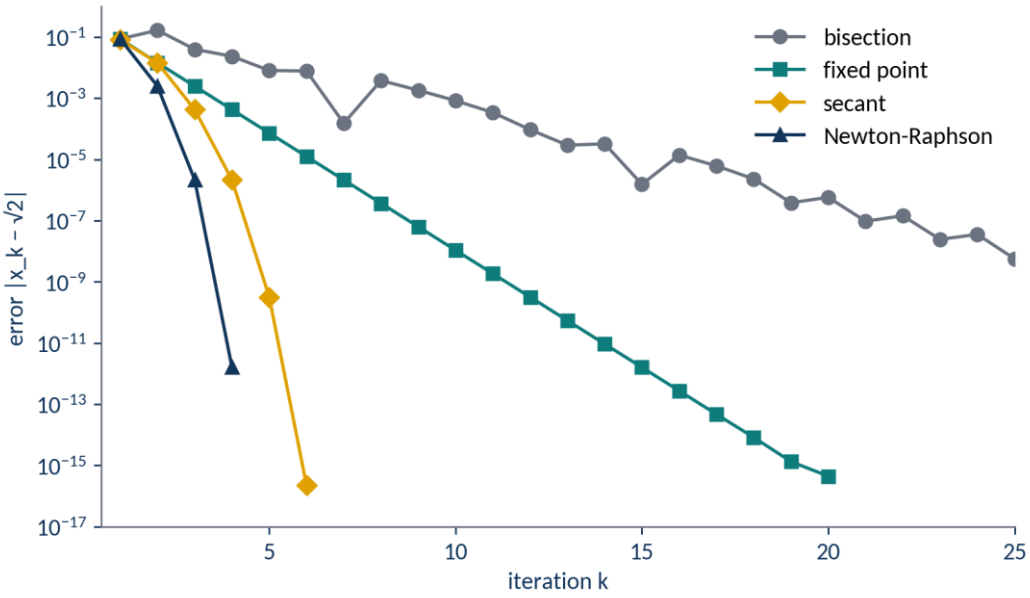


Figure 3.3 The error of four methods computing $\sqrt{2}$: bisection and fixed-point iteration fall along straight lines (linear convergence), the secant and Newton curves bend downward ever more steeply (orders 1.618 and 2). A line stops where the iterate equals the double nearest to $\sqrt{2}$.

The first Newton step is also a nice example of Heron's method: $\frac{1}{2}(1 + 2/1) = 1.5$, then $\frac{1}{2}(1.5 + 2/1.5) = 1.41667$, then 1.414215686, the values in the first column. A clay tablet from ancient Babylon, YBC 7289, shows $\sqrt{2}$ to about six significant digits, about as accurate as the third step.

3.9 When Newton Goes Wrong

Listing 3.7 and **Listing 3.8** run the failures of Section 3.3.3 and, for contrast, a good start for the same cubic.

Listing 3.7 `failures.py`: divergence, a cycle, a flat tangent, and a good start.

```
"""Three ways Newton-Raphson goes wrong, and the secant method on the same."""
import math
from roots2 import newton, secant


def cube_root(x):
    return math.copysign(abs(x) ** (1.0 / 3.0), x)


def main():
    print("1. f(x) = cube root of x, x0 = 0.1: each step doubles the distance")
    x = 0.1
    for k in range(1, 6):
        x = x - cube_root(x) / (cube_root(x) / (3 * x))   # f / f'
        print(f"   x{k} = {x:.6f}")

    print("2. f(x) = x^3 - 2x + 2, x0 = 0: a cycle 0, 1, 0, 1, ...")
    x = 0.0
    for k in range(1, 7):
        x = x - (x * x * x - 2 * x + 2) / (3 * x * x - 2)
        print(f"   x{k} = {x:.6f}")

    print("3. f(x) = x^2 - 4x + 5 has no real root; f'(2) = 0")
    try:
        newton(lambda t: t * t - 4 * t + 5, lambda t: 2 * t - 4, 2.0)
    except ArithmeticError as err:
        print(f"   stopped: {err}")

    print("4. Newton from a good start, x0 = -2 for x^3 - 2x + 2")
    root, k = newton(lambda t: t * t * t - 2 * t + 2,
                     lambda t: 3 * t * t - 2, -2.0)
    print(f"   root {root:.12f} after {k} iterations")
    root, k = secant(lambda t: t * t * t - 2 * t + 2, -2.0, -1.9)
    print(f"   secant: root {root:.12f} after {k} iterations")


if __name__ == "__main__":
    main()
```

Listing 3.8 `Failures.java`: the same four cases in Java.

```
import java.util.Locale;
import java.util.function.DoubleUnaryOperator;

/** Three ways Newton-Raphson goes wrong, and the secant method on the same. */
public class Failures {
    static double cubeRoot(double x) {
        return Math.copySign(Math.pow(Math.abs(x), 1.0 / 3.0), x);
    }

    public static void main(String[] args) {
        Locale.setDefault(Locale.ROOT);
        System.out.println("1. f(x) = cube root of x, x0 = 0.1: each step "
                           + "doubles the distance");
        double x = 0.1;
        for (int k = 1; k <= 5; k++) {
            x = x - cubeRoot(x) / (cubeRoot(x) / (3 * x));   // f / f'
            System.out.printf("   x%d = %.6f%n", k, x);
        }

        System.out.println("2. f(x) = x^3 - 2x + 2, x0 = 0: a cycle 0, 1, 0, "
                           + "1, ...");
        x = 0.0;
        for (int k = 1; k <= 6; k++) {
            x = x - (x * x * x - 2 * x + 2) / (3 * x * x - 2);
            System.out.printf("   x%d = %.6f%n", k, x);
        }

        System.out.println("3. f(x) = x^2 - 4x + 5 has no real root; "
                           + "f'(2) = 0");
        try {
            Roots2.newton(t -> t * t - 4 * t + 5, t -> 2 * t - 4, 2.0, 1e-10,
                          50, false);
        } catch (ArithmeticException err) {
            System.out.println("   stopped: " + err.getMessage());
        }
```

Listing 3.8 (Continued) *Failures.java: the same four cases in Java.*

```
        System.out.println("4. Newton from a good start, x0 = -2 for "
                           + "x^3 - 2x + 2");
        DoubleUnaryOperator f = t -> t * t * t - 2 * t + 2;
        Roots2.Result r = Roots2.newton(f, t -> 3 * t * t - 2, -2.0, 1e-10, 50,
                                        false);
        System.out.printf("   root %.12f after %d iterations%n", r.root(),
                          r.iterations());
        r = Roots2.secant(f, -2.0, -1.9, 1e-10, 50, false);
        System.out.printf("   secant: root %.12f after %d iterations%n",
                          r.root(), r.iterations());
    }
}
```

```
1. f(x) = cube root of x, x0 = 0.1: each step doubles the distance
   x1 = -0.200000
   x2 = 0.400000
   x3 = -0.800000
   x4 = 1.600000
   x5 = -3.200000
2. f(x) = x^3 - 2x + 2, x0 = 0: a cycle 0, 1, 0, 1, ...
   x1 = 1.000000
   x2 = 0.000000
   x3 = 1.000000
   x4 = 0.000000
   x5 = 1.000000
   x6 = 0.000000
3. f(x) = x^2 - 4x + 5 has no real root; f'(2) = 0
   stopped: f'(x) = 0 at x = 2.0
4. Newton from a good start, x0 = -2 for x^3 - 2x + 2
   root -1.769292354239 after 5 iterations
   secant: root -1.769292354239 after 6 iterations
```

The cube root doubles its distance at every step, $0.1, -0.2, 0.4, -0.8, 1.6, -3.2$, exactly as the algebra predicted. The cubic from $x_0 = 0$ bounces between 0 and 1 without end; a program without a maximum number of iterations would never stop. The function $x^2 - 4x + 5 = (x-2)^2 + 1$ is never zero, and at its lowest point $x = 2$ the tangent is flat, so `newton` refuses at once. But from $x_0 = -2$ the same cubic that cycled from 0 converges in 5 steps, and the secant method in 6: the difference is entirely in the starting point.

3.10 Testing It

Listing 3.9 and **Listing 3.10** check the three methods on $\sqrt{2}$, check that a flat tangent is refused, and check one property of the dipstick that we can verify without any computing: when the tank holds exactly half of its volume, the depth must be exactly the radius, 0.90 m.

Listing 3.9 `test_roots2.py`: unit tests of the open methods and the dipstick.

```python
"""Tests of the open root finders: python -m unittest test_roots2.py"""
import math
import unittest
from roots2 import fixed_point, newton, secant


class Roots2Test(unittest.TestCase):
    def test_newton_sqrt2(self):
        root, k = newton(lambda x: x * x - 2, lambda x: 2 * x, 1.0)
        self.assertAlmostEqual(root, math.sqrt(2), places=12)
        self.assertLessEqual(k, 6)

    def test_secant_sqrt2(self):
        root, _ = secant(lambda x: x * x - 2, 1.0, 2.0)
        self.assertAlmostEqual(root, math.sqrt(2), places=12)

    def test_fixed_point_sqrt2(self):
        root, _ = fixed_point(lambda x: 1 + 1 / (1 + x), 1.0)
        self.assertAlmostEqual(root, math.sqrt(2), places=9)

    def test_zero_derivative(self):
        with self.assertRaises(ArithmeticError):
            newton(lambda x: x * x + 1, lambda x: 2 * x, 0.0)

    def test_dipstick_full_and_half(self):
        from dipstick import FULL, depth
        h, _ = depth(FULL * 1000 / 2)      # half full: depth = radius
        self.assertAlmostEqual(h, 0.90, places=9)


if __name__ == "__main__":
    unittest.main()
```

Listing 3.10 `Roots2Test.java`: the same checks in Java.

```java
/** Tests of the open root finders: java -cp out Roots2Test */
public class Roots2Test {
    static int failures = 0;

    static void check(boolean ok, String name) {
        System.out.println((ok ? "ok    " : "FAILED") + "  " + name);
        if (!ok) {
            failures++;
        }
    }

    public static void main(String[] args) {
        double s2 = Math.sqrt(2);
        Roots2.Result n = Roots2.newton(x -> x * x - 2, x -> 2 * x, 1.0, 1e-10,
                                        50, false);
        check(Math.abs(n.root() - s2) < 1e-12 && n.iterations() <= 6,
              "Newton finds sqrt 2 in at most 6 steps");
        Roots2.Result s = Roots2.secant(x -> x * x - 2, 1.0, 2.0, 1e-10, 50,
                                        false);
        check(Math.abs(s.root() - s2) < 1e-12, "secant finds sqrt 2");
        Roots2.Result p = Roots2.fixedPoint(x -> 1 + 1 / (1 + x), 1.0, 1e-10,
                                            200, false);
        check(Math.abs(p.root() - s2) < 1e-9, "fixed point finds sqrt 2");
        try {
            Roots2.newton(x -> x * x + 1, x -> 2 * x, 0.0, 1e-10, 50, false);
            check(false, "zero derivative is refused");
        } catch (ArithmeticException e) {
            check(true, "zero derivative is refused");
        }
        double h = Dipstick.depth(Dipstick.FULL * 1000 / 2, false).root();
        check(Math.abs(h - 0.90) < 1e-9, "half-full tank: depth = radius");
        System.out.println(failures == 0 ? "all tests passed"
                                         : failures + " test(s) failed");
    }
}
```

```
$ python3 -m unittest test_roots2.py
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
$ java -cp out Roots2Test
```

```
ok      Newton finds sqrt 2 in at most 6 steps
ok      secant finds sqrt 2
ok      fixed point finds sqrt 2
ok      zero derivative is refused
ok      half-full tank: depth = radius
all tests passed
```

The half-full test is a good example of a test that does not merely repeat the program's own arithmetic. It checks the program against an independent fact, the symmetry of the circle. A test that compared the output with numbers copied from an earlier run of the same program would pass even if the volume formula were wrong. Other independent facts make equally good tests: the full tank must give $2R$, and the marks for V and for the full volume minus V must add up to $2R$, because the tank is symmetric about its middle.

3.11 Comparing with SciPy

SciPy's `newton` function uses Newton's method when it is given the derivative (`fprime`) and the secant method when it is not; `brentq` is the bracketing method of Chapter 2. **Listing 3.11** asks all three for the 3,000-litre mark, reusing the volume functions of `dipstick.py`.

Listing 3.11 `compare_scipy.py`: the 3,000-litre mark with SciPy's `newton` (with and without the derivative) and `brentq`.

```
"""The dipstick depth for 3000 L with SciPy (needs SciPy)."""
from scipy.optimize import brentq, newton

from dipstick import FULL, R, dvolume, volume

TARGET = 3.0                            # cubic metres


def f(h):
    return volume(h) - TARGET


def main():
    guess = 2 * R * TARGET / FULL
    h1 = newton(f, guess, fprime=dvolume)           # Newton-Raphson
    h2 = newton(f, guess)                           # no fprime: secant
    h3 = brentq(f, 0.0, 2 * R)                      # bracketing, Brent
    print(f"newton with fprime : {h1:.12f} m")
    print(f"newton, secant     : {h2:.12f} m")
    print(f"brentq on [0, 2R]  : {h3:.12f} m")


if __name__ == "__main__":
    main()
```

```
newton with fprime : 0.712612138378 m
newton, secant     : 0.712612138378 m
brentq on [0, 2R]  : 0.712612138378 m
```

All three agree with our 0.712612138378 m to twelve decimals, a millionth of a micrometre. SciPy's documentation notes the same trade-offs that this chapter derived: Newton converges fastest near a simple root but needs a good start and a derivative, the secant method needs no derivative, and a bracketing method such as `brentq` is the safe choice when a bracket is known [4].

3.12 Choosing a Method

Chapters 2 and 3 have built six root finders. **Table 3.1** puts them side by side.

Table 3.1 The root finders of Chapters 2 and 3 compared: what each needs, how fast it converges near a simple root, and whether it is guaranteed to succeed.

Method	Needs	Order	Evaluations per step	Guaranteed?
Bisection	a bracket	1 (rate 1/2)	1 value	yes, with a known number of steps
Regula falsi	a bracket	1 (rate varies)	1 value	yes, but can crawl
Illinois	a bracket	about 1.44	1 value	yes
Secant	two starting points	about 1.618	1 value	no
Newton-Raphson	a start and f'	2	1 value and 1 derivative	no
Fixed point	a form $x = g(x)$	1 (rate $\lvert g'(r)\rvert$)	1 value of g	only if $\lvert g'\rvert < 1$

The table suggests a simple way of thinking. If failure is not acceptable and you have a bracket, use a bracketing method: Illinois or, in Python, SciPy's `brentq`, which combines bisection's guarantee with the secant method's speed. If you need the same kind of root thousands of times,

have a good starting guess (as the box guess for the tank) and the derivative is easy, use Newton: it will usually finish in three or four steps. If the derivative is hard, use the secant method. Use fixed-point iteration when the problem already comes in the form $x = g(x)$ and you have checked that $|g'|$ is small.

In professional software the choice is often not either-or. A **safeguarded Newton** method keeps a bracket $[a, b]$ as bisection does, takes a Newton step from the current point, and accepts it only if it lands inside the bracket and reduces $|f|$ enough; otherwise it takes a bisection step instead. After every step the bracket is updated with the sign of f at the new point. Such a method inherits bisection's guarantee (it can never lose the root) and Newton's speed near the root (once the Newton steps are accepted, they are never replaced). Brent's method, met in Section 2.9, is the same idea with the secant step and inverse quadratic interpolation instead of Newton's step [2].

One more consideration decides many real choices: the *cost of evaluating* f. For Pak Harun's tank, one evaluation is a few arithmetic operations and all six methods finish in microseconds. When one evaluation of f means running a simulation for an hour, or measuring something in a laboratory, the number of evaluations is the only cost that matters, and a method of order 1.618 with one evaluation per step can beat a method of order 2 with two.

3.13 Numerical Methods in Practice: Profitina

Many quantities in a product are given by a smooth, increasing curve, and the questions people ask run the curve *backwards*. Profitina's PROFIT coin has a price oracle that converts between PROFIT and rupiah; a Pro subscription costs Rp50,000 for thirty days and can be paid with PROFIT. Converting a rupiah amount into coins at a fixed rate is a division, but as soon as the amount depends on the price in a non-linear way (a fee that shrinks with the amount, a discount tier, a decaying emission that sets how many coins exist), "how many coins do I need for this many rupiah?" becomes the inverse of a curve, which is exactly Pak Harun's problem: the formula runs from depth to volume, the question from volume to depth.

The same happens with scores. The AI Visibility Engine reports a score on a 0 to 100 scale and the historical dashboard shows its weekly trend. When a score is produced from a raw measurement through a smooth normalization curve, the question "what raw value would lift the score to 70?" inverts that curve. For smooth, monotone curves with a known derivative, Newton's method answers such questions in a handful of steps, and the box-guess idea of Section 3.2.2 (start from the answer that a straight-line approximation would give) is precisely how a production system chooses its starting point. Where the curve is only known from data, the secant method, which needs no derivative, is the natural choice.

3.14 Using It for Real

The dipstick program works for any lying cylinder; only `R` and `L` change. A few practical points make the difference between marks that are right and marks that only look right.

- **Measure the inside, not the outside.** The volume depends on the inner radius and length. A steel tank with 4 mm walls has an inner radius 4 mm smaller than the outer one, which at $R = 0.9$ m changes the full volume by almost 1%, about 70 litres.
- **Account for the ends.** Many truck tanks have slightly domed ends, which hold extra water not included in Equation (3.1). Compare the computed full volume with the volume stated by the tank maker, or better, with a measured fill.

- **Calibrate once with known volumes.** Fill the tank from a meter or with a known number of 1,000-litre cisterns, read the stick after each, and compare with the table. If the readings are consistently off by the same fraction, adjust L; if they are off mostly near the bottom, check the inner radius.
- **Other shapes.** An upright cylinder needs no root finding at all ($V = \pi R^2 h$). A spherical tank uses the volume of Exercise 2.6 with the derivative $V'(h) = \pi(2Rh - h^2)$, and Newton then needs about 4 steps where bisection needed 21. An elliptical tank, common on fuel trucks, works like the lying cylinder with the circle stretched: every volume is multiplied by the ratio of the ellipse's width to its height, and the depths of the marks stretch by the ratio of heights.
- **The accuracy you actually need.** The derivative Equation (3.2) also tells how much a reading error matters. Reading the stick 2 mm wrong at half depth, where the surface is widest, means $V' \times 0.002 = 2 \times 3.2 \times 0.9 \times 0.002 = 0.0115$ m^3, about 12 litres; near the 1,000-litre mark, where $V' = 4.43$ m^3 per metre, the same 2 mm mean about 9 litres. Water also expands by roughly 0.2% between a cool morning and a hot afternoon, about 16 litres in a full tank. Marks to the millimetre are therefore more than accurate enough; the computation is not the weak link, the reading is. It is worth remembering whenever a program prints many digits: they are only useful as far as the real world can read them.

The same program shape (a formula from input to output, its derivative, a good first guess, and Newton) solves many other inverse questions: the diameter of a pipe that delivers a given flow, the time at which a cooling drink reaches a given temperature, the rate hidden in an instalment plan (Exercise 3.5 does it with the secant method, without a derivative).

3.15 Common Mistakes, and How to Avoid Them

- **Starting where the derivative is zero.** A guess of $h = 0$ for the tank makes $V'(0) = 0$ and the first step divides by zero. Start from a sensible estimate, such as the box guess, and check `d == 0` in the code.
- **No maximum number of iterations.** A cycle such as 0,1,0,1 runs forever. Always cap the iterations and raise an error, rather than return the last value, when the cap is reached.
- **A wrong derivative.** Newton with a wrong f' usually still converges, but only linearly, and slowly, which hides the mistake. Check the derivative against a finite difference $(f(x + h) - f(x))/h$ with $h \approx 10^{-8}$ (Section 1.12.2) at one or two points.
- **Trusting a single run from a poor start.** An open method may converge to a different root than the one wanted, or to none. Check that the result is in the expected range and that $|f|$ is small there.
- **Choosing a fixed-point form without checking $|g'|$.** $x = 2/x$ and $x = x^2 + x - 2$ are both correct rewritings of $x^2 = 2$ and both fail. Compute g' at the expected root first: the iteration converges only if $|g'(r)| < 1$, and it is fast only if $|g'(r)|$ is small.
- **Expecting quadratic convergence at a multiple root.** At a double root Newton only halves the error per step. If Newton is unexpectedly slow, check whether $f'(r)$ is also near zero.

3.16 Chapter Summary

- Open methods follow the function's shape instead of keeping a bracket. They are fast near a root but need a good start.

- Newton-Raphson follows the tangent, $x_{k+1} = x_k - f(x_k)/f'(x_k)$ (Equation (3.3), **Pseudocode 3.1**). Near a simple root its error is squared at every step (Equation (3.4)): quadratic convergence, order 2 in the sense of Equation (3.5).
- Newton fails when the tangent is flat, when the function is too steep at the root, when it cycles, and it slows to linear convergence at a multiple root.
- The secant method replaces the derivative by the slope through the last two points (Equation (3.6)); its order is the golden ratio 1.618 (Equation (3.7)), and each step needs only one function value.
- Fixed-point iteration $x_{k+1} = g(x_k)$ (Equation (3.8)) converges when $|g'| < 1$ near the root, linearly with rate $|g'(r)|$ (Equation (3.9)); Newton is the fixed-point iteration whose g' vanishes at the root.
- The order of convergence can be measured from three consecutive errors (Equation (3.10)); for $\sqrt{2}$ the measurements gave 2, about 1.6, and 1 with ratio 0.1716.
- The water truck's volume Equation (3.1) and its derivative Equation (3.2) gave all eight dipstick marks in 3 to 5 Newton steps each, against 31 for bisection, in agreement with SciPy's `newton` and `brentq`.

3.17 Exercises

1. Carry out two Newton steps by hand for $f(x) = x^2 - 5$ from $x_0 = 2$. How many correct digits does each iterate have? Compare with $\sqrt{5} = 2.2360680$.
2. Derive the Newton iteration for computing $1/a$ using only multiplication and subtraction, from $f(x) = 1/x - a$. (Early computers without a division instruction worked this way.) Carry out three steps for $a = 3$ from $x_0 = 0.3$.
3. For the tank, compute the constant C of Equation (3.4) at the 1,000-litre root $h = 0.3247$ m, and predict the size of the second error if the first is 10^{-2} m.
4. Show that Newton's method for $f(x) = (x-1)^2$ gives $e_{k+1} = e_k/2$. Then show that the modified step $x_{k+1} = x_k - 2f(x_k)/f'(x_k)$ finds the root in one step. What is the general rule for a root of multiplicity m?
5. Find Dimas's loan rate of Chapter 2 with the secant method, starting from 1% and 5%, with the tolerance 10^{-12}. How many steps does it need, and why is Newton less convenient here?
6. Three fixed-point forms of $x^3 - x - 1 = 0$, whose root is $r \approx 1.3247$, are (a) $x = x^3 - 1$, (b) $x = (x+1)^{1/3}$ and (c) $x = 1/x + 1/x^2$. Check that each is a correct rewriting, and decide which converge and how fast.
7. For the dipstick, why does the secant method need 7 iterations for 8,000 litres but only 3 for 4,000 litres?

3.18 Solutions to the Exercises

1. $f'(x) = 2x$, so $x_{k+1} = x_k - (x_k^2 - 5)/(2x_k) = \frac{1}{2}(x_k + 5/x_k)$. From $x_0 = 2$: $x_1 = \frac{1}{2}(2 + 2.5) = 2.25$, and $x_2 = \frac{1}{2}(2.25 + 5/2.25) = \frac{1}{2}(2.25 + 2.222222) = 2.2361111$. Against $\sqrt{5} = 2.2360680$, x_0 has 1 correct digit, x_1 has 2 (error 0.014), x_2 has 5 (error 0.000043); the next step gives 2.2360680, correct to 10 digits.

2. $f(x) = 1/x - a$ and $f'(x) = -1/x^2$, so $x_{k+1} = x_k - \frac{1/x_k - a}{-1/x_k^2} = x_k + x_k - a x_k^2 = x_k(2 - a x_k)$: only multiplications and a subtraction. For $a = 3$: $x_1 = 0.3(2 - 0.9) = 0.33$, $x_2 = 0.33(2 -$

$0.99) = 0.3333$, $x_3 = 0.3333(2 - 0.9999) = 0.33333333$. The number of 3s doubles at every step. The error satisfies $e_{k+1} = -ae_k^2$ exactly, so the method converges for any start with $0 < x_0 < 2/a$.

3. At $h = 0.3247$: $2Rh - h^2 = 0.58446 - 0.10543 = 0.47903$, its square root is 0.69212, so $V' = 2 \times 3.2 \times 0.69212 = 4.4296$ and $V'' = 2L(R - h)/\sqrt{2Rh - h^2} = 6.4 \times 0.5753/0.69212 = 5.3198$. Then $C = 5.3198/(2 \times 4.4296) = 0.600$, about five times the value at 3,000 litres: the curve bends more near the bottom. A first error of 10^{-2} gives a second of about $0.6 \times 10^{-4} = 6 \times 10^{-5}$ m, and a third of about $0.6 \times (6 \times 10^{-5})^2 = 2 \times 10^{-9}$ m. That is why the 1,000-litre mark needs one step more than the 3,000-litre mark.

4. $f'(x) = 2(x - 1)$, so $x_{k+1} = x_k - (x_k - 1)^2/(2(x_k - 1)) = x_k - (x_k - 1)/2$, and $e_{k+1} = x_{k+1} - 1 = e_k - e_k/2 = e_k/2$: linear with rate $1/2$. With the factor 2, $x_{k+1} = x_k - (x_k - 1) = 1$: the root in one step. In general, for a root of multiplicity m (where $f = (x - r)^m h(x)$ with $h(r) \neq 0$), plain Newton converges linearly with rate $1 - 1/m$, and the step $x_{k+1} = x_k - mf(x_k)/f'(x_k)$ restores quadratic convergence, provided m is known.

5. **Listing 3.12** and **Listing 3.13** reuse the loan function of Chapter 2 with the secant method.

Listing 3.12 `loan_secant.py`: Dimas's loan rate by the secant method.

```
"""Exercise 3.5: the loan rate of Chapter 2 with the secant method."""
from roots2 import HEADER, secant

PRICE, PAY, MONTHS = 6000000.0, 600000.0, 12


def f(r):
    q = 1.0
    for _ in range(MONTHS):
        q *= 1 + r
    return PAY * (1 - 1 / q) / r - PRICE


def main():
    print(HEADER)
    r, k = secant(f, 0.01, 0.05, tol=1e-12, trace=True)
    print(f"monthly rate {r * 100:.10f} % after {k} iterations")


if __name__ == "__main__":
    main()
```

Listing 3.13 `LoanSecant.java`: the same in Java.

```java
import java.util.Locale;

/** Exercise 3.5: the loan rate of Chapter 2 with the secant method. */
public class LoanSecant {
    static final double PRICE = 6000000.0, PAY = 600000.0;
    static final int MONTHS = 12;

    static double f(double r) {
        double q = 1.0;
        for (int i = 0; i < MONTHS; i++) {
            q *= 1 + r;
        }
        return PAY * (1 - 1 / q) / r - PRICE;
    }

    public static void main(String[] args) {
        Locale.setDefault(Locale.ROOT);
        System.out.println(Roots2.HEADER);
        Roots2.Result r = Roots2.secant(LoanSecant::f, 0.01, 0.05, 1e-12, 50,
                                        true);
        System.out.printf("monthly rate %.10f %% after %d iterations%n",
                          r.root() * 100, r.iterations());
    }
}
```

```
  k                  x           f(x)  |x - x_old|
  1     0.030989445871  -6.271988e+04    1.901e-02
  2     0.029064234179   5.902075e+03    1.925e-03
  3     0.029229818857  -4.587681e+01    1.656e-04
  4     0.029228541695  -3.324991e-02    1.277e-06
  5     0.029228540769   2.067536e-07    9.263e-10
  6     0.029228540769   4.656613e-09    5.759e-15
monthly rate 2.9228540769 % after 6 iterations
```

Six steps reach a tolerance of 10^{-12}, where bisection needed 19 steps for the much looser 10^{-7}. The first step is the same as regula falsi's in Chapter 2, but then the secant method abandons the bracket: step 2 uses the two latest points, 5% and 3.10%, lands on the other side of the root ($f > 0$), and from then on the errors fall superlinearly: 1.8×10^{-3}, 1.6×10^{-4}, 1.3×10^{-6}, 9×10^{-10}. Newton would need the derivative of the loan function of Section 2.2.2 with respect to r, which exists but is a long expression with powers and quotients, and a single mistake in it would silently slow the method down. The secant method gets almost the same speed for free.

6. All three are correct: (a) adds x to both sides; (b) takes the cube root of $x^3 = x + 1$; (c) divides $x^3 = x + 1$ by x^2. (a) $g(x) = x^3 - 1$ has $g'(r) = 3r^2 = 5.26 > 1$: it diverges. (b) $g(x) = (x+1)^{1/3}$ has $g'(x) = \frac{1}{3}(x+1)^{-2/3}$, and at the root, where $(r+1)^{1/3} = r$, $g'(r) = 1/(3r^2) = 0.19$: it converges linearly, each step gaining about $-\log_{10} 0.19 = 0.72$ digits. (c) $g(x) = 1/x + 1/x^2$ has $g'(r) = -1/r^2 - 2/r^3 = -0.570 - 0.860 = -1.43$: it diverges, oscillating. Form (b) is the one to use; Newton, whose g' vanishes at the root, is better still. Always check a rewriting by multiplying back: it is easy to produce a form such as $x = 1 + 1/x$, which looks similar but belongs to the different equation $x^2 = x + 1$.

7. Both start from the box guess, and the number of steps depends on how far that guess is from the root and how strongly the curve bends there. At 4,000 litres the tank is almost exactly half full, where the curve is almost straight (its bend changes sign at half depth, $V''(R) = 0$), and the box guess 0.8842 m is within 0.35 cm of the root 0.8876 m: the secant through two nearby points is almost the tangent, and 3 steps suffice. At 8,000 litres the box guess 1.769 m is 5.5 cm from the root 1.7137 m, the curve is flattening toward the top, and the secant method, whose order is only 1.618, needs 7 steps where Newton needs 5.

References

[1] T. J. Ypma, "Historical development of the Newton–Raphson method," *SIAM Review*, vol. 37, no. 4, pp. 531–551, 1995.

[2] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical Analysis*, 10th ed. Boston, MA, USA: Cengage Learning, 2016.

[3] S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers*, 8th ed. New York, NY, USA: McGraw-Hill Education, 2021.

[4] SciPy Developers, "scipy.optimize.newton," *SciPy 1.18.1 Reference Guide*, 2026. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.newton.html

Chapter 4

Linear Systems I: Gaussian Elimination and Pivoting

Chapters 2 and 3 found one unknown from one equation. Most real problems have many unknowns tied together by many equations at once: several prices hidden in several receipts, several fertilizers mixed to meet several nutrient targets, thousands of temperatures in a heated plate, millions of weights in a model. When the equations are linear, a single, very old method solves them all: eliminate one unknown after another until one equation with one unknown remains, then work backward. This chapter builds that method, Gaussian elimination, with the care that a computer needs. By the end of the chapter you will be able to write a linear system in matrix form and solve it by hand with elimination and back substitution; count the arithmetic operations it costs and predict how the cost grows with the size of the system; explain why a zero or tiny pivot ruins the computation and how partial pivoting repairs it; measure how sensitive a system is to small changes with the condition number, and tell a small residual from a small error; implement the whole method in Java and Python with a step-by-step trace; and use it to mix fertilizers for a durian orchard, checking every answer against NumPy.

4.1 Bu Rahma's Durian Trees

Bu Rahma and her husband tend a fruit and durian orchard on a hillside near Jember: rambutan and mangosteen along the path, and forty durian trees that have borne fruit for six years. A durian tree that bears heavily year after year takes a great deal out of the soil, and the agricultural extension officer (*penyuluh pertanian*, PPL) of her district has written down, after a soil test, what each producing tree should receive over a year: 480 g of nitrogen (N), 490 g of phosphate (counted as P_2O_5), 660 g of potash (K_2O) and 320 g of sulfur (S).

The farm shop in the village does not sell nitrogen or phosphate. It sells bags of fertilizer, and each fertilizer contains several nutrients at once. The label of a compound fertilizer "NPK 15-15-15" means 15% N, 15% P_2O_5 and 15% K_2O by mass, and the one in the shop also carries 10% sulfur. DAP (diammonium phosphate, "18-46-0") has 18% N and 46% P_2O_5. KCl ("0-0-60") has 60% K_2O. ZA (ammonium sulfate) has 21% N and 24% S. How many grams of each should each tree get so that the four nutrients come out exactly right?

Bu Rahma tried with a calculator. More DAP gives more phosphate but also more nitrogen, which means less ZA, which means less sulfur, which means more NPK, which brings more phosphate again… Every change pulls on everything else. That is the signature of a *system* of equations: four unknowns, four conditions, all coupled. It is also exactly the kind of problem that people in China solved with counting rods two thousand years ago, and that every spreadsheet, engineering program and machine-learning library solves today by the method of this chapter [1].

We start with a smaller version that anyone can check at the kitchen table, and return to the orchard in Section 4.7. Everything said about fertilizer here is arithmetic: the recommendation itself must come from a soil test and your own extension officer, because soils, trees and seasons differ.

4.2 The Idea in Plain Words

4.2.1 Three receipts, three prices

Bu Rahma's daughter keeps three shopping receipts from the same warung, from three weeks in which prices did not change, but the receipts show only totals:

- week 1: 2 kg of rice, 1 litre of cooking oil and 2 kg of sugar, total Rp82,000;
- week 2: 5 kg of rice, 2 litres of oil and 1 kg of sugar, total Rp128,000;
- week 3: 3 kg of rice, 2 litres of oil and 3 kg of sugar, total Rp132,000.

What does each item cost? Call the prices (in thousands of rupiah) x_1 for rice, x_2 for oil and x_3 for sugar. Each receipt is one equation:

$$\begin{aligned} 2x_1 + x_2 + 2x_3 &= 82 \\ 5x_1 + 2x_2 + x_3 &= 128 \\ 3x_1 + 2x_2 + 3x_3 &= 132 \end{aligned} \tag{4.1}$$

Each equation is **linear**: every unknown appears only multiplied by a number and added, never squared, multiplied by another unknown, or inside a function. It is convenient to separate the numbers from the unknowns. The coefficients form a **matrix** A, a rectangular table of numbers with entry a_{ij} in row i and column j; the unknowns form a column vector x; the totals form a column vector b; and the whole system is written

$$Ax = b, \qquad A = \begin{pmatrix} 2 & 1 & 2 \\ 5 & 2 & 1 \\ 3 & 2 & 3 \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, \quad b = \begin{pmatrix} 82 \\ 128 \\ 132 \end{pmatrix} \tag{4.2}$$

Row i of $Ax = b$ says $a_{i1}x_1 + a_{i2}x_2 + a_{i3}x_3 = b_i$, which is the i-th receipt. For hand work we write A and b side by side as the **augmented matrix** $[A \mid b]$, and every operation on equations becomes an operation on its rows.

Why three receipts? With one receipt there are infinitely many price combinations that fit its total; with two, still infinitely many, because three unknowns need three independent pieces of information. Each receipt that is not a combination of the others removes one degree of freedom, and with as many independent receipts as items, exactly one set of prices remains. A fourth receipt would be a welcome check: if the prices really did not change, it must fit the three prices exactly, and if it does not, a price changed or a receipt was copied wrongly. Chapter 8 shows how to use *all* the receipts when there are more of them than items and they do not fit perfectly.

4.2.2 Elimination by hand

School algebra solves such a system by *elimination*: use one equation to remove an unknown from the others. Three kinds of row operations change the form of a system but not its solution: adding a multiple of one equation to another, swapping two equations, and multiplying an equation by a non-zero number. Gaussian elimination uses mostly the first.

Step 1: remove x_1 from rows 2 and 3. The first row has $2x_1$. To cancel the $5x_1$ of row 2, subtract $5/2 = 2.5$ times row 1 from row 2; to cancel the $3x_1$ of row 3, subtract $3/2 = 1.5$ times row 1. The numbers 2.5 and 1.5 are called **multipliers**, and the entry 2 that we divide by is the **pivot**:

- row 2: $(5,2,1 \mid 128) - 2.5 \times (2,1,2 \mid 82) = (0, -0.5, -4 \mid -77)$;
- row 3: $(3,2,3 \mid 132) - 1.5 \times (2,1,2 \mid 82) = (0,0.5,0 \mid 9)$.

Step 2: remove x_2 from row 3. The new pivot is -0.5, in row 2. The multiplier for row 3 is $0.5/(-0.5) = -1$, so row 3 becomes $(0,0.5,0 \mid 9) - (-1) \times (0,-0.5,-4 \mid -77) = (0,0,-4 \mid -68)$.

The system is now **upper triangular**: every entry below the diagonal is zero.

$$\left(\begin{array}{ccc|c} 2 & 1 & 2 & 82 \\ 0 & -0.5 & -4 & -77 \\ 0 & 0 & -4 & -68 \end{array}\right)$$

Step 3: back substitution. The last row says $-4x_3 = -68$, so $x_3 = 17$. The middle row says $-0.5x_2 - 4 \times 17 = -77$, so $-0.5x_2 = -9$ and $x_2 = 18$. The first row says $2x_1 + 18 + 2 \times 17 = 82$, so $2x_1 = 30$ and $x_1 = 15$. Rice costs Rp15,000 per kg, oil Rp18,000 per litre, sugar Rp17,000 per kg, and putting these back into the three receipts gives exactly 82, 128 and 132 thousand. The program of Section 4.6 prints every one of these intermediate matrices.

4.3 Gaussian Elimination

4.3.1 Forward elimination

For a general system of n equations in n unknowns, the procedure above repeats for each column $k = 1,2,\dots,n-1$. The entry a_{kk} on the diagonal is the pivot. For each row i below it, the multiplier is

$$m_{ik} = \frac{a_{ik}}{a_{kk}} \tag{4.3}$$

and row i is updated by subtracting m_{ik} times row k:

$$a_{ij} \leftarrow a_{ij} - m_{ik}\, a_{kj} \quad (j = k+1, \dots, n), \qquad b_i \leftarrow b_i - m_{ik}\, b_k \tag{4.4}$$

after which a_{ik} is zero by construction. **Figure 4.1** shows the pattern for four equations: each stage clears one column below the pivot, and only the shaded block to the lower right changes. The zeros form a staircase, and the work shrinks as the staircase grows.

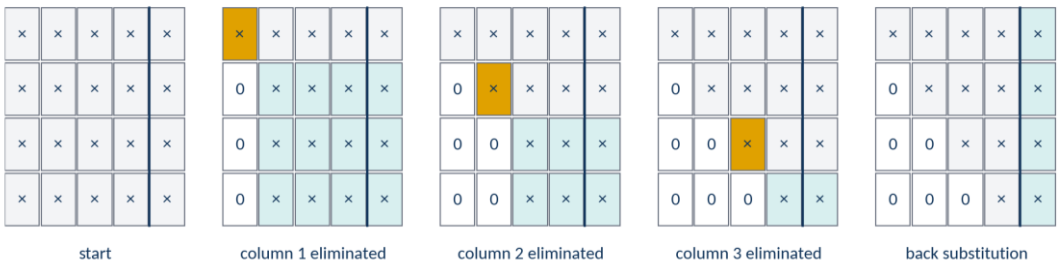


Figure 4.1 Gaussian elimination on four equations: at each stage the pivot (amber) clears its column below it and only the shaded entries change; back substitution then turns the right-hand column into the solution.

The method is named after Carl Friedrich Gauss, who used it around 1809 to solve the systems of his method of least squares, but it is much older. The Chinese classic *Nine Chapters on the Mathematical Art* (*Jiuzhang Suanshu*), compiled around the first century, describes the same elimination on a counting board as the method *fangcheng*, applied to problems about grain yields that look strikingly like our receipts [1].

4.3.2 Back substitution

When elimination is finished, the matrix is upper triangular, and the unknowns are found from the last to the first. Row i reads $a_{ii}x_i + a_{i,i+1}x_{i+1} + \cdots + a_{in}x_n = b_i$, where $x_{i+1}, \dots, x_n$ are already known, so

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j=i+1}^{n} a_{ij}\, x_j \right), \qquad i = n, n-1, \dots, 1 \tag{4.5}$$

Pseudocode 4.1 puts the two phases together.

Pseudocode 4.1 Gaussian elimination with optional partial pivoting (lines 2 to 4) and back substitution.

```
GAUSS(A, b, pivot)                  // A is n x n, b has n entries
 1  for k = 1 to n - 1                          // forward elimination
 2      if pivot:
 3          p = the row i >= k with the largest |a_ik|
 4          swap rows k and p of A and b
 5      if a_kk == 0: error "zero pivot"
 6      for i = k + 1 to n
 7          m = a_ik / a_kk                     // the multiplier
 8          a_ik = 0
 9          for j = k + 1 to n: a_ij = a_ij - m * a_kj
10          b_i = b_i - m * b_k
11  if a_nn == 0: error "zero pivot"
12  for i = n downto 1                          // back substitution
13      s = b_i
14      for j = i + 1 to n: s = s - a_ij * x_j
15      x_i = s / a_ii
16  return x
```

4.3.3 Counting the work

How long does elimination take for 10, 100 or 10,000 unknowns? We count **flops**, floating-point operations (one addition, subtraction, multiplication or division each). At stage k, there are $m = n - k$ rows below the pivot. For each of them we compute one multiplier (1 division), update m entries of the row (one multiplication and one subtraction each, $2m$ flops) and update b_i (2 flops): $2m + 3$ flops per row, $m(2m + 3)$ for the stage. Summing over $m = 1, \dots, n-1$ with the formulas $\sum m = \frac{(n-1)n}{2}$ and $\sum m^2 = \frac{(n-1)n(2n-1)}{6}$ gives the elimination cost $\frac{(n-1)n(2n-1)}{3} + \frac{3(n-1)n}{2}$. Back substitution needs, for row i, $2(n-i)$ flops for the sum and 1 for the division, n^2 in total. Altogether

$$\text{flops}(n) = \frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n \approx \frac{2}{3}n^3 \tag{4.6}$$

For large n the cubic term dominates: doubling the number of unknowns multiplies the work by eight. Elimination costs $\frac{2}{3}n^3$ while back substitution costs only n^2, which becomes important in Chapter 6, where one elimination is reused for many right-hand sides. **Listing 4.1** and **Listing 4.2** count the flops of our implementation and compare them with Equation (4.6).

Listing 4.1 `op_count.py`: counted flops of Gaussian elimination against the formula Equation (4.6).

```python
"""Counting the work: flops of Gaussian elimination against 2n^3/3."""
import gauss


def main():
    print(f"{'n':>4} {'flops counted':>14} {'formula':>14} {'2n^3/3':>14}")
    seed = 7
    for n in (3, 10, 50, 100, 200):
        a, b = [], []
        for i in range(n):                          # a random, safe matrix
            row = []
            for j in range(n):
                seed = (1103515245 * seed + 12345) % 2147483648
                row.append(seed / 2147483648.0 + (n if i == j else 0))
            a.append(row)
            b.append(1.0)
        gauss.solve(a, b)
        exact = (4 * n ** 3 + 9 * n ** 2 - 7 * n) // 6
        print(f"{n:>4} {gauss.flops:>14,} {exact:>14,} {2 * n ** 3 / 3:>14,.0f}")


if __name__ == "__main__":
    main()
```

Listing 4.2 `OpCount.java`: the same count in Java.

```java
import java.util.Locale;

/** Counting the work: flops of Gaussian elimination against 2n^3/3. */
public class OpCount {
    public static void main(String[] args) {
        Locale.setDefault(Locale.ROOT);
        System.out.printf("%4s %14s %14s %14s%n", "n", "flops counted",
                          "formula", "2n^3/3");
        long seed = 7;
        for (int n : new int[] {3, 10, 50, 100, 200}) {
            double[][] a = new double[n][n];
            double[] b = new double[n];
            for (int i = 0; i < n; i++) {          // a random, safe matrix
                for (int j = 0; j < n; j++) {
                    seed = (1103515245L * seed + 12345L) % 2147483648L;
                    a[i][j] = seed / 2147483648.0 + (i == j ? n : 0);
                }
                b[i] = 1.0;
            }
            Gauss.solve(a, b);
            long nn = n;
            long exact = (4 * nn * nn * nn + 9 * nn * nn - 7 * nn) / 6;
            System.out.printf("%4d %,14d %,14d %,14.0f%n", n, Gauss.flops, exact,
                              2.0 * n * n * n / 3);
        }
    }
}
```

```
   n  flops counted        formula         2n^3/3
   3              28             28             18
  10             805            805            667
  50          87,025         87,025         83,333
 100         681,550        681,550        666,667
 200       5,393,100      5,393,100      5,333,333
```

The count matches the formula exactly for every size, and for $n = 200$ the leading term $\frac{2}{3}n^3$ is already within about 1% of the total. The matrices are made *diagonally dominant* (each diagonal entry is larger than the sum of the others in its row) by adding n to the diagonal, which guarantees that no pivot is zero; Chapter 5 explains why. At a few billion flops per second, a

laptop solves 1,000 equations in well under a second (6.7×10^8 flops) but 100,000 equations would take 6.7×10^{14} flops, several days. That is why large systems in practice are either *sparse* (mostly zeros, which a good method skips) or solved by the iterative methods of Chapter 5.

4.3.4 When there is no unique solution

Elimination can meet a pivot that is exactly zero, with only zeros below it in its column. Then no row swap helps, and the system does not have exactly one solution: it has none (the equations contradict each other) or infinitely many (one equation repeats the information of the others). Two receipts for "1 kg of rice and 2 litres of oil" at Rp51,000 and "2 kg of rice and 4 litres of oil" at Rp102,000 say the same thing twice and cannot fix two prices. Such a matrix is called **singular**. For a non-singular matrix, elimination with row swaps always succeeds, and it gives the **determinant** as a by-product: row operations of the kind in Equation (4.4) do not change it, each swap changes its sign, and the determinant of a triangular matrix is the product of its diagonal, so

$$\det A = (-1)^s \prod_{k=1}^{n} a_{kk}^{(\text{final})} \tag{4.7}$$

where s is the number of row swaps. For the receipts, no swaps and pivots $2, -0.5, -4$ give $\det A = 4$. A determinant of exactly zero means singular, but the size of a non-zero determinant says little about how *trustworthy* a solution is; Section 4.8 introduces the right measure.

4.3.5 Many right-hand sides

Bu Rahma's orchard has trees of different ages. Young trees that have not yet borne fruit receive a different recommendation from producing trees, and old trees a third. The *matrix* of Equation (4.11) depends only on the fertilizers, so it is the same for every age; only the right-hand side b changes. Repeating the whole elimination for each b would waste work, because the multipliers m_{ik} depend only on A. The efficient way is to eliminate once, *keep the multipliers*, and for each new b apply the same multipliers to b (the updates $b_i \leftarrow b_i - m_{ik} b_k$ of Equation (4.4), about n^2 flops) and then back-substitute (another n^2). For three ages and four fertilizers the saving is trivial, but for a model that solves the same system with a thousand different right-hand sides, the cost falls from $1000 \times \frac{2}{3}n^3$ to $\frac{2}{3}n^3 + 1000 \times 2n^2$. Keeping the multipliers turns out to be the same as writing A as the product of a lower triangular matrix L (holding the multipliers) and an upper triangular matrix U (the result of elimination): the LU decomposition, which Chapter 6 develops together with its relatives, Cholesky and QR.

4.4 Pivoting

4.4.1 A zero pivot

Suppose the daughter had written the receipts in a different order, and the first receipt of the new list had no rice in it: 1 litre of oil and 2 kg of sugar for Rp52,000. The system

$$\begin{pmatrix} 0 & 1 & 2 \\ 5 & 2 & 1 \\ 3 & 2 & 3 \end{pmatrix} x = \begin{pmatrix} 52 \\ 128 \\ 132 \end{pmatrix} \tag{4.8}$$

has the same prices as its solution (check: $18 + 34 = 52$), but plain elimination stops at once, because the first pivot is 0 and the multipliers $5/0$ and $3/0$ do not exist. The cure is obvious to a

person: use another equation first. Swapping rows 1 and 2 gives a pivot of 5 and elimination proceeds normally. A computer must be told to do this.

4.4.2 A tiny pivot

A zero pivot at least announces itself. A *tiny* pivot is worse, because it silently destroys the answer. Consider

$$10^{-20}\,x + y = 1, \qquad x + y = 2 \tag{4.9}$$

whose solution is extremely close to $x = 1$, $y = 1$. Plain elimination uses the pivot 10^{-20} and the multiplier $m = 1/10^{-20} = 10^{20}$. Row 2 becomes $(1 - 10^{20})\,y = 2 - 10^{20}$. In double precision, with about 16 significant digits, $1 - 10^{20}$ rounds to -10^{20} and $2 - 10^{20}$ rounds to -10^{20}: the 1 and the 2 have disappeared, exactly as a few rupiah disappear when added to a billion in single precision (Section 1.13). So $y = 1$, which happens to be right. But back substitution in row 1 then gives $10^{-20}x = 1 - y = 0$, so $x = 0$, completely wrong. The information of the second equation was swamped by the huge multiplier.

With the rows swapped, the pivot is 1, the multiplier is 10^{-20}, and row 2 becomes $(1 - 10^{-20})y = 1 - 2 \times 10^{-20}$, which rounds to $y = 1$; row 1 then gives $x = 2 - y = 1$. Correct. The difference is entirely in the size of the multiplier: a multiplier larger than 1 magnifies the rounding errors of the pivot row when they are subtracted from the other rows.

4.4.3 Partial pivoting

Partial pivoting makes the choice systematic. At stage k, before eliminating, look down column k from the diagonal and bring the row with the largest entry in absolute value to position k:

$$p = \operatorname*{argmax}_{i \geq k} |a_{ik}|, \qquad \text{swap rows } k \text{ and } p \tag{4.10}$$

Then every multiplier satisfies $|m_{ik}| \leq 1$, so no row is ever multiplied up before it is subtracted. Lines 2 to 4 of **Pseudocode 4.1** implement it. The swap costs no flops, only a comparison per row, and it removes both the zero pivot (a non-singular matrix always has a non-zero entry somewhere in the column) and the tiny pivot.

Is partial pivoting always enough? Not in theory. Wilkinson showed that the entries can still grow during elimination, in the worst case by a factor 2^{n-1}, and constructed matrices for which that happens. But in practice such growth is so rare that Gaussian elimination with partial pivoting is regarded as stable for all practical purposes, and it is the method that LAPACK, and therefore NumPy, MATLAB and R, use for general systems [2], [3]. Two stronger variants exist: *scaled* partial pivoting compares $|a_{ik}|$ relative to the largest entry of row i, which protects against rows written in very different units, and *complete* pivoting searches the whole remaining block for the largest entry and swaps columns too, at a cost of $O(n^3)$ extra comparisons. Both are rarely needed [4].

4.5 Building It in Java and Python

Listing 4.3 and **Listing 4.4** implement **Pseudocode 4.1**, a function for the residual (Section 4.8), and a small reader for comma-separated data files. The matrix and the right-hand side are copied first, so the caller's data are never changed.

Listing 4.3 `gauss.py`: Gaussian elimination with optional partial pivoting, back substitution, the residual, and a data reader.

```
"""Gaussian elimination with (optional) partial pivoting and back
substitution, written out in full so that every step can be followed."""

flops = 0                   # floating-point operations of the last solve


def show(a, b, title):
    """Print the augmented matrix [A | b]."""
    print(title)
    for row, rhs in zip(a, b):
        print(" " + "".join(f" {v:10.4f}" for v in row) + f" | {rhs:10.4f}")


def solve(a, b, pivot=True, trace=False):
    """Solve A x = b. A and b are copied, never changed.
    Raises ZeroDivisionError when a pivot is exactly zero."""
    global flops
    flops = 0
    n = len(b)
    a = [list(map(float, row)) for row in a]
    b = [float(v) for v in b]
    if trace:
        show(a, b, "start")
    for k in range(n - 1):
        if pivot:                                   # partial pivoting:
            p = max(range(k, n), key=lambda i: abs(a[i][k]))
            if p != k:                              # bring the largest |a_ik|
                a[k], a[p] = a[p], a[k]             # of column k to row k
                b[k], b[p] = b[p], b[k]
                if trace:
                    print(f"swap rows {k + 1} and {p + 1}")
        if a[k][k] == 0:
            raise ZeroDivisionError(f"zero pivot in column {k + 1}")
        for i in range(k + 1, n):
            m = a[i][k] / a[k][k]                   # the multiplier
            flops += 1
            a[i][k] = 0.0
            for j in range(k + 1, n):
                a[i][j] -= m * a[k][j]
                flops += 2
            b[i] -= m * b[k]
            flops += 2
        if trace:
            show(a, b, f"after eliminating column {k + 1}")
    if a[n - 1][n - 1] == 0:
        raise ZeroDivisionError(f"zero pivot in column {n}")
```

Listing 4.3 (Continued) *gauss.py: Gaussian elimination with optional partial pivoting, back substitution, the residual, and a data reader.*

```
    x = [0.0] * n
    for i in range(n - 1, -1, -1):              # back substitution
        s = b[i]
        for j in range(i + 1, n):
            s -= a[i][j] * x[j]
            flops += 2
        x[i] = s / a[i][i]
        flops += 1
    return x


def residual(a, x, b):
    """The largest |b_i - (A x)_i|."""
    worst = 0.0
    for row, rhs in zip(a, b):
        s = rhs
        for aij, xj in zip(row, x):
            s -= aij * xj
        worst = max(worst, abs(s))
    return worst


def read_table(path):
    """Read a comma-separated file of numbers; the first field of a line may
    be a name. Lines starting with # are comments."""
    names, rows = [], []
    with open(path, encoding="utf-8") as fh:
        for line in fh:
            line = line.strip()
            if not line or line.startswith("#"):
                continue
            fields = [f.strip() for f in line.split(",")]
            try:
                float(fields[0])
                names.append("")
                rows.append([float(f) for f in fields])
            except ValueError:
                names.append(fields[0])
                rows.append([float(f) for f in fields[1:]])
    return names, rows
```

Listing 4.4 `Gauss.java`: the same module in Java.

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;

/**
 * Gaussian elimination with (optional) partial pivoting and back
 * substitution, written out in full so that every step can be followed.
 */
public class Gauss {
    static long flops = 0;          // floating-point operations of last solve

    /** Print the augmented matrix [A | b]. */
    static void show(double[][] a, double[] b, String title) {
        System.out.println(title);
        for (int i = 0; i < b.length; i++) {
            StringBuilder line = new StringBuilder(" ");
            for (double v : a[i]) {
                line.append(String.format(" %10.4f", v));
            }
            System.out.println(line + String.format(" | %10.4f", b[i]));
        }
    }

    /** Solve A x = b; A and b are copied, never changed. */
    static double[] solve(double[][] a0, double[] b0, boolean pivot,
                          boolean trace) {
        flops = 0;
        int n = b0.length;
        double[][] a = new double[n][];
        for (int i = 0; i < n; i++) {
            a[i] = a0[i].clone();
        }
        double[] b = b0.clone();
        if (trace) {
            show(a, b, "start");
        }
        for (int k = 0; k < n - 1; k++) {
            if (pivot) {                          // partial pivoting: bring the
                int p = k;                        // largest |a_ik| of column k
                for (int i = k + 1; i < n; i++) {    // to row k
                    if (Math.abs(a[i][k]) > Math.abs(a[p][k])) {
                        p = i;
                    }
                }
```

Listing 4.4 (Continued) *Gauss.java: the same module in Java.*

```
                if (p != k) {
                    double[] row = a[k];
                    a[k] = a[p];
                    a[p] = row;
                    double t = b[k];
                    b[k] = b[p];
                    b[p] = t;
                    if (trace) {
                        System.out.printf("swap rows %d and %d%n", k + 1, p + 1);
                    }
                }
            }
            if (a[k][k] == 0) {
                throw new ArithmeticException("zero pivot in column " + (k + 1));
            }
            for (int i = k + 1; i < n; i++) {
                double m = a[i][k] / a[k][k];    // the multiplier
                flops += 1;
                a[i][k] = 0.0;
                for (int j = k + 1; j < n; j++) {
                    a[i][j] -= m * a[k][j];
                    flops += 2;
                }
                b[i] -= m * b[k];
                flops += 2;
            }
            if (trace) {
                show(a, b, "after eliminating column " + (k + 1));
            }
        }
        if (a[n - 1][n - 1] == 0) {
            throw new ArithmeticException("zero pivot in column " + n);
        }
        double[] x = new double[n];
        for (int i = n - 1; i >= 0; i--) {       // back substitution
            double s = b[i];
            for (int j = i + 1; j < n; j++) {
                s -= a[i][j] * x[j];
                flops += 2;
            }
            x[i] = s / a[i][i];
            flops += 1;
        }
        return x;
    }
```

Listing 4.4 (Continued) *Gauss.java: the same module in Java.*

```
    static double[] solve(double[][] a, double[] b) {
        return solve(a, b, true, false);
    }

    /** The largest |b_i - (A x)_i|. */
    static double residual(double[][] a, double[] x, double[] b) {
        double worst = 0.0;
        for (int i = 0; i < b.length; i++) {
            double s = b[i];
            for (int j = 0; j < x.length; j++) {
                s -= a[i][j] * x[j];
            }
            worst = Math.max(worst, Math.abs(s));
        }
        return worst;
    }

    /** One line of a data file: an optional name and its numbers. */
    record Line(String name, double[] values) { }

    /** Read a comma-separated file; lines starting with # are comments. */
    static List<Line> readTable(String path) throws IOException {
        List<Line> lines = new ArrayList<>();
        for (String raw : Files.readAllLines(Path.of(path))) {
            String line = raw.strip();
            if (line.isEmpty() || line.startsWith("#")) {
                continue;
            }
            String[] f = line.split(",");
            String name = "";
            int start = 0;
            try {
                Double.parseDouble(f[0].strip());
            } catch (NumberFormatException e) {
                name = f[0].strip();
                start = 1;
            }
            double[] v = new double[f.length - start];
            for (int i = start; i < f.length; i++) {
                v[i - start] = Double.parseDouble(f[i].strip());
            }
            lines.add(new Line(name, v));
        }
        return lines;
    }
}
```

A matrix is a list of rows in Python and a `double[][]` in Java, an array of row arrays. Swapping two rows therefore swaps two references and copies no numbers, which is why pivoting is cheap. The global counter `flops` (a static field in Java) adds up the operations of the last call, exactly as counted in Section 4.3.3: one for each multiplier, two for each updated entry and each update of b, two for each term of the back-substitution sum and one for each division. The data reader accepts lines that start with a name (“NPK 15-15-15, 0.15, …”) or with a number, and skips comments starting with `#`, so that the data files can explain themselves. A zero pivot raises `ZeroDivisionError` in Python and `ArithmeticException` in Java, with the column where it happened.

4.6 Running It: The Receipts

The receipts are stored in `data/receipts.csv`:

```
# Three receipts: kg of rice, litres of oil, kg of sugar, total (thousand Rp)
2, 1, 2, 82
5, 2, 1, 128
3, 2, 3, 132
```

Listing 4.5 and **Listing 4.6** read the file, solve Equation (4.1) *without* pivoting so that every step matches our hand computation, and print the prices and the residual.

Listing 4.5 `receipts.py`: three receipts, three prices, with every elimination step printed.

```python
"""Three shopping receipts, three unknown prices: the worked example."""
from pathlib import Path
from gauss import read_table, residual, solve

DATA = Path(__file__).resolve().parent.parent / "data" / "receipts.csv"


def main():
    _, rows = read_table(DATA)
    a = [row[:3] for row in rows]
    b = [row[3] for row in rows]
    x = solve(a, b, pivot=False, trace=True)
    for name, price in zip(("rice", "oil", "sugar"), x):
        print(f"{name:<6}: Rp {price * 1000:,.0f}")
    print(f"largest residual: {residual(a, x, b):.3e}")


if __name__ == "__main__":
    main()
```

Listing 4.6 `Receipts.java`: the same program in Java.

```java
import java.io.IOException;
import java.util.List;
import java.util.Locale;

/** Three shopping receipts, three unknown prices: the worked example. */
public class Receipts {
    public static void main(String[] args) throws IOException {
        Locale.setDefault(Locale.ROOT);
        List<Gauss.Line> rows = Gauss.readTable("../data/receipts.csv");
        double[][] a = new double[3][3];
        double[] b = new double[3];
        for (int i = 0; i < 3; i++) {
            System.arraycopy(rows.get(i).values(), 0, a[i], 0, 3);
            b[i] = rows.get(i).values()[3];
        }
        double[] x = Gauss.solve(a, b, false, true);
        String[] names = {"rice", "oil", "sugar"};
        for (int i = 0; i < 3; i++) {
            System.out.printf("%-6s: Rp %,.0f%n", names[i], x[i] * 1000);
        }
        System.out.printf("largest residual: %.3e%n", Gauss.residual(a, x, b));
    }
}
```

```
$ python3 receipts.py          (or, in the java folder: java -cp out Receipts)
```

```
start
      2.0000     1.0000     2.0000 |    82.0000
      5.0000     2.0000     1.0000 |   128.0000
      3.0000     2.0000     3.0000 |   132.0000
after eliminating column 1
      2.0000     1.0000     2.0000 |    82.0000
      0.0000    -0.5000    -4.0000 |   -77.0000
      0.0000     0.5000     0.0000 |     9.0000
after eliminating column 2
      2.0000     1.0000     2.0000 |    82.0000
      0.0000    -0.5000    -4.0000 |   -77.0000
      0.0000     0.0000    -4.0000 |   -68.0000
rice  : Rp 15,000
oil   : Rp 18,000
sugar : Rp 17,000
largest residual: 0.000e+00
```

The three matrices are exactly those of Section 4.2.2, and the prices are Rp15,000, Rp18,000 and Rp17,000. Every number here is a small integer or a half, all exactly representable in binary, so the arithmetic is exact and the residual is zero. The orchard will not be so kind.

4.7 The Durian Orchard's Fertilizer

The fertilizer data go into `data/fertilizer.csv`: one line per fertilizer with the mass fractions of its four nutrients, and a last line with the targets in grams per tree per year.

```
# One line per fertilizer: name, then the fraction of its mass that is
# N, P2O5, K2O and S. The last line is the target in grams per tree per year.
NPK 15-15-15, 0.15, 0.15, 0.15, 0.10
DAP 18-46-0, 0.18, 0.46, 0.00, 0.00
KCl 0-0-60, 0.00, 0.00, 0.60, 0.00
ZA 21-0-0-24S, 0.21, 0.00, 0.00, 0.24
target, 480, 490, 660, 320
```

Let x_1, x_2, x_3, x_4 be the grams per tree of NPK, DAP, KCl and ZA. The nitrogen delivered is $0.15x_1 + 0.18x_2 + 0x_3 + 0.21x_4$, and it must equal 480; the other nutrients give three more equations:

$$\begin{pmatrix} 0.15 & 0.18 & 0 & 0.21 \\ 0.15 & 0.46 & 0 & 0 \\ 0.15 & 0 & 0.60 & 0 \\ 0.10 & 0 & 0 & 0.24 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 480 \\ 490 \\ 660 \\ 320 \end{pmatrix} \tag{4.11}$$

Note that the file lists one *fertilizer* per line, but in the matrix each *nutrient* is a row: the matrix is the transpose of the table in the file. **Listing 4.7** and **Listing 4.8** build it, solve with partial pivoting, print the amounts per tree and for the whole orchard, and check each nutrient.

Listing 4.7 `fertilizer.py`: the grams of each fertilizer per tree, and the kilograms for the orchard.

```python
"""How much of each fertilizer gives a durian tree its N, P, K and S?
Usage: python fertilizer.py [number of trees]"""
import sys
from pathlib import Path
from gauss import read_table, residual, solve

DATA = Path(__file__).resolve().parent.parent / "data" / "fertilizer.csv"
NUTRIENTS = ("N", "P2O5", "K2O", "S")


def main():
    trees = int(sys.argv[1]) if len(sys.argv) > 1 else 40
    names, rows = read_table(DATA)
    ferts, target = names[:-1], rows[-1]
    # row i of A is nutrient i, column j is fertilizer j
    a = [[rows[j][i] for j in range(len(ferts))] for i in range(len(NUTRIENTS))]
    x = solve(a, target, pivot=True, trace=True)
    print(f"\n{'fertilizer':<14} {'g per tree':>11} {'kg for ' + str(trees):>11}")
    for name, grams in zip(ferts, x):
        print(f"{name:<14} {grams:>11.1f} {grams * trees / 1000:>11.2f}")
    if min(x) < 0:
        print("A negative amount means: this target cannot be met with "
              "these fertilizers.")
    print(f"largest residual: {residual(a, x, target):.3e} g")
    for i, nut in enumerate(NUTRIENTS):
        got = sum(a[i][j] * x[j] for j in range(len(x)))
        print(f"  {nut:<5} wanted {target[i]:6.1f} g, gets {got:8.3f} g")


if __name__ == "__main__":
    main()
```

Listing 4.8 `Fertilizer.java`: the same program in Java.

```java
import java.io.IOException;
import java.util.List;
import java.util.Locale;

/**
 * How much of each fertilizer gives a durian tree its N, P, K and S?
 * Usage: java Fertilizer [number of trees]   (run from the java folder)
 */
public class Fertilizer {
    static final String[] NUTRIENTS = {"N", "P2O5", "K2O", "S"};

    public static void main(String[] args) throws IOException {
        Locale.setDefault(Locale.ROOT);
        int trees = args.length > 0 ? Integer.parseInt(args[0]) : 40;
        List<Gauss.Line> rows = Gauss.readTable("../data/fertilizer.csv");
        int m = rows.size() - 1, n = NUTRIENTS.length;
        double[] target = rows.get(m).values();
        double[][] a = new double[n][m];        // row: nutrient, column: fert.
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                a[i][j] = rows.get(j).values()[i];
            }
        }
        double[] x = Gauss.solve(a, target, true, true);
        System.out.printf("%n%-14s %11s %11s%n", "fertilizer", "g per tree",
                          "kg for " + trees);
        double least = x[0];
        for (int j = 0; j < m; j++) {
            System.out.printf("%-14s %11.1f %11.2f%n", rows.get(j).name(), x[j],
                              x[j] * trees / 1000);
            least = Math.min(least, x[j]);
        }
        if (least < 0) {
            System.out.println("A negative amount means: this target cannot be "
                               + "met with these fertilizers.");
        }
        System.out.printf("largest residual: %.3e g%n",
                          Gauss.residual(a, x, target));
        for (int i = 0; i < n; i++) {
            double got = 0;
            for (int j = 0; j < m; j++) {
                got += a[i][j] * x[j];
            }
            System.out.printf("  %-5s wanted %6.1f g, gets %8.3f g%n",
                              NUTRIENTS[i], target[i], got);
        }
    }
}
```

Listing 4.8 (Continued) *Fertilizer.java: the same program in Java.*

```
start
      0.1500     0.1800     0.0000     0.2100 |   480.0000
      0.1500     0.4600     0.0000     0.0000 |   490.0000
      0.1500     0.0000     0.6000     0.0000 |   660.0000
      0.1000     0.0000     0.0000     0.2400 |   320.0000
after eliminating column 1
      0.1500     0.1800     0.0000     0.2100 |   480.0000
      0.0000     0.2800     0.0000    -0.2100 |    10.0000
      0.0000    -0.1800     0.6000    -0.2100 |   180.0000
      0.0000    -0.1200     0.0000     0.1000 |    -0.0000
after eliminating column 2
      0.1500     0.1800     0.0000     0.2100 |   480.0000
      0.0000     0.2800     0.0000    -0.2100 |    10.0000
      0.0000     0.0000     0.6000    -0.3450 |   186.4286
      0.0000     0.0000     0.0000     0.0100 |     4.2857
after eliminating column 3
      0.1500     0.1800     0.0000     0.2100 |   480.0000
      0.0000     0.2800     0.0000    -0.2100 |    10.0000
      0.0000     0.0000     0.6000    -0.3450 |   186.4286
      0.0000     0.0000     0.0000     0.0100 |     4.2857

fertilizer      g per tree   kg for 40
NPK 15-15-15        2171.4       86.86
DAP 18-46-0          357.1       14.29
KCl 0-0-60           557.1       22.29
ZA 21-0-0-24S        428.6       17.14
largest residual: 1.421e-14 g
  N     wanted  480.0 g, gets  480.000 g
  P2O5  wanted  490.0 g, gets  490.000 g
  K2O   wanted  660.0 g, gets  660.000 g
  S     wanted  320.0 g, gets  320.000 g
```

No row swaps were needed: the first column holds 0.15, 0.15, 0.15 and 0.10, and the first row already has the largest entry (ties keep the upper row). Each tree needs about 2.17 kg of NPK, 357 g of DAP, 557 g of KCl and 429 g of ZA a year; for the forty trees Bu Rahma buys about 87 kg of NPK, 14 kg of DAP, 22 kg of KCl and 17 kg of ZA, which she will split over the two or three applications that her extension officer recommends. The check at the bottom delivers every nutrient to within a thousandth of a gram, and the largest residual, 1.4×10^{-14} g, is a few units of round-off in numbers of size 500. (The `-0.0000` in the fourth row after the first stage is the right-hand side $320 - \frac{0.10}{0.15} \times 480$, which is zero in exact arithmetic but a tiny negative number after rounding; it is printed with its sign.)

There is one surprise, and it leads to the most important idea of this chapter. The extension officer's recommendation was rounded: perhaps the phosphate target is really 500 g, not 490. Changing only that one number and solving again (Exercise 4.5) gives 1,143 g of NPK instead of 2,171, and 714 g of DAP instead of 357. A change of 2% in one target changes two of the answers by a factor of two. And the target 480 g of phosphate, only 2% below 490, gives the answer "3,200 g of NPK, 0 g of DAP, 300 g of KCl, 0 g of ZA", while any lower phosphate target gives *negative* amounts, which cannot be bought. The arithmetic is not wrong; the system is *sensitive*. Section 4.8 measures that sensitivity.

4.8 Conditioning: A Small Residual Is Not a Small Error

4.8.1 Two lines that are almost parallel

With two unknowns, each linear equation is a straight line in the plane, and the solution is the point where the lines cross. **Figure 4.2** shows two systems. On the left, $x + y = 2$ and $x - y = 0$

cross at a right angle, and moving either line a little moves the crossing a little. On the right, $x + y = 2$ and $x + 1.01y = 2.01$ are almost parallel. They still cross at (1,1), but changing the second right-hand side by only 0.01, to 2.02, moves the crossing all the way to (0,2).

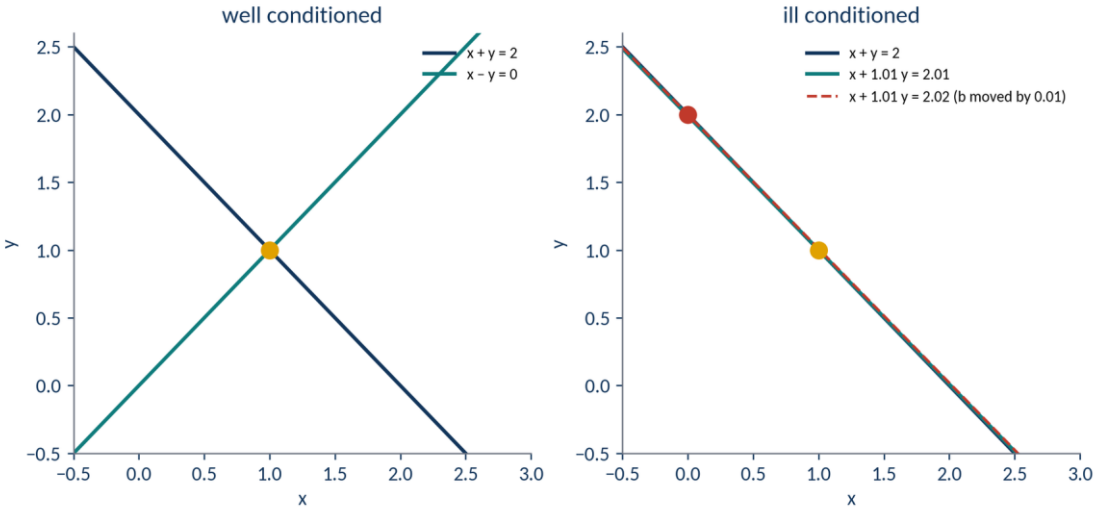


Figure 4.2 A well-conditioned system (left) and an ill-conditioned one (right): when the lines are almost parallel, a change of 0.01 in the data moves the solution from (1,1) to (0,2).

A system whose solution changes a lot when its data change a little is called **ill-conditioned**. The property belongs to the *problem*, not to the method: no algorithm can find the crossing of two almost parallel lines accurately if the lines themselves are only known to two decimals. Even with exact arithmetic, the uncertainty of the data becomes a large uncertainty of the answer. On a computer, the data are always slightly uncertain, because they are rounded when stored (Section 1.10), so an ill-conditioned system also amplifies round-off.

4.8.2 The condition number

To measure the effect we need a size for vectors and matrices. The **infinity norm** of a vector is its largest entry in absolute value, $\| x \|_\infty = \max_i |x_i|$, and the induced norm of a matrix is its largest row sum of absolute values, $\| A \|_\infty = \max_i \sum_j |a_{ij}|$. The **condition number** of a non-singular matrix is

$$\kappa(A) = \| A \| \, \| A^{-1} \| \tag{4.12}$$

It is at least 1, and it bounds how much a relative change in b can be magnified in the solution. If $Ax = b$ and $A(x + \delta x) = b + \delta b$, then $A\,\delta x = \delta b$, so $\| \delta x \| \le \| A^{-1} \| \, \| \delta b \|$, and $\| b \| \le \| A \| \, \| x \|$; dividing the first inequality by the second gives [2]

$$\frac{\| \delta x \|}{\| x \|} \le \kappa(A) \, \frac{\| \delta b \|}{\| b \|} \tag{4.13}$$

For the almost parallel lines, $A = \begin{pmatrix} 1 & 1 \\ 1 & 1.01 \end{pmatrix}$ has the inverse $\frac{1}{0.01}\begin{pmatrix} 1.01 & -1 \\ -1 & 1 \end{pmatrix}$, so $\| A \|_\infty = 2.01$, $\| A^{-1} \|_\infty = 201$ and $\kappa = 404$. A change of 0.5% in b (0.01 in 2.01) may change x by up to $404 \times 0.5\% = 200\%$. The actual change, from (1,1) to (0,2), is 1 in the infinity norm, 100% of $\| x \|_\infty = 1$: well inside the bound, and still enormous.

A useful rule of thumb follows from Equation (4.13) and the unit round-off of Chapter 1: since the data carry relative errors of about 10^{-16}, **a solution computed in double precision can lose**

about $\log_{10}\kappa(A)$ of its 16 significant digits [3]. With $\kappa = 10^4$, about 12 digits remain; with $\kappa = 10^{16}$, none.

The fertilizer matrix Equation (4.11) has $\kappa_\infty = 447$ (Section 4.11 computes it). Changing the phosphate target by 10 g is a change of $10/660 = 1.5\%$ in $\| b \|_\infty$; Equation (4.13) allows up to $447 \times 1.5\% = 677\%$ in x, and the actual change, 1,029 g out of 2,171 g, is 47%. The bound is not reached, but it explains why a rounded recommendation gives such different answers. The chemistry explains it too: NPK and DAP both deliver nitrogen and phosphate, in similar proportions for nitrogen, so the columns of NPK and DAP are *almost parallel* in the nitrogen-phosphate plane, like the lines of **Figure 4.2**.

4.8.3 Residual and error

The **residual** of a computed solution $\tilde{x}$ is $r = b - A\tilde{x}$: how well $\tilde{x}$ satisfies the equations. It is easy to compute, and our programs print it. The **error** is $\tilde{x} - x$: how far $\tilde{x}$ is from the true solution, which we do not know. The relation between them is again the condition number: $\| \tilde{x} - x \| / \| x \| \le \kappa(A) \, \| r \| / \| b \|$. Gaussian elimination with partial pivoting always produces a *small residual*, of the order of round-off; whether the error is small depends on κ.

The classic demonstration uses the **Hilbert matrix**, $H_{ij} = 1/(i + j - 1)$, which Hilbert met in 1894 in a problem of approximating functions and which is notoriously ill-conditioned. **Listing 4.9** and **Listing 4.10** choose the right-hand side so that the exact solution is all ones, solve, and print the condition number (computed from the inverse, one column at a time), the residual and the error.

Listing 4.9 `conditioning.py`: Hilbert matrices of growing size: condition number, residual and error.

```
"""Small residual, large error: Hilbert matrices H_ij = 1 / (i + j - 1).
The exact answer is chosen to be all ones, so the error can be measured."""
from gauss import residual, solve


def inverse_norm(a):
    """||A^-1|| in the infinity norm, one column of A^-1 at a time."""
    n = len(a)
    cols = []
    for j in range(n):
        e = [1.0 if i == j else 0.0 for i in range(n)]
        cols.append(solve(a, e))
    return max(sum(abs(cols[j][i]) for j in range(n)) for i in range(n))


def main():
    print(f"{'n':>3} {'cond(H)':>10} {'residual':>10} {'max error':>10}")
    for n in range(2, 14, 2):
        h = [[1.0 / (i + j + 1) for j in range(n)] for i in range(n)]
        b = [sum(row) for row in h]                  # so that x = all ones
        x = solve(h, b)
        err = max(abs(v - 1.0) for v in x)
        norm = max(sum(abs(v) for v in row) for row in h)
        cond = norm * inverse_norm(h)
        print(f"{n:>3} {cond:>10.2e} {residual(h, x, b):>10.2e} {err:>10.2e}")


if __name__ == "__main__":
    main()
```

Listing 4.10 `Conditioning.java`: the same experiment in Java.

```
import java.util.Locale;

/**
 * Small residual, large error: Hilbert matrices H_ij = 1 / (i + j - 1).
 * The exact answer is chosen to be all ones, so the error can be measured.
 */
public class Conditioning {
    /** ||A^-1|| in the infinity norm, one column of A^-1 at a time. */
    static double inverseNorm(double[][] a) {
        int n = a.length;
        double[][] cols = new double[n][];
        for (int j = 0; j < n; j++) {
            double[] e = new double[n];
            e[j] = 1.0;
            cols[j] = Gauss.solve(a, e);
        }
        double best = 0;
        for (int i = 0; i < n; i++) {
            double s = 0;
            for (int j = 0; j < n; j++) {
                s += Math.abs(cols[j][i]);
            }
            best = Math.max(best, s);
        }
        return best;
    }

    public static void main(String[] args) {
        Locale.setDefault(Locale.ROOT);
        System.out.printf("%3s %10s %10s %10s%n", "n", "cond(H)", "residual",
                          "max error");
        for (int n = 2; n < 14; n += 2) {
            double[][] h = new double[n][n];
            double[] b = new double[n];
            double norm = 0;
            for (int i = 0; i < n; i++) {
                double rowSum = 0, absSum = 0;
                for (int j = 0; j < n; j++) {
                    h[i][j] = 1.0 / (i + j + 1);
                    rowSum += h[i][j];
                    absSum += Math.abs(h[i][j]);
                }
                b[i] = rowSum;                    // so that x = all ones
                norm = Math.max(norm, absSum);
            }
            double[] x = Gauss.solve(h, b);
```

Listing 4.10 (Continued) *Conditioning.java: the same experiment in Java.*

```
            double err = 0;
            for (double v : x) {
                err = Math.max(err, Math.abs(v - 1.0));
            }
            double cond = norm * inverseNorm(h);
            System.out.printf("%3d %10.2e %10.2e %10.2e%n", n, cond,
                              Gauss.residual(h, x, b), err);
        }
    }
}
```

```
  n    cond(H)   residual  max error
  2   2.70e+01   1.11e-16   6.66e-16
  4   2.84e+04   1.67e-16   6.11e-13
  6   2.91e+07   8.33e-17   5.26e-10
  8   3.39e+10   1.67e-16   4.19e-07
 10   3.54e+13   1.80e-16   3.18e-04
 12   3.83e+16   4.30e-16   3.56e-01
```

The residual stays at round-off level, about 10^{-16}, for every size: each computed solution satisfies its equations to the last digit. But the error grows with the condition number exactly as the rule of thumb predicts: $\kappa = 3 \times 10^{7}$ for $n = 6$ and an error of 5×10^{-10} (about 7 of 16 digits lost); $\kappa = 3.5 \times 10^{13}$ for $n = 10$ and an error of 3×10^{-4}; $\kappa = 3.8 \times 10^{16}$ for $n = 12$ and an error of 0.36, where the "solution" has no correct digit. A program that checked only the residual would declare all six answers perfect. (The condition number for $n = 12$ is itself only roughly right, because it was computed from an inverse that is equally inaccurate.)

4.9 Why Pivoting Matters, in Code

Listing 4.11 and **Listing 4.12** run the two examples of Section 4.4 with and without pivoting.

Listing 4.11 `pivot_demo.py`: the zero pivot and the tiny pivot, with and without partial pivoting.

```python
"""Why pivoting matters: a zero pivot and a tiny pivot."""
from gauss import solve


def main():
    print("1. The receipts with rice missing from the first one")
    a = [[0, 1, 2], [5, 2, 1], [3, 2, 3]]
    b = [52, 128, 132]
    try:
        solve(a, b, pivot=False)
    except ZeroDivisionError as err:
        print(f"   without pivoting: {err}")
    x = solve(a, b, pivot=True)
    print("   with pivoting   : " + ", ".join(f"{v:.6f}" for v in x))

    print("2. A tiny pivot: 1e-20 x + y = 1, x + y = 2 (answer near 1, 1)")
    a = [[1e-20, 1], [1, 1]]
    b = [1, 2]
    x = solve(a, b, pivot=False)
    print(f"   without pivoting: x = {x[0]:.6f}, y = {x[1]:.6f}")
    x = solve(a, b, pivot=True)
    print(f"   with pivoting   : x = {x[0]:.6f}, y = {x[1]:.6f}")


if __name__ == "__main__":
    main()
```

Listing 4.12 `PivotDemo.java`: the same demonstration in Java.

```
import java.util.Locale;

/** Why pivoting matters: a zero pivot and a tiny pivot. */
public class PivotDemo {
    public static void main(String[] args) {
        Locale.setDefault(Locale.ROOT);
        System.out.println("1. The receipts with rice missing from the first "
                           + "one");
        double[][] a = {{0, 1, 2}, {5, 2, 1}, {3, 2, 3}};
        double[] b = {52, 128, 132};
        try {
            Gauss.solve(a, b, false, false);
        } catch (ArithmeticException err) {
            System.out.println("   without pivoting: " + err.getMessage());
        }
        double[] x = Gauss.solve(a, b, true, false);
        System.out.printf("   with pivoting   : %.6f, %.6f, %.6f%n",
                          x[0], x[1], x[2]);

        System.out.println("2. A tiny pivot: 1e-20 x + y = 1, x + y = 2 "
                           + "(answer near 1, 1)");
        double[][] t = {{1e-20, 1}, {1, 1}};
        double[] c = {1, 2};
        x = Gauss.solve(t, c, false, false);
        System.out.printf("   without pivoting: x = %.6f, y = %.6f%n", x[0], x[1]);
        x = Gauss.solve(t, c, true, false);
        System.out.printf("   with pivoting   : x = %.6f, y = %.6f%n", x[0], x[1]);
    }
}
```

```
1. The receipts with rice missing from the first one
   without pivoting: zero pivot in column 1
   with pivoting   : 15.000000, 18.000000, 17.000000
2. A tiny pivot: 1e-20 x + y = 1, x + y = 2 (answer near 1, 1)
   without pivoting: x = 0.000000, y = 1.000000
   with pivoting   : x = 1.000000, y = 1.000000
```

Without pivoting, the reordered receipts Equation (4.8) stop with “zero pivot in column 1”, and the tiny pivot Equation (4.9) gives $x = 0$ instead of 1 with no warning at all, exactly as the hand computation of Section 4.4.2 predicted. With pivoting, both are solved correctly. The second case is the dangerous one: a method that fails loudly can be fixed, a method that returns a wrong number with confidence cannot even be noticed. Our `solve` therefore pivots by default, and only the teaching trace of the receipts turns it off.

4.10 Testing It

Listing 4.13 and **Listing 4.14** check the receipts, check that the caller’s matrix is not changed, check that pivoting handles a zero pivot and that a singular matrix is refused, and check the flop count for $n = 3$ against Equation (4.6): $\frac{2}{3} \cdot 27 + \frac{3}{2} \cdot 9 - \frac{7}{6} \cdot 3 = 18 + 13.5 - 3.5 = 28$.

Listing 4.13 `test_gauss.py`: unit tests of Gaussian elimination.

```python
"""Tests of Gaussian elimination: python -m unittest test_gauss.py"""
import unittest
import gauss


class GaussTest(unittest.TestCase):
    def test_receipts(self):
        a = [[2, 1, 2], [5, 2, 1], [3, 2, 3]]
        b = [82, 128, 132]
        x = gauss.solve(a, b, pivot=False)
        for got, want in zip(x, (15, 18, 17)):
            self.assertAlmostEqual(got, want, places=12)
        self.assertEqual(a[0][0], 2)          # the input is not changed

    def test_zero_pivot_needs_pivoting(self):
        x = gauss.solve([[0, 1], [1, 1]], [1, 2])
        self.assertAlmostEqual(x[0], 1.0)
        self.assertAlmostEqual(x[1], 1.0)
        with self.assertRaises(ZeroDivisionError):
            gauss.solve([[0, 1], [1, 1]], [1, 2], pivot=False)

    def test_singular(self):
        with self.assertRaises(ZeroDivisionError):
            gauss.solve([[1, 2], [2, 4]], [3, 6])

    def test_flop_count(self):
        gauss.solve([[4, 1, 0], [1, 4, 1], [0, 1, 4]], [1, 1, 1])
        self.assertEqual(gauss.flops, 28)


if __name__ == "__main__":
    unittest.main()
```

Listing 4.14 `GaussTest.java`: the same checks in Java.

```java
/** Tests of Gaussian elimination: java -cp out GaussTest */
public class GaussTest {
    static int failures = 0;

    static void check(boolean ok, String name) {
        System.out.println((ok ? "ok    " : "FAILED") + "  " + name);
        if (!ok) {
            failures++;
        }
    }

    public static void main(String[] args) {
        double[][] a = {{2, 1, 2}, {5, 2, 1}, {3, 2, 3}};
        double[] b = {82, 128, 132};
        double[] x = Gauss.solve(a, b, false, false);
        check(Math.abs(x[0] - 15) < 1e-12 && Math.abs(x[1] - 18) < 1e-12
              && Math.abs(x[2] - 17) < 1e-12, "receipts give 15, 18, 17");
        check(a[0][0] == 2 && b[0] == 82, "the input is not changed");
        double[][] z = {{0, 1}, {1, 1}};
        double[] y = Gauss.solve(z, new double[] {1, 2});
        check(Math.abs(y[0] - 1) < 1e-15 && Math.abs(y[1] - 1) < 1e-15,
              "pivoting handles a zero pivot");
        try {
            Gauss.solve(new double[][] {{1, 2}, {2, 4}}, new double[] {3, 6});
            check(false, "a singular matrix is refused");
        } catch (ArithmeticException e) {
            check(true, "a singular matrix is refused");
        }
        Gauss.solve(new double[][] {{4, 1, 0}, {1, 4, 1}, {0, 1, 4}},
                    new double[] {1, 1, 1});
        check(Gauss.flops == 28, "28 flops for n = 3");
        System.out.println(failures == 0 ? "all tests passed"
                                         : failures + " test(s) failed");
    }
}
```

```
$ python3 -m unittest test_gauss.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
$ java -cp out GaussTest
```

```
ok      receipts give 15, 18, 17
ok      the input is not changed
ok      pivoting handles a zero pivot
ok      a singular matrix is refused
ok      28 flops for n = 3
all tests passed
```

The singular test deserves a remark. For $\begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix}$, pivoting swaps the rows, the multiplier is 0.5, and the second pivot is $2 - 0.5 \times 4 = 0$ *exactly*, so the program detects it. For a singular matrix whose entries are not exactly representable, round-off usually leaves a pivot of about 10^{-16} instead of zero, the program happily divides by it, and the "solution" is enormous. Exact zero tests catch only the clean cases; the condition number catches the rest.

4.11 Comparing with NumPy

NumPy's `np.linalg.solve` calls the LAPACK routine `gesv`, which performs Gaussian elimination with partial pivoting in the form of an LU factorization (Chapter 6) using highly optimized block

operations [5]. **Listing 4.15** solves the fertilizer system with it, computes the condition number in two norms, and repeats the Hilbert experiment with SciPy's `hilbert` function.

Listing 4.15 `compare_numpy.py`: the fertilizer system and the Hilbert matrices with NumPy and SciPy.

```
"""The fertilizer system and the Hilbert matrices with NumPy (needs NumPy)."""
import numpy as np
import scipy.linalg

A = np.array([[0.15, 0.18, 0.00, 0.21],
              [0.15, 0.46, 0.00, 0.00],
              [0.15, 0.00, 0.60, 0.00],
              [0.10, 0.00, 0.00, 0.24]])
T = np.array([480.0, 490.0, 660.0, 320.0])


def main():
    x = np.linalg.solve(A, T)                    # LAPACK gesv: LU + pivoting
    print("np.linalg.solve :", np.array2string(x, precision=1))
    print(f"cond(A), 2-norm : {np.linalg.cond(A):.1f}")
    print(f"cond(A), inf    : {np.linalg.cond(A, np.inf):.1f}")
    for n in (6, 12):
        h = scipy.linalg.hilbert(n)
        xh = np.linalg.solve(h, h.sum(axis=1))
        print(f"Hilbert n = {n:>2}: cond {np.linalg.cond(h, np.inf):.2e}, "
              f"max error {np.max(np.abs(xh - 1)):.2e}")


if __name__ == "__main__":
    main()
```

```
np.linalg.solve : [2171.4  357.1  557.1  428.6]
cond(A), 2-norm : 265.7
cond(A), inf    : 446.8
Hilbert n =  6: cond 2.91e+07, max error 2.60e-10
Hilbert n = 12: cond 3.99e+16, max error 5.16e-01
```

NumPy's amounts agree with ours to the printed gram. Its condition number in the infinity norm, 446.8, is the value used in Section 4.8.2; in the 2-norm, the default of `np.linalg.cond`, it is 265.7. The two norms measure size differently and give different numbers, but always of the same order. For the Hilbert matrices, NumPy's errors (2.6×10^{-10} for $n = 6$, 0.52 for $n = 12$) are of the same size as ours (5.3×10^{-10} and 0.36), not smaller: professional software cannot beat the condition number either. What it offers is speed, and stability safeguards that matter for large matrices.

4.12 Numerical Methods in Practice: Profitina

Profitina's AI Visibility Engine scores a website from many signals, such as how extractable its content is, how complete its structured data are, and how technically sound it is, and it explains each signal. A score built from many signals is, at its heart, a weighted sum: each normalized signal times its weight, added up. The platform does not publish its weights, and the details of how it sets them are its own; but the mathematical shape of the question "which weights would reproduce these reference assessments?" is a linear system, with one equation per reference site and one unknown per weight. When there are exactly as many reliable references as weights, it is a square system like the ones in this chapter; when there are more, it becomes the least-squares problem of Chapter 8.

This chapter's warning applies directly. Two signals that measure almost the same thing (say, two different checks that both mostly detect whether a page has a structured-data block) are like the NPK and DAP columns of Equation (4.11): nearly parallel. The system for their weights is ill-conditioned, and a tiny change in the reference data can swing one weight up and the other down while the scores themselves barely change. The symptoms are the same as in the orchard: a small residual, large and unstable coefficients, and sometimes negative weights that make no sense. The remedies are the same too: measure the condition number, merge or drop signals that duplicate each other, and prefer a method that stays stable, all of which is general practice in any scoring system that combines measurements.

4.13 Using It for Real

The two programs adapt to many everyday problems by changing only their data files.

- **Your own receipts.** Any set of n receipts with n items whose prices did not change solves for the prices. Make sure the receipts are genuinely different: two receipts with the same proportions (one exactly double the other) make the matrix singular.
- **Your own fertilizers.** Replace the lines of `fertilizer.csv` with the fertilizers your shop sells, from the percentages on their bags, and the last line with the targets from your soil test and extension officer. The program works for any number of fertilizers as long as there are as many fertilizers as nutrients. If the answer contains a negative amount, the target cannot be met exactly with those fertilizers: add a different fertilizer (for example urea, "46-0-0", for extra nitrogen), or accept a target that is met approximately, which is the least-squares approach of Chapter 8.
- **Check the sensitivity.** Change each target by its uncertainty, perhaps 5%, and solve again. If the amounts change a lot, the fertilizers overlap too much (a large condition number); choosing fertilizers whose nutrient profiles are more different, such as single-nutrient fertilizers, makes the system well-conditioned and the plan robust.
- **Feed ratios and mixtures.** The same program mixes feed for chickens or catfish from ingredients with known protein, energy and fiber contents, blends two grades of coffee or rice to a target price and quality, or finds the concentrations of solutions from several measurements. Every one of them is $Ax = b$.
- **Size.** Our programs are fine up to a few hundred unknowns. For thousands, use NumPy (`np.linalg.solve`) or a Java linear-algebra library, which use the same method with far faster inner loops; for sparse systems with millions of unknowns, see Chapter 5.

4.14 Common Mistakes, and How to Avoid Them

- **Elimination without pivoting.** It fails on a zero pivot and, worse, returns wrong answers without warning on a tiny one. Always use partial pivoting unless you know the matrix does not need it (for example, a diagonally dominant matrix).
- **Mixing up rows and columns.** In the fertilizer problem the file has one fertilizer per line, but the matrix needs one nutrient per row. Write down what row i and column j mean before building the matrix, and check one equation by hand.
- **Judging a solution by its residual.** Gaussian elimination with pivoting always gives a small residual. Whether the solution is accurate depends on the condition number; estimate it, and expect to lose about $\log_{10}\kappa$ digits.

- **Computing the inverse to solve a system.** $x = A^{-1}b$ is correct in algebra, but computing A^{-1} first costs three times as much as elimination and is less accurate. Solve $Ax = b$ directly; compute an inverse only when its entries themselves are needed.
- **Trusting a tiny determinant or ignoring a large one.** The determinant of 10^{-3} times the identity matrix is 10^{-3n}, tiny for large n, yet the matrix is perfectly conditioned. Use the condition number, not the determinant, to judge sensitivity.
- **Modifying the caller's data.** Elimination overwrites the matrix. Copy it first, as our `solve` does, or document clearly that it is destroyed.
- **Too much faith in the data.** An answer can be no better than the targets and percentages that went into it. When the system is ill-conditioned, small rounding in the data becomes large changes in the answer.

4.15 Chapter Summary

- A system of n linear equations in n unknowns is written $Ax = b$ (Equation (4.2)); each equation is a row of the augmented matrix $[A \mid b]$.
- Gaussian elimination makes the matrix upper triangular with multipliers $m_{ik} = a_{ik}/a_{kk}$ (Equation (4.3)) and row updates (Equation (4.4)), then finds the unknowns from the last to the first by back substitution (Equation (4.5)); **Pseudocode 4.1** combines both.
- It costs about $\frac{2}{3}n^3$ flops (Equation (4.6)): doubling n multiplies the work by eight. Back substitution costs only n^2.
- A zero pivot stops plain elimination and a tiny pivot silently ruins it (Equation (4.8), Equation (4.9)); partial pivoting (Equation (4.10)) keeps every multiplier at most 1 in size and is stable in practice. The determinant is the signed product of the pivots (Equation (4.7)).
- The condition number $\kappa(A) = \| A \| \| A^{-1} \|$ (Equation (4.12)) bounds how much relative changes in the data are magnified in the solution (Equation (4.13)); about $\log_{10}\kappa$ digits are lost. A small residual does not mean a small error.
- The receipts gave Rp15,000, Rp18,000 and Rp17,000 exactly; the orchard's four fertilizers came out at 2,171, 357, 557 and 429 g per tree, in agreement with NumPy, but the system is moderately ill-conditioned ($\kappa_\infty = 447$), so rounded targets change the plan considerably.

4.16 Exercises

1. Solve by hand, with elimination and back substitution, the receipts: 1 kg of rice, 2 kg of eggs and 1 kg of shallots for Rp111,000; 3 kg of rice, 1 kg of eggs and 2 kg of shallots for Rp153,000; 2 kg of rice, 3 kg of eggs and 4 kg of shallots for Rp274,000.
2. Solve the reordered receipts Equation (4.8) by hand with partial pivoting. Write down each swap and each multiplier, and check that every multiplier is at most 1 in size.
3. Compute the determinant of the receipt matrix of Exercise 1 from its pivots with Equation (4.7), and check it with the usual 3×3 formula.
4. Using Equation (4.6), how many flops does Gaussian elimination need for $n = 1{,}000$ and for $n = 10{,}000$? If a computer performs 10^{10} flops per second on this kind of work, how long does each take?
5. Change the phosphate target in `fertilizer.csv` from 490 to 500 and solve again. Then restore it and change the potash target from 660 to 700. Which change moves the plan more, and why?

6. For the 2×2 matrix $\begin{pmatrix} 1 & 1 \\ 1 & 1+\varepsilon \end{pmatrix}$, compute κ_∞ as a function of ε. How small can ε be before a solution in double precision has no correct digit?
7. Explain, using the Hilbert results of Section 4.8.3, why a program should never report "solution verified" only because the residual is small. What could it report instead?

4.17 Solutions to the Exercises

1. The augmented matrix is (1,2,1 | 111), (3,1,2 | 153), (2,3,4 | 274), in thousands of rupiah. With pivot 1 and multipliers 3 and 2: row 2 becomes $(0,-5,-1 \mid 153-333=-180)$ and row 3 becomes $(0,-1,2 \mid 274-222=52)$. With pivot -5 and multiplier $(-1)/(-5)=0.2$: row 3 becomes $(0,0,2+0.2 \mid 52-0.2\times(-180)) = (0,0,2.2 \mid 88)$. Back substitution: $x_3 = 88/2.2 = 40$; $-5x_2 - 40 = -180$, so $x_2 = 28$; $x_1 + 56 + 40 = 111$, so $x_1 = 15$. Rice Rp15,000, eggs Rp28,000, shallots Rp40,000 per kg. (Without pivoting the program follows exactly these steps; with pivoting it would first swap row 2 to the top.)

2. Column 1 holds 0,5,3: the largest is 5, in row 2, so swap rows 1 and 2: (5,2,1 | 128), (0,1,2 | 52), (3,2,3 | 132). Multipliers $0/5=0$ and $3/5=0.6$: row 3 becomes $(0,2-1.2,3-0.6 \mid 132-76.8) = (0,0.8,2.4 \mid 55.2)$. Column 2 below the diagonal holds 1 and 0.8: the largest is already in place, no swap. Multiplier $0.8/1=0.8$: row 3 becomes $(0,0,2.4-1.6 \mid 55.2-41.6) = (0,0,0.8 \mid 13.6)$. Back substitution: $x_3 = 17$, $x_2 = 52-34 = 18$, $5x_1 = 128-36-17 = 75$, $x_1 = 15$. The multipliers were 0, 0.6 and 0.8, all at most 1.

3. No swaps, pivots 1, -5 and 2.2: $\det = 1\times(-5)\times 2.2 = -11$. The 3×3 formula: $1(1\cdot 4-2\cdot 3)-2(3\cdot 4-2\cdot 2)+1(3\cdot 3-1\cdot 2) = -2-16+7 = -11$.

4. For $n = 1{,}000$: $\frac{2}{3}\times 10^9 + 1.5\times 10^6 - 1{,}167 \approx 6.68\times 10^8$ flops, about 0.07 s at 10^{10} flops per second. For $n = 10{,}000$: $\approx 6.67\times 10^{11}$ flops, about 67 s, a thousand times longer for ten times the size. The matrix alone then needs 10^8 doubles, 800 MB of memory, which is often the real limit.

5. Phosphate 500 (with the rest unchanged) gives NPK 1,142.9 g, DAP 714.3 g, KCl 814.3 g and ZA 857.1 g: every amount changes, NPK by -47% and DAP by $+100\%$. Potash 700 gives NPK 2,171.4 g, DAP 357.1 g, KCl 623.8 g and ZA 428.6 g: only KCl changes, by $40/0.60 = 66.7$ g. Potash comes almost only from KCl (NPK also contains some, but the NPK amount is fixed by the other three equations), so its equation is well separated from the others. Phosphate comes from NPK and DAP, which also share nitrogen, and those two columns are nearly parallel: the ill-conditioned part of the system.

6. $\det = \varepsilon$, $A^{-1} = \frac{1}{\varepsilon}\begin{pmatrix} 1+\varepsilon & -1 \\ -1 & 1 \end{pmatrix}$, so $\| A \|_\infty = 2+\varepsilon$ and $\| A^{-1} \|_\infty = (2+\varepsilon)/\varepsilon$, giving $\kappa_\infty = (2+\varepsilon)^2/\varepsilon \approx 4/\varepsilon$. With about 16 digits available, all are lost when $4/\varepsilon \approx 10^{16}$, that is $\varepsilon \approx 4\times 10^{-16}$; below about 2.2×10^{-16}, the matrix entry $1+\varepsilon$ is not even representable and rounds to 1, making the stored matrix singular.

7. For $n = 12$ the residual was 4.3×10^{-16}, as small as for $n = 2$, but the error was 0.36: the "verified" solution had no correct digit. A small residual only shows that $\tilde{x}$ solves a system *very close to* the given one; if the system is ill-conditioned, very close systems have very different solutions. A responsible program reports the residual together with an estimate of the condition number, and from both an estimate of the number of correct digits, $16 - \log_{10}\kappa$, warning when it is small. NumPy offers `np.linalg.cond` for this; LAPACK has routines that estimate κ cheaply during the solve.

References

[1] J. F. Grcar, “How ordinary elimination became Gaussian elimination,” *Historia Mathematica*, vol. 38, no. 2, pp. 163–218, 2011.

[2] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed. Baltimore, MD, USA: Johns Hopkins University Press, 2013.

[3] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: SIAM, 2002.

[4] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical Analysis*, 10th ed. Boston, MA, USA: Cengage Learning, 2016.

[5] NumPy Developers, “numpy.linalg.solve,” *NumPy Reference*, 2026. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.linalg.solve.html

Chapter 5

Linear Systems II: Jacobi, Gauss-Seidel and SOR

Chapter 4 solved a linear system the way a careful clerk works through a ledger: eliminate, eliminate, substitute back, done, in a number of steps fixed in advance. That is the right tool for a few hundred unknowns. But many real systems have tens of thousands or millions of unknowns, and in almost all of them each equation mentions only a handful of the unknowns. For such systems a completely different idea wins: guess an answer, then improve every unknown in turn from its neighbors, and repeat until nothing moves. This chapter builds that idea from a flat iron griddle at a martabak cart. By the end of the chapter you will be able to turn a physical "each point is the average of its neighbors" rule into a linear system; carry out Jacobi, Gauss-Seidel and successive over-relaxation (SOR) sweeps by hand; write them as matrix splittings and prove when they converge; predict how many sweeps they need from the spectral radius; choose the relaxation factor ω; stop at the right moment, knowing how far you still are from the answer; build all three methods in Java and Python; and check the results against SciPy's sparse solvers.

5.1 Pak Joni's Griddle

Every evening at five, Pak Joni lights the gas under the flat iron griddle of his martabak and roti bakar cart. The griddle is a square plate of thick iron, forty centimeters on a side. A long burner runs under its left edge; a second, smaller burner keeps the front edge warm so that finished martabak can wait there; the back edge, near the tins of margarine, and the right edge, where he cuts the bread, get no flame at all. After a quarter of an hour the plate stops getting hotter. It has reached a *steady state*: heat flows in from the burners and out through the cool edges at the same rate, and the temperature at each spot no longer changes.

Pak Joni knows his griddle the way a driver knows a road. Martabak telur wants a hot spot, but not the hottest one, or the egg burns before the dough is crisp; roti bakar wants a gentler place; the far right corner is only good for keeping things warm. He found these places over years, one burned martabak at a time. His nephew, who has just started helping, keeps putting the dough in the wrong place, and a new griddle with a different burner would mean learning everything again.

What he would like to have is a map: the temperature at every spot of the plate, with the zone between 140 and 170 degrees Celsius, where martabak telur cooks best, marked on it. An infrared thermometer can measure a few points, but a map needs hundreds, and measuring a new griddle before buying it is not possible at all. This chapter computes the map from four numbers: the temperatures along the four edges. Pak Joni measured them with a borrowed infrared thermometer: about 200 °C over the main burner on the left, 150 °C along the front edge, 80 °C at the back and 60 °C on the right.

The physics behind the map is old and simple, and it leads to a linear system of exactly the kind Chapter 4 taught us to solve, except that it is large. A useful map of a 40 cm plate needs a point every centimeter or two, which means 400 to 1,600 unknowns; a careful engineering model of the same plate would use a million. Each equation, however, involves only five of the unknowns. This

combination, many unknowns and very few per equation, is what the methods of this chapter are made for.

5.2 The Idea in Plain Words

5.2.1 Each point is the average of its neighbors

Imagine the plate divided by a grid of lines into small squares, and look at one point where the lines cross. In the steady state no heat piles up at that point: what flows in from the hotter neighbors flows out to the cooler ones. For a thin uniform plate this balance has a beautiful consequence: *the temperature at each inner point equals the average of the temperatures at its four neighbors*, left, right, above and below. If the neighbors are 120, 100, 90 and 110 °C, the point itself is at 105 °C.

Why the average? Heat flows from hot to cold at a rate proportional to the temperature difference (Fourier's law). The heat flowing from the left neighbor into our point is proportional to $T_{\text{left}} - T$, and similarly for the other three. In the steady state the four flows add up to zero:

$$(T_{\text{left}} - T) + (T_{\text{right}} - T) + (T_{\text{up}} - T) + (T_{\text{down}} - T) = 0,$$

and solving for T gives

$$T = \frac{T_{\text{left}} + T_{\text{right}} + T_{\text{up}} + T_{\text{down}}}{4}. \tag{5.1}$$

The same equation comes out of the heat equation of physics, Laplace's equation $\partial^2 T/\,\partial x^2 + \partial^2 T/\,\partial y^2 = 0$, when each second derivative is replaced by its central difference, which Chapter 9 derives: $\partial^2 T/\,\partial x^2 \approx (T_{\text{left}} - 2T + T_{\text{right}})/h^2$ for grid spacing h [1]. The error of this replacement shrinks like h^2, so a finer grid gives a more faithful map. Equation (5.1) is the whole of the physics we need. It ignores the heat that leaves through the top of the plate into the air, so it is a first model, not the last word; Section 5.9 says how to check it with a thermometer.

5.2.2 Four points by hand

Start with the coarsest grid that has anything inside: two lines across and two lines down, which leave four inner points. **Figure 5.1** names them T_1 (top left), T_2 (top right), T_3 (bottom left) and T_4 (bottom right). Every inner point touches two edges and two other inner points.

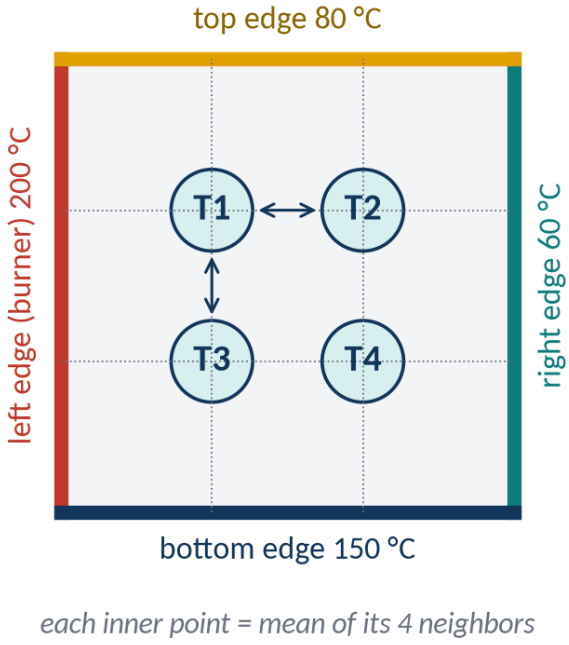


Figure 5.1 The coarsest griddle: four unknown inner temperatures, each the mean of its four neighbors; two of the neighbors are always edges with known temperatures.

Write Equation (5.1) for T_1. Its neighbors are the left edge (200), the top edge (80), T_2 and T_3, so $4T_1 = 200 + 80 + T_2 + T_3$. Moving the unknowns to the left and doing the same for the other three points gives four equations:

$$\begin{aligned} 4T_1 - T_2 - T_3 &= 280 \\ -T_1 + 4T_2 - T_4 &= 140 \\ -T_1 + 4T_3 - T_4 &= 350 \\ -T_2 - T_3 + 4T_4 &= 210 \end{aligned} \tag{5.2}$$

The right-hand sides are sums of edge temperatures: 280 is left plus top, 140 is right plus top, 350 is left plus bottom and 210 is right plus bottom. Gaussian elimination from Chapter 4 solves Equation (5.2) exactly: $T_1 = 131.25$, $T_2 = 96.25$, $T_3 = 148.75$, $T_4 = 113.75$. We will now reach the same answer in a completely different way, and see why that way scales to a million points while elimination does not.

5.2.3 Jacobi: everybody updates from the old picture

Guess that the whole inside of the plate is at 0 °C. The guess is terrible, but it does not matter. Now apply Equation (5.1) to every inner point, using the *old* guess for every neighbor:

- $T_1 = (280 + T_2 + T_3)/4 = (280 + 0 + 0)/4 = 70$
- $T_2 = (140 + T_1 + T_4)/4 = 140/4 = 35$
- $T_3 = (350 + T_1 + T_4)/4 = 350/4 = 87.5$
- $T_4 = (210 + T_2 + T_3)/4 = 210/4 = 52.5$

That was one *sweep*. The new picture (70, 35, 87.5, 52.5) is still wrong, because the neighbors we used were wrong, but it is less wrong: the edges have started to "leak" their heat inward. Do it again, now with the numbers of the first sweep as the old picture:

- $T_1 = (280 + 35 + 87.5)/4 = 402.5/4 = 100.625$
- $T_2 = (140 + 70 + 52.5)/4 = 262.5/4 = 65.625$
- $T_3 = (350 + 70 + 52.5)/4 = 472.5/4 = 118.125$
- $T_4 = (210 + 35 + 87.5)/4 = 332.5/4 = 83.125$

This is the **Jacobi method**, named after Carl Gustav Jacobi, who used it in 1845. Its rule is easy to say: in each sweep every unknown is recomputed from its own equation, and every other unknown in that equation takes its value from the previous sweep. Look at how far each value moved: in the first sweep the largest move was 87.5, in the second 30.625. The program of Section 5.5 continues the table; the moves go 87.5, 30.6, 15.3, 7.7, 3.8, 1.9, each one half of the previous one after the first two. After 27 sweeps no value moves by as much as a millionth of a degree, and the four temperatures agree with the exact ones to six decimals.

5.2.4 Gauss-Seidel: use the newest numbers at once

Something in the Jacobi sweep is wasteful. When we compute T_2 we already know the new T_1 (70), yet we use the old one (0). Carl Friedrich Gauss did not waste information like that in his own hand computations, and Philipp Ludwig von Seidel wrote the method down in 1874. The **Gauss-Seidel method** is Jacobi with one change: every new value is used as soon as it has been computed. From the same start of zeros:

- $T_1 = (280 + 0 + 0)/4 = 70$
- $T_2 = (140 + \mathbf{70} + 0)/4 = 210/4 = 52.5$
- $T_3 = (350 + \mathbf{70} + 0)/4 = 420/4 = 105$
- $T_4 = (210 + \mathbf{52.5} + \mathbf{105})/4 = 367.5/4 = 91.875$

The bold numbers are the fresh ones. The second sweep gives 109.375, 85.3125, 137.8125 and 108.28125, already closer to the answer than Jacobi's second sweep. The moves now shrink by a factor of four per sweep instead of two, and 15 sweeps are enough where Jacobi needed 27. There is a practical bonus: Gauss-Seidel overwrites each value in place and needs only one copy of the unknowns, where Jacobi needs two, the old picture and the new one.

5.2.5 SOR: go a little further than you are told

Watch the Gauss-Seidel values of T_1: 70, 109.4, 125.8, 129.9, 130.9, 131.2. Every step goes in the same direction, upward toward 131.25, and every step stops short. When a correction always falls short in the same direction, the obvious fix is to take a bigger step. **Successive over-relaxation** (SOR) computes the Gauss-Seidel value and then moves past it: if the old value is x_{old} and Gauss-Seidel proposes x_{GS}, SOR takes

$$x_{\text{new}} = x_{\text{old}} + \omega\,(x_{\text{GS}} - x_{\text{old}}),$$

where ω (omega) is the *relaxation factor*. With $\omega = 1$ this is Gauss-Seidel; with $\omega = 1.1$ each step is 10% longer. By hand, in the first SOR sweep with $\omega = 1.1$, T_1 goes from 0 to $0 + 1.1 \times (70 - 0) = 77$, and T_2 uses this 77: the Gauss-Seidel proposal is $(140 + 77 + 0)/4 = 54.25$, and SOR takes $1.1 \times 54.25 = 59.675$. With $\omega = 1.1$ the four-point griddle converges in 10 sweeps. With $\omega = 1.5$ it needs 29, more than Gauss-Seidel: stepping too far overshoots, and the values then swing back and forth around the answer. There is a best ω, and Section 5.4 shows how to find it. For the four-point griddle it is 1.072; for the 400-point griddle it is 1.741 and it cuts the work from 683 sweeps to 80.

KEY INSIGHT: An iterative method never solves the system in one go. It keeps a complete guess of the answer and improves it sweep after sweep, each sweep costing about as much as checking the equations once. Whether that is a good deal depends on two questions: does the guess get better every time, and how quickly? The rest of this chapter answers both.

5.3 The Methods Precisely

5.3.1 A general linear system

Let A be an $n \times n$ matrix with entries a_{ij}, b a vector of n numbers and x the vector of n unknowns in $Ax = b$. Equation i reads $\sum_{j=1}^{n} a_{ij}\, x_j = b_i$. If the diagonal entry a_{ii} is not zero, we can solve equation i for its "own" unknown x_i:

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij}\, x_j \right). \tag{5.3}$$

For the griddle, $a_{ii} = 4$ and the off-diagonal entries are -1 for neighbors, so Equation (5.3) is exactly the average rule Equation (5.1). All three methods apply Equation (5.3) to $i = 1,2, \ldots, n$ in turn; they differ only in which values of the other unknowns they plug in. Write $x^{(k)}$ for the guess after k sweeps (the superscript in brackets is a sweep counter, not a power), with $x^{(0)}$ the starting guess.

Jacobi uses only the previous sweep:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij}\, x_j^{(k)} \right), \qquad i = 1, \ldots, n. \tag{5.4}$$

Gauss-Seidel uses the new values for the unknowns already updated in this sweep ($j < i$) and the old values for the rest ($j > i$):

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}\, x_j^{(k+1)} - \sum_{j>i} a_{ij}\, x_j^{(k)} \right). \tag{5.5}$$

SOR mixes the old value with the Gauss-Seidel value through the factor ω:

$$x_i^{(k+1)} = (1-\omega)\, x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}\, x_j^{(k+1)} - \sum_{j>i} a_{ij}\, x_j^{(k)} \right). \tag{5.6}$$

Pseudocode 5.1 writes a whole SOR solver in pseudocode; with $\omega = 1$ it is Gauss-Seidel. Note line 5: the new value overwrites the old one in the same array, which is what makes the sum in line 4 use the new values for $j < i$ automatically.

Pseudocode 5.1 Successive over-relaxation; with omega = 1 it is the Gauss-Seidel method.

```
SOR(A, b, x, omega, tol, maxSweeps)
 1  for k = 1 to maxSweeps
 2      change = 0
 3      for i = 1 to n
 4          s = b[i] - (sum over j != i of A[i][j] * x[j])
 5          new = (1 - omega) * x[i] + omega * s / A[i][i]
 6          change = max(change, |new - x[i]|)
 7          x[i] = new
 8      if change < tol
 9          return x, k
10  return x, maxSweeps        // not converged: report it
```

5.3.2 Matrix splittings

The three methods look like three recipes, but they are one idea. Split A into its diagonal part D, its strictly lower part and its strictly upper part. It is customary to write the lower and upper parts with a minus sign, $A = D - L - U$, where $-L$ holds the entries below the diagonal and $-U$ those above it. For the griddle, $D = 4I$ and L and U contain ones where the neighbors are. Then $Ax = b$ becomes $Dx = (L + U)x + b$, and Equation (5.4) is this equation with the new x on the left and the old one on the right. Gauss-Seidel moves the lower part to the left as well, because those unknowns are already new, and SOR does the same after scaling by ω:

$$\begin{aligned} \text{Jacobi:} \quad & D\,x^{(k+1)} = (L+U)\,x^{(k)} + b \\ \text{Gauss-Seidel:} \quad & (D-L)\,x^{(k+1)} = U\,x^{(k)} + b \\ \text{SOR:} \quad & (D-\omega L)\,x^{(k+1)} = ((1-\omega)D + \omega U)\,x^{(k)} + \omega b \end{aligned} \tag{5.7}$$

Each line has the form $Mx^{(k+1)} = Nx^{(k)} + c$ with $A = M - N$ (for SOR, up to the factor ω). A good splitting makes M easy to "invert": D is diagonal, $D - L$ and $D - \omega L$ are lower triangular, so solving with them is just the forward substitution of Chapter 4, which is what the sweep does one row at a time. Multiplying by M^{-1} gives the standard form of a *stationary iteration*,

$$x^{(k+1)} = T\,x^{(k)} + c, \qquad T = M^{-1}N, \tag{5.8}$$

with the *iteration matrix* $T_J = D^{-1}(L + U)$ for Jacobi, $T_{GS} = (D - L)^{-1}U$ for Gauss-Seidel and $T_\omega = (D - \omega L)^{-1}((1 - \omega)D + \omega U)$ for SOR [2]. Nobody computes these matrices in a program. They exist to answer the questions of the next section.

5.4 When and How Fast They Converge

5.4.1 The error obeys the same iteration

Let x be the exact solution. It satisfies the iteration too, $x = Tx + c$, because that is just $Ax = b$ rearranged. Subtract it from Equation (5.8). The constant c cancels, and the error $e^{(k)} = x^{(k)} - x$ obeys

$$e^{(k+1)} = T\,e^{(k)}, \qquad \text{so} \qquad e^{(k)} = T^k e^{(0)}. \tag{5.9}$$

Everything is in this line. The starting guess only enters through $e^{(0)}$; after that the error is multiplied by the same matrix T in every sweep. The iteration converges from every starting guess exactly when the powers T^k shrink to the zero matrix.

To see when that happens, suppose for a moment that T has n independent eigenvectors $v_1, \dots, v_n$ with eigenvalues $\lambda_1, \dots, \lambda_n$ (so $Tv_i = \lambda_i v_i$; Chapter 12 is about computing them). Write the first error as a combination $e^{(0)} = c_1 v_1 + \cdots + c_n v_n$. Then $T^k e^{(0)} = c_1 \lambda_1^k v_1 + \cdots + c_n \lambda_n^k v_n$. Each term shrinks if $|\lambda_i| < 1$ and survives or grows otherwise. The largest $|\lambda_i|$ decides. It is called the **spectral radius**,

$$\rho(T) = \max_i |\lambda_i|. \tag{5.10}$$

DEFINITION. The iteration $x^{(k+1)} = Tx^{(k)} + c$ *converges* if $x^{(k)}$ approaches the solution for every starting guess $x^{(0)}$. **Theorem** [2]: it converges if and only if $\rho(T) < 1$. When it does, the error shrinks roughly by the factor $\rho(T)$ per sweep, and $-\log_{10}\rho(T)$, the number of correct decimal digits gained per sweep, is called the *rate of convergence*.

The theorem also holds for matrices without a full set of eigenvectors; the proof uses the Jordan form instead [2], [3]. One warning comes with that case: the error can grow for a while before the factor ρ takes over, which we will see for SOR in Section 5.6.

Check this on the four-point griddle. Its Jacobi matrix is $T_J = 1/4\,(L + U)$, whose eigenvalues are $1/2\,, 0, 0, -1/2$ (Exercise 2 finds them). So $\rho(T_J) = 0.5$, and the error should halve every sweep. The Jacobi table in Section 5.6 shows exactly that: the moves 7.66, 3.83, 1.91 halve, and so do the distances from 131.25 (7.66, 3.83, 1.91 as well). For Gauss-Seidel, $\rho(T_{GS}) = 0.25$, and the moves shrink by four.

5.4.2 A test you can do by eye: diagonal dominance

Eigenvalues are expensive to compute, so we want a test that can be read off the matrix. A matrix is **strictly diagonally dominant by rows** if in every row the diagonal entry outweighs all the others together:

$$|a_{ii}| > \sum_{j \neq i} |\,a_{ij}\,| \qquad \text{for every } i. \tag{5.11}$$

Theorem [2]: if A is strictly diagonally dominant, both Jacobi and Gauss-Seidel converge from every starting guess. The proof for Jacobi takes three lines. Row i of T_J has the entries $-a_{ij}/a_{ii}$ for $j \neq i$ and zero on the diagonal, so the largest row sum of absolute values of T_J is $q = \max_i \sum_{j \neq i} |\,a_{ij}|/|a_{ii}\,|$, and Equation (5.11) says $q < 1$. For any vector e, each component of $T_J e$ is at most q times the largest component of e in absolute value. So the largest error component shrinks at least by the factor q every sweep, and the error goes to zero.

Each row of the four-point system has $|4| > |-1| + |-1| = 2$, so it is strictly diagonally dominant and the theorem guarantees convergence ($q = 0.5$, which here equals ρ). An inner point of a bigger griddle has four neighbors and $|4| = 4$, which is not strict. The theorem still holds for such matrices when they are *irreducibly* diagonally dominant, meaning every point is connected to the edges and at least one row is strict; every row next to an edge is. The griddle matrix is also symmetric positive definite, and for such matrices another theorem (Ostrowski and Reich) guarantees that Gauss-Seidel and SOR with $0 < \omega < 2$ converge [2], [4]. Outside $0 < \omega < 2$ SOR never converges, because Kahan's theorem gives $\rho(T_\omega) \geq |\omega - 1|$ for every matrix [5].

WARNING: Diagonal dominance depends on the *order* in which the equations are written. The same four griddle equations written in the order 2, 1, 4, 3 put -1 on the diagonal of the first row and 4 beside it. Nothing about the solution has changed, but Jacobi on the reordered

system diverges: after 16 sweeps the values move by more than 10^{12}, as the output in Section 5.6 shows. Before iterating, put in each row's diagonal position the unknown with the largest coefficient.

5.4.3 The model problem: exact spectral radii

For the griddle with $n \times n$ inner points, the eigenvalues of T_J are known in closed form [2], [5]. The eigenvectors are sine waves across the plate, and the eigenvalues are

$$\mu_{p,q} = 1/2\left(\cos\frac{p\pi}{n+1} + \cos\frac{q\pi}{n+1}\right), \qquad p, q = 1, \ldots, n,$$

so the largest in absolute value is the smoothest wave, $p = q = 1$:

$$\rho_J = \cos\frac{\pi}{n+1}. \tag{5.12}$$

For $n = 2$ this is $\cos(\pi/3) = 0.5$, the four-point griddle. For $n = 20$ it is $\cos(\pi/21) = 0.98883$ and for $n = 40$ it is 0.99707. The finer the grid, the closer ρ_J creeps to 1 and the slower Jacobi becomes. The smoothest wave is the culprit: the average rule moves heat only one grid step per sweep, so a wave that spans the whole plate needs many sweeps to be corrected.

For matrices with the structure of the griddle ("consistently ordered", in the language of David Young's 1950 thesis [4]), Young proved three relations that make SOR predictable:

$$\rho_{GS} = \rho_J^2, \qquad \omega_{\text{opt}} = \frac{2}{1 + \sqrt{1 - \rho_J^2}}, \qquad \rho(T_{\omega_{\text{opt}}}) = \omega_{\text{opt}} - 1. \tag{5.13}$$

Gauss-Seidel is therefore exactly twice as fast as Jacobi on the griddle: squaring ρ doubles the digits gained per sweep. With Equation (5.12), $\sqrt{1 - \rho_J^2} = \sin(\pi/(n+1))$ and the best factor is $\omega_{\text{opt}} = 2/(1 + \sin(\pi/(n+1)))$. For $n = 2$ that is $2/(1 + 0.8660) = 1.0718$; for $n = 20$ it is 1.741 and for $n = 40$ it is 1.858.

5.4.4 How many sweeps?

If the error shrinks by ρ per sweep, reducing it from an initial size E_0 to a tolerance ε takes k sweeps with $\rho^k E_0 \approx \varepsilon$, that is,

$$k \approx \frac{\ln(\varepsilon/E_0)}{\ln\rho}. \tag{5.14}$$

For ρ close to 1, $\ln\rho \approx -(1 - \rho)$, so the number of sweeps grows like $1/(1 - \rho)$. With $\rho_J = \cos(\pi/(n+1)) \approx 1 - \pi^2/(2(n+1)^2)$, Jacobi needs a number of sweeps proportional to n^2: doubling the grid in each direction quadruples the sweeps. Gauss-Seidel halves that count but has the same n^2 growth. Optimal SOR has $1 - \rho = 2 - \omega_{\text{opt}} \approx 2\pi/(n+1)$, so its sweeps grow only like n. On a 40×40 griddle, that is the difference between thousands and a hundred and fifty, as **Table 5.1** will confirm.

5.4.5 When to stop, and what a small change really means

A program cannot compare its guess with the exact answer, because it does not know it. The natural stopping rule is "stop when no unknown moves by as much as a tolerance in one sweep". The danger is that a small change is not a small error. If the error shrinks by ρ per sweep, then after k sweeps the remaining error is the sum of all future changes, a geometric series:

$$\| x^{(k)} - x \| \leq \frac{\rho}{1-\rho} \| x^{(k)} - x^{(k-1)} \|. \tag{5.15}$$

For Jacobi on the 20×20 griddle, $\rho/(1-\rho) = 0.98883/0.01117 \approx 88.5$: when the last change was 10^{-6}, the error can still be about 8.9×10^{-5}, almost ninety times larger. Section 5.6 measures the true error and finds 8.76×10^{-5}. The bound Equation (5.15) is not pessimism; it is what actually happens. A method that converges slowly looks converged long before it is. So either divide your tolerance by $\rho/(1-\rho)$ when ρ is known, or stop on the *residual* $b - Ax^{(k)}$, which measures how badly the equations are violated and is honest for any method (Chapter 4 showed its relation to the error through the condition number).

5.4.6 Why bother: the cost of a sweep

One sweep touches each nonzero of A once. The griddle has at most five nonzeros per row, so a sweep over $N = n^2$ unknowns costs about $5N$ multiplications, and nothing but the unknowns needs to be stored: the matrix is never built, because Equation (5.1) *is* the matrix. Gaussian elimination on the full matrix needs about $\frac{2}{3}N^3$ operations and N^2 numbers of storage. For $n = 40$, $N = 1600$: elimination costs about 2.7×10^9 operations and 20 MB for the matrix; optimal SOR needs 155 sweeps of 8000 operations each, about 1.2×10^6, two thousand times less, and 13 KB for the unknowns. For a million unknowns the full matrix would need 8 terabytes; the SOR vector needs 8 megabytes. Elimination can be made cleverer by exploiting the band structure, but it always fills in zeros inside the band and its cost grows faster than the iterative one [5], [3]. This is why every large simulation of heat, groundwater, electric fields or structures uses iterative methods or their descendants.

5.5 Building It in Java and Python

Two programs make up this chapter. The first, `iterative_solvers`, runs Jacobi, Gauss-Seidel and SOR on the four equations Equation (5.2), prints the first sweeps so that you can check them against Section 5.2, tests diagonal dominance and shows the reordered system diverging. The second, `griddle`, solves the $n \times n$ griddle without ever building a matrix, measures the sweeps and the true error of each method, scans ω and prints a coarse temperature map.

Both programs are written twice, and the Java and Python versions print exactly the same characters. That takes two small precautions, which the listings show and which Chapter 1 explained. First, Python and Java format numbers with slightly different rounding rules in rare cases, so the Java programs format through `BigDecimal` with the same half-even rule that Python uses. Second, since Python 3.12 the built-in `sum()` adds floating-point numbers with a compensated (Neumaier) algorithm that is more accurate than a plain loop [6]; to keep the two languages in step, the Python programs add with a plain loop called `total`.

5.5.1 The small system

Listing 5.1 holds the three sweeps as functions. `jacobi_step` builds a new list from the old one, exactly Equation (5.4); `sor_step` copies the list once and then overwrites it element by element, so that later rows see the new values, which is Equation (5.6) (and Equation (5.5) when `omega` is 1). The driver `solve` is **Pseudocode 5.1** with a table printer and a guard that stops a diverging run.

Listing 5.1 `iterative_solvers.py`: Jacobi, Gauss-Seidel and SOR sweeps for a small dense system, with a diagonal-dominance test and a residual check.

```
"""Jacobi, Gauss-Seidel and SOR on the four-point griddle, step by step."""

# 4 T1 - T2 - T3 = 280 and so on: one equation per inner point
A = [[4.0, -1.0, -1.0, 0.0],
     [-1.0, 4.0, 0.0, -1.0],
     [-1.0, 0.0, 4.0, -1.0],
     [0.0, -1.0, -1.0, 4.0]]
B = [280.0, 140.0, 350.0, 210.0]



def total(values):
    """Plain left-to-right sum. Python 3.12+ sum() compensates rounding,
    which would make our digits differ from the Java version."""
    s = 0.0
    for v in values:
        s += v
    return s

def is_diagonally_dominant(a):
    """True when every |a_ii| exceeds the sum of the other |a_ij| in its row."""
    for i, row in enumerate(a):
        others = total(abs(v) for j, v in enumerate(row) if j != i)
        if abs(row[i]) <= others:
            return False
    return True


def jacobi_step(a, b, x):
    """One Jacobi sweep: every new value uses only the old vector x."""
    n = len(b)
    new = [0.0] * n
    for i in range(n):
        s = b[i]
        for j in range(n):
            if j != i:
                s -= a[i][j] * x[j]
        new[i] = s / a[i][i]
    return new
```

Listing 5.1 (Continued) *iterative_solvers.py: Jacobi, Gauss-Seidel and SOR sweeps for a small dense system, with a diagonal-dominance test and a residual check.*

```
def sor_step(a, b, x, omega):
    """One SOR sweep; omega = 1 is Gauss-Seidel. Fresh values used at once."""
    x = list(x)
    n = len(b)
    for i in range(n):
        s = b[i]
        for j in range(n):
            if j != i:
                s -= a[i][j] * x[j]
        x[i] = (1.0 - omega) * x[i] + omega * s / a[i][i]
    return x


def residual(a, b, x):
    """Largest |b_i - (Ax)_i|: how badly x fails the equations."""
    n = len(b)
    return max(abs(b[i] - total(a[i][j] * x[j] for j in range(n)))
               for i in range(n))


def solve(name, step, a, b, tol=1e-6, max_iter=200, show=0):
    x = [0.0] * len(b)
    print(name)
    if show:
        print("   k       x1        x2        x3        x4     change")
    for k in range(1, max_iter + 1):
        new = step(a, b, x)
        change = max(abs(p - q) for p, q in zip(new, x))
        x = new
        if k <= show:
            cells = "".join(f"{v:10.4f}" for v in x)
            print(f"{k:4d}{cells}{change:11.4f}")
        if change < tol:
            print(f"  converged after {k} iterations,"
                  f" residual {residual(a, b, x):.3e}")
            return x
        if change > 1e12:
            print(f"  diverged: change {change:.3e} at iteration {k}")
            return x
    print(f"  stopped after {max_iter} iterations")
    return x
```

Listing 5.1 (Continued) *iterative_solvers.py: Jacobi, Gauss-Seidel and SOR sweeps for a small dense system, with a diagonal-dominance test and a residual check.*

```
def main():
    print("diagonally dominant:", is_diagonally_dominant(A))
    x = solve("Jacobi", jacobi_step, A, B, show=6)
    print("  T =", " ".join(f"{v:.6f}" for v in x))
    x = solve("Gauss-Seidel", lambda a, b, v: sor_step(a, b, v, 1.0),
              A, B, show=6)
    x = solve("SOR, omega = 1.1", lambda a, b, v: sor_step(a, b, v, 1.1),
              A, B, show=3)
    x = solve("SOR, omega = 1.5", lambda a, b, v: sor_step(a, b, v, 1.5),
              A, B)
    # the same equations written in a different order
    order = [1, 0, 3, 2]
    a2 = [A[i] for i in order]
    b2 = [B[i] for i in order]
    print("reordered, diagonally dominant:", is_diagonally_dominant(a2))
    solve("Jacobi on the reordered system", jacobi_step, a2, b2)


if __name__ == "__main__":
    main()
```

Listing 5.2 is the same program in Java. The methods are passed around as a small interface `Step`, so one `solve` method runs all three. The helpers at the end format numbers the way Python does.

Listing 5.2 `IterativeSolvers.java`: the same three methods in Java.

```
import java.math.BigDecimal;
import java.math.MathContext;
import java.math.RoundingMode;

/** Jacobi, Gauss-Seidel and SOR on the four-point griddle, step by step. */
public class IterativeSolvers {
    // 4 T1 - T2 - T3 = 280 and so on: one equation per inner point
    static final double[][] A = {{4, -1, -1, 0}, {-1, 4, 0, -1},
                                 {-1, 0, 4, -1}, {0, -1, -1, 4}};
    static final double[] B = {280, 140, 350, 210};

    interface Step { double[] apply(double[][] a, double[] b, double[] x); }

    /** True when every |a_ii| exceeds the sum of the other |a_ij| in its row. */
    static boolean isDiagonallyDominant(double[][] a) {
        for (int i = 0; i < a.length; i++) {
            double others = 0;
            for (int j = 0; j < a.length; j++) {
                if (j != i) others += Math.abs(a[i][j]);
            }
            if (Math.abs(a[i][i]) <= others) return false;
        }
        return true;
    }

    /** One Jacobi sweep: every new value uses only the old vector x. */
    static double[] jacobiStep(double[][] a, double[] b, double[] x) {
        int n = b.length;
        double[] next = new double[n];
        for (int i = 0; i < n; i++) {
            double s = b[i];
            for (int j = 0; j < n; j++) {
                if (j != i) s -= a[i][j] * x[j];
            }
            next[i] = s / a[i][i];
        }
        return next;
    }
```

Listing 5.2 (Continued) *IterativeSolvers.java: the same three methods in Java.*

```
    /** One SOR sweep; omega = 1 is Gauss-Seidel. Fresh values used at once. */
    static double[] sorStep(double[][] a, double[] b, double[] x0, double omega) {
        double[] x = x0.clone();
        int n = b.length;
        for (int i = 0; i < n; i++) {
            double s = b[i];
            for (int j = 0; j < n; j++) {
                if (j != i) s -= a[i][j] * x[j];
            }
            x[i] = (1.0 - omega) * x[i] + omega * s / a[i][i];
        }
        return x;
    }

    /** Largest |b_i - (Ax)_i|: how badly x fails the equations. */
    static double residual(double[][] a, double[] b, double[] x) {
        double worst = 0;
        for (int i = 0; i < b.length; i++) {
            double s = 0;
            for (int j = 0; j < b.length; j++) s += a[i][j] * x[j];
            worst = Math.max(worst, Math.abs(b[i] - s));
        }
        return worst;
    }

    static double[] solve(String name, Step step, double[][] a, double[] b,
                          double tol, int maxIter, int show) {
        double[] x = new double[b.length];
        System.out.println(name);
        if (show > 0) {
            System.out.println("   k       x1        x2        x3        x4"
                               + "      change");
        }
        for (int k = 1; k <= maxIter; k++) {
            double[] next = step.apply(a, b, x);
            double change = 0;
            for (int i = 0; i < x.length; i++) {
                change = Math.max(change, Math.abs(next[i] - x[i]));
            }
            x = next;
            if (k <= show) {
                StringBuilder row = new StringBuilder(pad("" + k, 4));
                for (double v : x) row.append(f(v, 4, 10));
                System.out.println(row + f(change, 4, 11));
            }
            if (change < tol) {
```

Listing 5.2 (Continued) *IterativeSolvers.java: the same three methods in Java.*

```
                System.out.println("  converged after " + k + " iterations,"
                        + " residual " + e(residual(a, b, x), 3));
                return x;
            }
            if (change > 1e12) {
                System.out.println("  diverged: change " + e(change, 3)
                        + " at iteration " + k);
                return x;
            }
        }
        System.out.println("  stopped after " + maxIter + " iterations");
        return x;
    }

    public static void main(String[] args) {
        System.out.println("diagonally dominant: "
                + (isDiagonallyDominant(A) ? "True" : "False"));
        double[] x = solve("Jacobi", IterativeSolvers::jacobiStep, A, B,
                           1e-6, 200, 6);
        StringBuilder t = new StringBuilder("  T =");
        for (double v : x) t.append(" ").append(f(v, 6));
        System.out.println(t);
        solve("Gauss-Seidel", (a, b, v) -> sorStep(a, b, v, 1.0), A, B,
              1e-6, 200, 6);
        solve("SOR, omega = 1.1", (a, b, v) -> sorStep(a, b, v, 1.1), A, B,
              1e-6, 200, 3);
        solve("SOR, omega = 1.5", (a, b, v) -> sorStep(a, b, v, 1.5), A, B,
              1e-6, 200, 0);
        // the same equations written in a different order
        int[] order = {1, 0, 3, 2};
        double[][] a2 = new double[4][];
        double[] b2 = new double[4];
        for (int i = 0; i < 4; i++) {
            a2[i] = A[order[i]];
            b2[i] = B[order[i]];
        }
        System.out.println("reordered, diagonally dominant: "
                + (isDiagonallyDominant(a2) ? "True" : "False"));
        solve("Jacobi on the reordered system", IterativeSolvers::jacobiStep,
              a2, b2, 1e-6, 200, 0);
    }
```

Listing 5.2 (Continued) *IterativeSolvers.java: the same three methods in Java.*

```
    // ---- number formatting that matches Python's f"{x:w.df}" and ".de"
    static String f(double x, int d) {
        String s = new BigDecimal(x).setScale(d, RoundingMode.HALF_EVEN)
                                    .toPlainString();
        boolean neg = x < 0 || (x == 0 && 1 / x < 0);
        return neg && !s.startsWith("-") ? "-" + s : s;
    }

    static String f(double x, int d, int w) { return pad(f(x, d), w); }

    static String e(double x, int d) {
        if (x == 0) return "0." + "0".repeat(d) + "e+00";
        BigDecimal b = new BigDecimal(x).round(
                new MathContext(d + 1, RoundingMode.HALF_EVEN));
        int exp = b.precision() - b.scale() - 1;
        String digits = b.unscaledValue().abs().toString();
        digits = (digits + "0".repeat(d + 1)).substring(0, d + 1);
        String ex = (exp < 0 ? "-" : "+") + (Math.abs(exp) < 10 ? "0" : "")
                    + Math.abs(exp);
        return (x < 0 ? "-" : "") + digits.charAt(0) + "." + digits.substring(1)
               + "e" + ex;
    }

    static String pad(String s, int w) {
        return " ".repeat(Math.max(0, w - s.length())) + s;
    }
}
```

5.5.2 The griddle without a matrix

For the big griddle we do not store A at all. The grid is an $(n+2)\times(n+2)$ array whose border holds the edge temperatures and whose inside holds the unknowns; the neighbor of a point next to the edge is simply a border cell. A Jacobi sweep reads from a copy of the grid and writes the average into the real one; an SOR sweep works in place and over-relaxes. **Listing 5.3** does this in Python. The function `run` counts sweeps until no point moves by `tol` or more, and `optimal_omega` is Equation (5.13) with Equation (5.12). The main program solves the 20×20 griddle once very accurately (tolerance 10^{-12}) to have a reference, then measures every method against it.

Listing 5.3 `griddle.py`: the $n\times n$ griddle solved by Jacobi, Gauss-Seidel and SOR sweeps on a bordered grid, with sweep counts, true errors, an ω scan and a temperature map.

```
"""Steady temperature of a martabak griddle, found with Jacobi, Gauss-Seidel
and SOR on an n x n grid of inner points; the four edges are held fixed."""
import math

LEFT, RIGHT, TOP, BOTTOM = 200.0, 60.0, 80.0, 150.0   # edge temperatures, C


def new_grid(n):
    """(n+2) x (n+2) grid, inner points 0, the border holds the edges."""
    u = [[0.0] * (n + 2) for _ in range(n + 2)]
    for i in range(1, n + 1):
        u[i][0] = LEFT
        u[i][n + 1] = RIGHT
        u[0][i] = TOP
        u[n + 1][i] = BOTTOM
    return u


def jacobi_sweep(u, n):
    """Every point becomes the mean of its four OLD neighbors."""
    old = [row[:] for row in u]
    change = 0.0
    for i in range(1, n + 1):
        for j in range(1, n + 1):
            v = 0.25 * (old[i - 1][j] + old[i + 1][j]
                        + old[i][j - 1] + old[i][j + 1])
            change = max(change, abs(v - old[i][j]))
            u[i][j] = v
    return change


def sor_sweep(u, n, omega):
    """Gauss-Seidel value from the newest neighbors, then over-relaxed."""
    change = 0.0
    for i in range(1, n + 1):
        for j in range(1, n + 1):
            gs = 0.25 * (u[i - 1][j] + u[i + 1][j]
                         + u[i][j - 1] + u[i][j + 1])
            v = u[i][j] + omega * (gs - u[i][j])
            change = max(change, abs(v - u[i][j]))
            u[i][j] = v
    return change
```

Listing 5.3 (Continued) *griddle.py: the n \times n griddle solved by Jacobi, Gauss-Seidel and SOR sweeps on a bordered grid, with sweep counts, true errors, an \omega scan and a temperature map.*

```
def run(n, method, omega=1.0, tol=1e-6, max_iter=100000):
    """Sweep until no point moves by tol or more; return grid and sweeps."""
    u = new_grid(n)
    for k in range(1, max_iter + 1):
        if method == "jacobi":
            change = jacobi_sweep(u, n)
        else:
            change = sor_sweep(u, n, omega)
        if change < tol:
            return u, k
    return u, max_iter


def optimal_omega(n):
    return 2.0 / (1.0 + math.sin(math.pi / (n + 1)))


def max_diff(u, v, n):
    return max(abs(u[i][j] - v[i][j])
               for i in range(1, n + 1) for j in range(1, n + 1))


def main():
    n = 20
    w = optimal_omega(n)
    ref, _ = run(n, "sor", w, tol=1e-12)
    print(f"Griddle with {n} x {n} inner points ({n * n} unknowns)")
    print("stop when no point moves by 1e-06 or more in a sweep")
    print("method           omega  sweeps  max error")
    for name, method, om in [("Jacobi", "jacobi", 1.0),
                             ("Gauss-Seidel", "sor", 1.0),
                             ("SOR", "sor", 1.5),
                             ("SOR (optimal)", "sor", w)]:
        u, k = run(n, method, om)
        print(f"{name:15s}{om:7.3f}{k:8d}{max_diff(u, ref, n):11.2e}")

    print()
    print("   n  unknowns  rho_Jacobi  Jacobi  Gauss-Seidel  SOR  omega_opt")
    for m in (10, 20, 40):
        kj = run(m, "jacobi")[1]
        kg = run(m, "sor", 1.0)[1]
        ks = run(m, "sor", optimal_omega(m))[1]
        rho = math.cos(math.pi / (m + 1))
        print(f"{m:4d}{m * m:10d}{rho:12.5f}{kj:8d}{kg:14d}{ks:5d}"
              f"{optimal_omega(m):11.3f}")
```

Listing 5.3 (Continued) *griddle.py: the n \times n griddle solved by Jacobi, Gauss-Seidel and SOR sweeps on a bordered grid, with sweep counts, true errors, an \omega scan and a temperature map.*

```
    print()
    print("SOR sweeps for n = 20 and different omega")
    for step in range(0, 20):
        om = 1.0 + 0.05 * step
        print(f"  omega {om:.2f}: {run(n, 'sor', om)[1]:5d}")

    print()
    print("Temperature (C) at sample points, n = 20"
          " (row 1 lies next to the top edge)")
    cols = (1, 5, 10, 15, 20)
    print("row " + "".join(f"{'col ' + str(c):>8s}" for c in cols))
    for r in (1, 5, 10, 15, 20):
        print(f"{r:3d} " + "".join(f"{ref[r][c]:8.1f}" for c in cols))
    good = sum(1 for i in range(1, n + 1) for j in range(1, n + 1)
               if 140.0 <= ref[i][j] <= 170.0)
    print(f"points between 140 and 170 C: {good} of {n * n}")


if __name__ == "__main__":
    main()
```

Listing 5.4 is the Java twin. The grid is a `double[n + 2][n + 2]`, and `run` fills the grid it is given, so the reference solution and the measured ones are computed by the same code.

Listing 5.4 `Griddle.java`: the griddle solver in Java.

```
import java.math.BigDecimal;
import java.math.MathContext;
import java.math.RoundingMode;

/** Steady temperature of a martabak griddle, found with Jacobi, Gauss-Seidel
 *  and SOR on an n x n grid of inner points; the four edges are held fixed. */
public class Griddle {
    static final double LEFT = 200, RIGHT = 60, TOP = 80, BOTTOM = 150;

    /** (n+2) x (n+2) grid, inner points 0, the border holds the edges. */
    static double[][] newGrid(int n) {
        double[][] u = new double[n + 2][n + 2];
        for (int i = 1; i <= n; i++) {
            u[i][0] = LEFT;
            u[i][n + 1] = RIGHT;
            u[0][i] = TOP;
            u[n + 1][i] = BOTTOM;
        }
        return u;
    }

    /** Every point becomes the mean of its four OLD neighbors. */
    static double jacobiSweep(double[][] u, int n) {
        double[][] old = new double[n + 2][];
        for (int i = 0; i < n + 2; i++) old[i] = u[i].clone();
        double change = 0;
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                double v = 0.25 * (old[i - 1][j] + old[i + 1][j]
                                   + old[i][j - 1] + old[i][j + 1]);
                change = Math.max(change, Math.abs(v - old[i][j]));
                u[i][j] = v;
            }
        }
        return change;
    }
```

Listing 5.4 (Continued) *Griddle.java: the griddle solver in Java.*

```
    /** Gauss-Seidel value from the newest neighbors, then over-relaxed. */
    static double sorSweep(double[][] u, int n, double omega) {
        double change = 0;
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                double gs = 0.25 * (u[i - 1][j] + u[i + 1][j]
                                  + u[i][j - 1] + u[i][j + 1]);
                double v = u[i][j] + omega * (gs - u[i][j]);
                change = Math.max(change, Math.abs(v - u[i][j]));
                u[i][j] = v;
            }
        }
        return change;
    }

    /** Sweep until no point moves by tol or more; the grid is filled in u. */
    static int run(double[][] u, int n, boolean jacobi, double omega,
                   double tol) {
        for (int k = 1; k <= 100000; k++) {
            double change = jacobi ? jacobiSweep(u, n) : sorSweep(u, n, omega);
            if (change < tol) return k;
        }
        return 100000;
    }

    static int sweeps(int n, boolean jacobi, double omega) {
        return run(newGrid(n), n, jacobi, omega, 1e-6);
    }

    static double optimalOmega(int n) {
        return 2.0 / (1.0 + Math.sin(Math.PI / (n + 1)));
    }

    static double maxDiff(double[][] u, double[][] v, int n) {
        double worst = 0;
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                worst = Math.max(worst, Math.abs(u[i][j] - v[i][j]));
            }
        }
        return worst;
    }
```

Listing 5.4 (Continued) *Griddle.java: the griddle solver in Java.*

```
    public static void main(String[] args) {
        int n = 20;
        double w = optimalOmega(n);
        double[][] ref = newGrid(n);
        run(ref, n, false, w, 1e-12);
        System.out.println("Griddle with " + n + " x " + n + " inner points ("
                           + n * n + " unknowns)");
        System.out.println("stop when no point moves by 1e-06 or more in a sweep");
        System.out.println("method           omega  sweeps  max error");
        String[] names = {"Jacobi", "Gauss-Seidel", "SOR", "SOR (optimal)"};
        double[] omegas = {1.0, 1.0, 1.5, w};
        for (int m = 0; m < 4; m++) {
            double[][] u = newGrid(n);
            int k = run(u, n, m == 0, omegas[m], 1e-6);
            String name = (names[m] + " ".repeat(15)).substring(0, 15);
            System.out.println(name + f(omegas[m], 3, 7) + pad("" + k, 8)
                               + pad(e(maxDiff(u, ref, n), 2), 11));
        }

        System.out.println();
        System.out.println("   n  unknowns  rho_Jacobi  Jacobi  Gauss-Seidel"
                           + "  SOR  omega_opt");
        for (int m : new int[] {10, 20, 40}) {
            double rho = Math.cos(Math.PI / (m + 1));
            System.out.println(pad("" + m, 4) + pad("" + m * m, 10)
                    + f(rho, 5, 12) + pad("" + sweeps(m, true, 1.0), 8)
                    + pad("" + sweeps(m, false, 1.0), 14)
                    + pad("" + sweeps(m, false, optimalOmega(m)), 5)
                    + f(optimalOmega(m), 3, 11));
        }

        System.out.println();
        System.out.println("SOR sweeps for n = 20 and different omega");
        for (int step = 0; step < 20; step++) {
            double om = 1.0 + 0.05 * step;
            System.out.println("  omega " + f(om, 2) + ": "
                               + pad("" + sweeps(n, false, om), 5));
        }
```

Listing 5.4 (Continued) *Griddle.java: the griddle solver in Java.*

```
        System.out.println();
        System.out.println("Temperature (C) at sample points, n = 20"
                           + " (row 1 lies next to the top edge)");
        int[] cols = {1, 5, 10, 15, 20};
        StringBuilder head = new StringBuilder("row ");
        for (int c : cols) head.append(pad("col " + c, 8));
        System.out.println(head);
        for (int r : cols) {
            StringBuilder line = new StringBuilder(pad("" + r, 3) + " ");
            for (int c : cols) line.append(f(ref[r][c], 1, 8));
            System.out.println(line);
        }
        int good = 0;
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (ref[i][j] >= 140.0 && ref[i][j] <= 170.0) good++;
            }
        }
        System.out.println("points between 140 and 170 C: " + good + " of "
                           + n * n);
    }

    // ---- number formatting that matches Python's f"{x:w.df}" and ".de"
    static String f(double x, int d) {
        String s = new BigDecimal(x).setScale(d, RoundingMode.HALF_EVEN)
                                    .toPlainString();
        boolean neg = x < 0 || (x == 0 && 1 / x < 0);
        return neg && !s.startsWith("-") ? "-" + s : s;
    }

    static String f(double x, int d, int w) { return pad(f(x, d), w); }

    static String e(double x, int d) {
        if (x == 0) return "0." + "0".repeat(d) + "e+00";
        BigDecimal b = new BigDecimal(x).round(
                new MathContext(d + 1, RoundingMode.HALF_EVEN));
        int exp = b.precision() - b.scale() - 1;
        String digits = b.unscaledValue().abs().toString();
        digits = (digits + "0".repeat(d + 1)).substring(0, d + 1);
        String ex = (exp < 0 ? "-" : "+") + (Math.abs(exp) < 10 ? "0" : "")
                    + Math.abs(exp);
        return (x < 0 ? "-" : "") + digits.charAt(0) + "." + digits.substring(1)
               + "e" + ex;
    }
```

Listing 5.4 (Continued) *Griddle.java: the griddle solver in Java.*

```
    static String pad(String s, int w) {
        return " ".repeat(Math.max(0, w - s.length())) + s;
    }
}
```

5.6 Running It

Compile the Java programs once in the `java` folder with `javac -d out *.java` and run them with `java -cp out IterativeSolvers` and `java -cp out Griddle`; run the Python versions with `python3 iterative_solvers.py` and `python3 griddle.py` in the `python` folder. The two languages print exactly the same text. The small system first:

```
diagonally dominant: True
Jacobi
   k       x1        x2        x3        x4     change
   1   70.0000   35.0000   87.5000   52.5000    87.5000
   2  100.6250   65.6250  118.1250   83.1250    30.6250
   3  115.9375   80.9375  133.4375   98.4375    15.3125
   4  123.5938   88.5938  141.0938  106.0938     7.6562
   5  127.4219   92.4219  144.9219  109.9219     3.8281
   6  129.3359   94.3359  146.8359  111.8359     1.9141
  converged after 27 iterations, residual 1.825e-06
  T = 131.249999 96.249999 148.749999 113.749999
Gauss-Seidel
   k       x1        x2        x3        x4     change
   1   70.0000   52.5000  105.0000   91.8750   105.0000
   2  109.3750   85.3125  137.8125  108.2812    39.3750
   3  125.7812   93.5156  146.0156  112.3828    16.4062
   4  129.8828   95.5664  148.0664  113.4082     4.1016
   5  130.9082   96.0791  148.5791  113.6646     1.0254
   6  131.1646   96.2073  148.7073  113.7286     0.2563
  converged after 15 iterations, residual 9.779e-07
SOR, omega = 1.1
   k       x1        x2        x3        x4     change
   1   77.0000   59.6750  117.4250  106.4525   117.4250
   2  118.0025   94.2576  146.2326  113.2396    41.0025
   3  131.3346   96.3321  148.8846  113.8606    13.3321
  converged after 10 iterations, residual 1.122e-07
SOR, omega = 1.5
  converged after 29 iterations, residual 8.545e-07
reordered, diagonally dominant: False
Jacobi on the reordered system
  diverged: change 1.073e+12 at iteration 16
```

Read the Jacobi table against Section 5.2.3: the first two rows are the ones we computed by hand, and from the third row on the “change” column halves each time, as $\rho(T_J) = 0.5$ predicts. The last reported values, 131.249999 and so on, are within 10^{-6} of the exact answer. Gauss-Seidel reaches the same accuracy in 15 sweeps with changes that shrink by four, SOR with $\omega = 1.1$ (close to the optimum 1.072) in 10, and SOR with $\omega = 1.5$ overshoots and needs 29. The residual printed at the end is tiny for all of them. The reordered system is not diagonally dominant, and Jacobi runs away, just as the warning in Section 5.4.2 said.

Now the real griddle:

```
Griddle with 20 x 20 inner points (400 unknowns)
stop when no point moves by 1e-06 or more in a sweep
method           omega  sweeps  max error
Jacobi           1.000    1302   8.76e-05
Gauss-Seidel     1.000     683   4.31e-05
SOR              1.500     235   1.36e-05
SOR (optimal)    1.741      80   8.85e-07

   n  unknowns  rho_Jacobi  Jacobi  Gauss-Seidel  SOR  omega_opt
  10       100     0.95949     385           202   42      1.560
  20       400     0.98883    1302           683   80      1.741
  40      1600     0.99707    4518          2377  155      1.858

SOR sweeps for n = 20 and different omega
  omega 1.00:   683
  omega 1.05:   621
  omega 1.10:   565
  omega 1.15:   513
  omega 1.20:   465
  omega 1.25:   420
  omega 1.30:   378
  omega 1.35:   340
  omega 1.40:   303
  omega 1.45:   268
  omega 1.50:   235
  omega 1.55:   204
  omega 1.60:   173
  omega 1.65:   143
  omega 1.70:   111
  omega 1.75:    84
  omega 1.80:    92
  omega 1.85:   128
  omega 1.90:   189
  omega 1.95:   380

Temperature (C) at sample points, n = 20 (row 1 lies next to the top edge)
row    col 1   col 5  col 10  col 15  col 20
  1    139.8    94.4    85.6    81.1    70.5
  5    183.9   135.7   106.0    86.8    65.1
 10    190.7   156.4   123.9    96.6    66.4
 15    191.5   162.3   136.4   111.2    70.5
 20    174.5   153.7   147.7   141.6   105.2
points between 140 and 170 C: 94 of 400
```

The first table deserves a slow reading. All four runs stopped by the same rule, a change below 10^{-6}, yet their true errors differ by a factor of a hundred. Jacobi's error 8.76×10^{-5} matches the bound Equation (5.15), 88.5×10^{-6}, almost to the digit; Gauss-Seidel's 4.31×10^{-5} matches $\rho_{GS}/(1-\rho_{GS}) \times 10^{-6} = 44.0 \times 10^{-6}$. Only optimal SOR, whose ρ is far from 1, stops with an error below the tolerance. **Figure 5.2** shows the whole history: on a logarithmic scale the errors of Jacobi and Gauss-Seidel fall along straight lines, whose slopes are $\log_{10}\rho_J$ and $\log_{10}\rho_{GS}$, one twice as steep as the other; SOR drops much faster after a short hesitation at the start, the transient growth that Section 5.4.1 warned about.

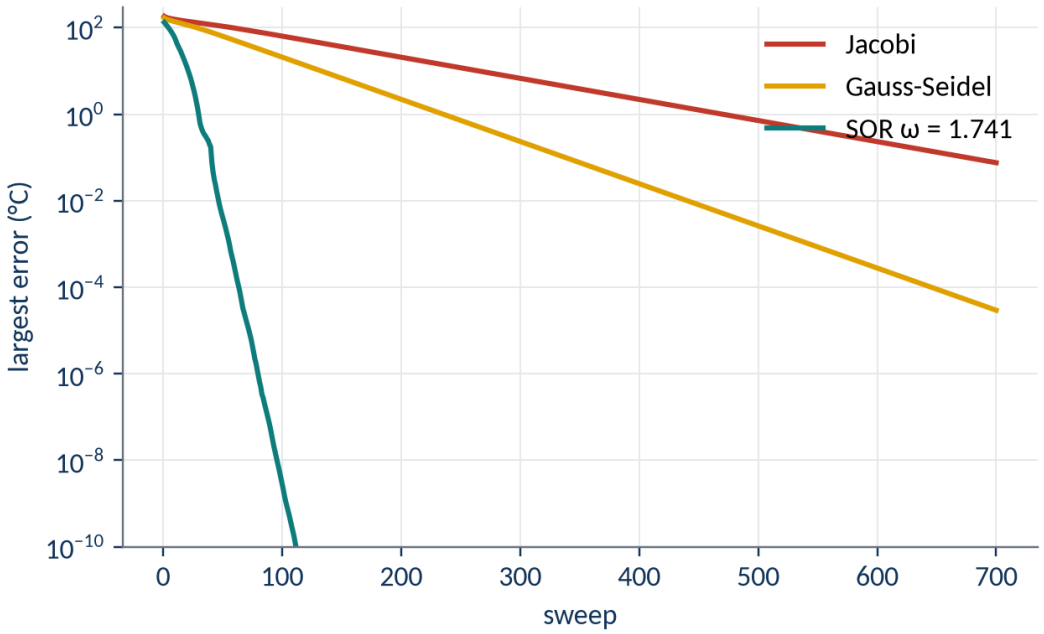


Figure 5.2 Largest error of the 20 × 20 griddle after each sweep, against a reference solution. Straight lines on this scale mean a constant factor per sweep: $\rho_J = 0.989$, $\rho_{GS} = 0.978$, and about 0.74 for SOR with $\omega = 1.741$.

The second table of the output is summarized, together with the predictions of Section 5.4, in **Table 5.1**. The prediction Equation (5.14) for Jacobi with $n = 20$, starting from an error of about 200 °C and stopping when the change is 10^{-6} (so when $(1-\rho)\,E_0\rho^k \approx 10^{-6}$), gives 1301.6 sweeps; the program needed 1302. Doubling n from 20 to 40 multiplies the Jacobi and Gauss-Seidel sweeps by about 3.5 (close to the factor 4 of the n^2 law) and the SOR sweeps by about 1.9 (close to the factor 2 of the n law).

Table 5.1 Sweeps needed to bring the largest change below 10^{-6} on the $n \times n$ griddle. Gauss-Seidel needs about half of Jacobi's sweeps, as $\rho_{GS} = \rho_J^2$ predicts; optimal SOR grows only linearly with n.

n	Unknowns	ρ_J	Jacobi	Gauss-Seidel	SOR, ω_{opt}	ω_{opt}
10	100	0.95949	385	202	42	1.560
20	400	0.98883	1302	683	80	1.741
40	1600	0.99707	4518	2377	155	1.858

The ω scan of the output is drawn in **Figure 5.3**. The curve falls slowly from 683 sweeps at $\omega = 1$, reaches its bottom of 84 sweeps at $\omega = 1.75$ (80 at the exact optimum 1.741) and then climbs steeply: 380 sweeps at $\omega = 1.95$, and divergence at $\omega = 2$. The asymmetry is a practical lesson. If you do not know ω_{opt} exactly, err on the high side, where the curve is steep but still low near the bottom, rather than far on the low side; better still, estimate ρ_J and use Equation (5.13).

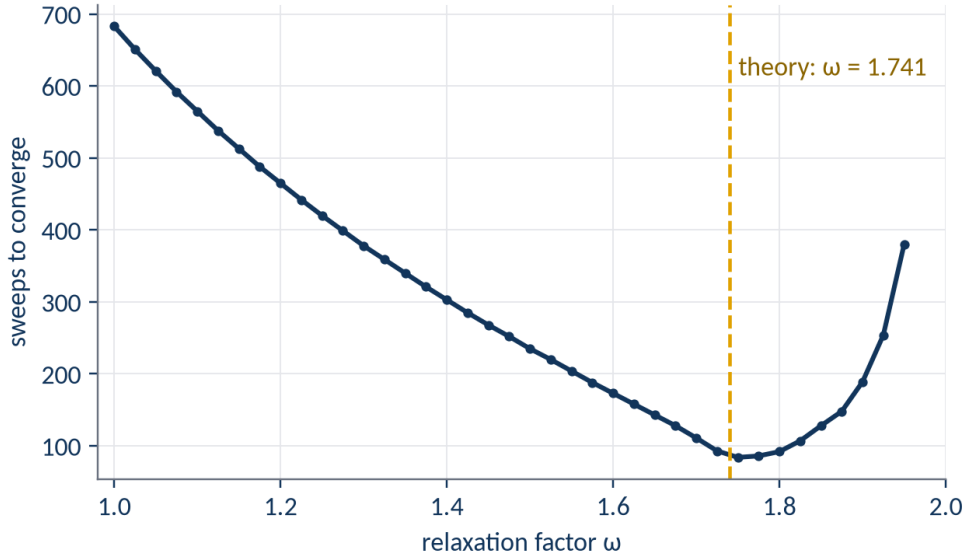


Figure 5.3 Sweeps needed by SOR on the 20 × 20 griddle for relaxation factors from 1 to 1.95. The minimum sits at Young's value $\omega_{\text{opt}} = 1.741$.

Finally, the map. The sample table in the output shows 191.5 °C close to the burner and 65 to 70 °C along the right edge. **Figure 5.4** draws the whole plate on a finer 40 × 40 grid, one point per centimeter, with the 140 °C and 170 °C lines. The martabak zone is a band eight to ten centimeters wide that starts a few centimeters from the burner and widens toward the warm front edge, bending along it to the right. On the 20 × 20 grid, 94 of the 400 points, almost a quarter of the plate, lie in the zone. Pak Joni's nephew now has a picture: martabak telur in the band, roti bakar just to its right, and the back right corner for keeping things warm.

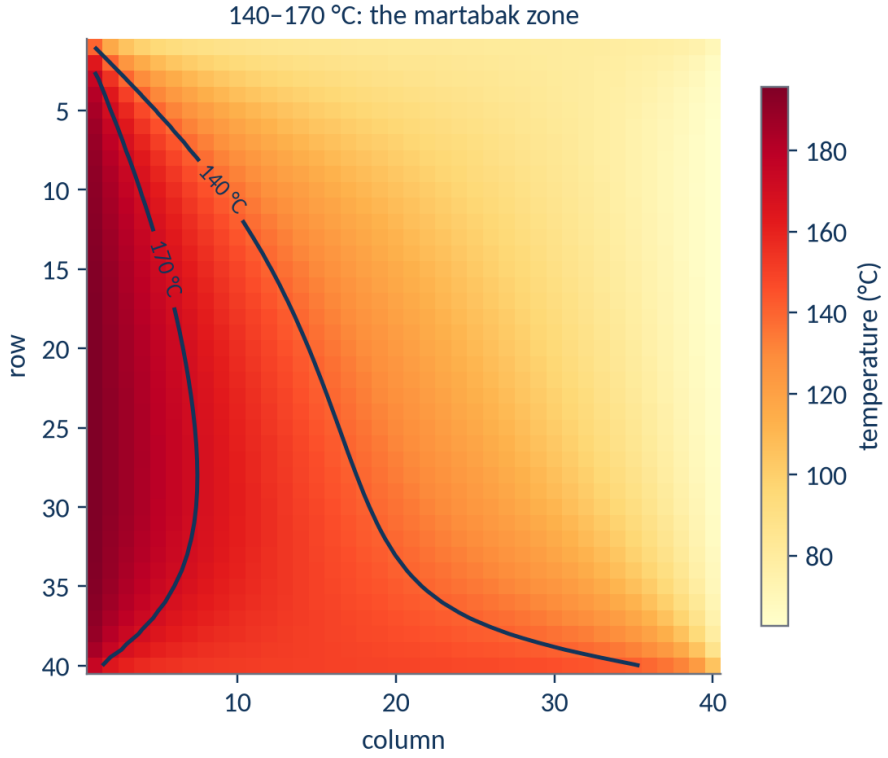


Figure 5.4 Steady temperature of the griddle on a 40 × 40 grid (burner on the left, warm front edge at the bottom), with the 140 °C and 170 °C lines that bound the martabak zone.

5.7 Checking Against the Library

Our programs are small enough to trust by reading, but a second opinion from a well-tested library is cheap. In Python the standard tools for sparse systems live in `scipy.sparse`; at the time of writing (September 2026) the current releases are NumPy 2.5.3 and SciPy 1.18.1 [7], [8]. **Listing 5.5** builds the griddle matrix explicitly with `diags` and a Kronecker product (a compact way of saying "the same one-dimensional pattern in both directions"), stores it in the compressed sparse row format that keeps only the nonzeros, and solves it twice: with the sparse direct solver `spsolve`, which is Gaussian elimination that works hard to avoid fill-in, and with the conjugate gradient method `cg` [9], an iterative method for symmetric positive definite matrices that is a smarter cousin of the methods in this chapter.

Listing 5.5 `compare_scipy.py`: the griddle checked against SciPy's sparse direct solver and its conjugate gradient method.

```
"""Check the griddle against SciPy's sparse direct solver and its CG."""
import numpy as np
import scipy
from scipy.sparse import diags, identity, kron
from scipy.sparse.linalg import cg, spsolve

import griddle
from iterative_solvers import A, B


def griddle_system(n):
    """Matrix and right side of the n*n griddle equations, row by row."""
    t = diags([-1.0, 4.0, -1.0], [-1, 0, 1], shape=(n, n))
    s = diags([-1.0, -1.0], [-1, 1], shape=(n, n))
    a = (kron(identity(n), t) + kron(s, identity(n))).tocsr()
    b = np.zeros(n * n)
    for i in range(n):
        for j in range(n):
            k = i * n + j
            if i == 0:
                b[k] += griddle.TOP
            if i == n - 1:
                b[k] += griddle.BOTTOM
            if j == 0:
                b[k] += griddle.LEFT
            if j == n - 1:
                b[k] += griddle.RIGHT
    return a, b


def main():
    print(f"NumPy {np.__version__}, SciPy {scipy.__version__}")
    print("4 x 4 system, numpy.linalg.solve:",
          np.round(np.linalg.solve(np.array(A), np.array(B)), 6))
    n = 20
    a, b = griddle_system(n)
    print(f"griddle matrix: {a.shape[0]} x {a.shape[1]},"
          f" {a.nnz} nonzeros of {a.shape[0] * a.shape[1]}")
    x = spsolve(a, b)
    u, k = griddle.run(n, "sor", griddle.optimal_omega(n), tol=1e-12)
    ours = np.array([u[i][j] for i in range(1, n + 1)
                     for j in range(1, n + 1)])
    print(f"SOR ({k} sweeps) vs spsolve: max difference"
          f" {np.max(np.abs(ours - x)):.1e}")
    count = []
    xc, info = cg(a, b, rtol=1e-10, callback=lambda xk: count.append(1))
```

Listing 5.5 (Continued) *compare_scipy.py: the griddle checked against SciPy's sparse direct solver and its conjugate gradient method.*

```
    print(f"conjugate gradient: {len(count)} iterations,"
          f" max difference {np.max(np.abs(xc - x)):.1e}")


if __name__ == "__main__":
    main()
```

```
NumPy 2.5.3, SciPy 1.18.1
4 x 4 system, numpy.linalg.solve: [131.25  96.25 148.75 113.75]
griddle matrix: 400 x 400, 1920 nonzeros of 160000
SOR (126 sweeps) vs spsolve: max difference 1.4e-12
conjugate gradient: 64 iterations, max difference 6.9e-09
```

The small system agrees with `numpy.linalg.solve` to the last digit printed. The griddle matrix has 1,920 nonzeros out of 160,000 entries, 1.2%; for a 1000×1000 grid the fraction would be 0.0005%, which is why storing only nonzeros matters. Our SOR solution, iterated to 10^{-12}, agrees with the direct solver to 1.4×10^{-12}. Conjugate gradient needs 64 iterations for a relative residual of 10^{-10}; each of its iterations costs about as much as an SOR sweep, and it needs no ω at all. In practice this is what you would call for a large symmetric problem, usually with a *preconditioner* that is often one SOR or incomplete-factorization sweep [5], [10]. The methods of this chapter are therefore not museum pieces: they are the building blocks inside the modern solvers.

5.8 Making It Better

Four improvements are worth knowing, even though the programs above do not need them.

- **Start from a good guess.** The number of sweeps depends on the size of the first error E_0 through Equation (5.14). When Pak Joni turns the burner up from 200 to 210 °C, the old map is an excellent starting guess for the new one; Exercise 5 shows that it saves about 30% of the sweeps even on this small grid. A system that is solved again and again with slowly changing data (weather models, a weekly re-computation) should always start from the previous answer.
- **Red-black ordering.** Color the grid like a chessboard. Every red point has only black neighbors and vice versa, so all red points can be updated at once from the black ones, then all black points from the red ones. This is Gauss-Seidel with a different order, it converges at the same rate on the griddle, and each half-sweep is perfectly parallel. Chapter 15 returns to it.
- **Multigrid.** The slow part of the error is the smooth wave of Equation (5.12). On a coarser grid that wave is no longer smooth, and a few sweeps kill it. Multigrid methods sweep on a hierarchy of grids and solve the griddle in a number of operations proportional to the number of unknowns, the best one can hope for [5].
- **Krylov methods.** Conjugate gradient, seen in Section 5.7, and its relatives for non-symmetric systems (GMRES, BiCGSTAB) choose each step using all previous ones. With a good preconditioner they are the default for large sparse systems in libraries such as SciPy [10], [9].

5.9 Using It for Real

To make a map of your own griddle, oven wall, drying floor or any flat piece that is heated or cooled along its edges, four things need to change in `griddle.py`, and nothing else.

1. **Measure the edges.** An infrared thermometer costs little. Measure each edge at three or four points once the plate has stopped warming. If an edge is not uniform, set the border cells to the

measured values, interpolated between measurement points (Chapter 7 shows how), instead of one constant per edge.

2. **Choose the grid.** One point per centimeter is plenty for cooking; the 40×40 grid of **Figure 5.4** takes a fraction of a second with SOR.
3. **Handle an insulated edge.** If one edge touches nothing and loses no heat, it has no fixed temperature. Instead, no heat crosses it, which means the temperature just inside equals the temperature just outside. Implement it by copying the first inner row into the border row before every sweep; the average rule then does the rest.
4. **Check the model.** Measure three or four inner points with the thermometer and compare. If the measured middle of the plate is clearly cooler than the map, heat escaping from the top surface matters, and the average rule needs a loss term: $T = (T_{\text{left}} + T_{\text{right}} + T_{\text{up}} + T_{\text{down}} + \beta T_{\text{air}})/(4 + \beta)$ for a small constant β fitted to your measurements (Chapter 8 shows how to fit it). The matrix stays diagonally dominant, even strictly, so everything in this chapter still applies.

A word of caution for any appliance: a computed map shows where the plate is hotter or cooler, not whether the appliance is safe. A gas burner that heats unevenly, smells of gas or has a yellow flame needs a technician, not a model.

5.10 Numerical Methods in Practice: Profitina

Profitina's AI Visibility Engine looks at a website as a whole, page by page, and scores how ready it is to be cited by AI answer engines. The pages of a site link to one another, and several quantities one may want to know about a site are defined in the same way as the griddle temperatures: the value of a page depends on the values of the pages around it. Written out, such a definition is a large sparse linear system, one unknown per page and only as many nonzeros per row as the page has links. That is precisely the shape this chapter is about, and the class of method used for it is the sweep: start from a guess, update every page from its neighbors, repeat. Chapter 12 meets the most famous example, PageRank, which is a Jacobi-type iteration on the link graph of the web.

Two lessons of this chapter carry straight over to a product that measures things repeatedly. The first is the warm start. Profitina's scheduler re-scans every brand automatically at a cadence set by its tier and keeps a weekly score trend in the historical analytics dashboard. From one scan to the next only a few pages change, so the previous week's values are an excellent starting guess, just as the old griddle map is for a slightly hotter burner. The second lesson is honesty about convergence. Equation (5.15) showed that "the numbers have stopped moving" can hide an error ninety times larger than the last change. A dashboard that shows a score to a brand owner, who is not a numerical analyst, must stop its computations on a criterion that bounds the real error, or report the uncertainty. The app advises and the owner decides, and advice is only as good as the numbers under it.

5.11 Common Mistakes, and How to Avoid Them

- **Iterating on a badly ordered system.** Jacobi and Gauss-Seidel need large diagonal entries. Before iterating, check Equation (5.11) or at least put the largest coefficient of each equation on the diagonal; the reordered griddle of Section 5.6 diverges although it has the same solution.

- **Taking a small change for a small error.** When ρ is close to 1, the error after stopping is about $\rho/(1-\rho)$ times the last change, 88 times for Jacobi on the 20×20 griddle. Tighten the tolerance, or stop on the residual.
- **Mixing Jacobi and Gauss-Seidel by accident.** Writing the Jacobi update into the same array you read from turns it into a Gauss-Seidel sweep with an order you did not choose (in Python, `new = x` instead of `new = x[:]` does exactly that). Jacobi needs a separate copy of the old values.
- **Choosing ω outside (0,2).** SOR diverges for every matrix when $\omega \leq 0$ or $\omega \geq 2$. Values a little above the optimum are cheap; values far below it waste most of the benefit.
- **Using the optimal ω of one grid for another.** ω_{opt} depends on n: 1.560 for $n = 10$, 1.858 for $n = 40$. Recompute it with Equation (5.13) when the grid changes.
- **Forgetting the maximum sweep count.** A loop `while change >= tol` runs forever on a diverging or very slowly converging problem. Always bound the sweeps and report when the bound was reached, as line 10 of **Pseudocode 5.1** does.
- **Building the matrix when you do not need it.** For a grid, the average rule is the matrix. Storing N^2 numbers for $N = 10^6$ is impossible; storing the N unknowns is trivial.

5.12 Chapter Summary

- Many large linear systems have only a few unknowns per equation. The steady temperature of a plate is one: each inner point is the average of its four neighbors (Equation (5.1)), which gives the four-point system Equation (5.2) for the coarsest grid and a system with n^2 unknowns for an $n \times n$ grid.
- Iterative methods improve a complete guess sweep by sweep. Jacobi (Equation (5.4)) updates every unknown from the previous sweep, Gauss-Seidel (Equation (5.5)) uses new values at once, and SOR (Equation (5.6)) over-relaxes the Gauss-Seidel step by a factor ω (**Pseudocode 5.1**).
- All three are splittings $A = M - N$ (Equation (5.7)) and stationary iterations $x^{(k+1)} = Tx^{(k)} + c$ (Equation (5.8)). The error is multiplied by T every sweep (Equation (5.9)), so the method converges from every start exactly when the spectral radius (Equation (5.10)) is below 1.
- Strict diagonal dominance (Equation (5.11)) guarantees convergence of Jacobi and Gauss-Seidel and can be checked by eye; symmetric positive definite matrices guarantee Gauss-Seidel and SOR with $0 < \omega < 2$.
- On the griddle, $\rho_J = \cos(\pi/(n+1))$ (Equation (5.12)), $\rho_{GS} = \rho_J^2$ and $\omega_{\text{opt}} = 2/(1+\sin(\pi/(n+1)))$ (Equation (5.13)). The sweep count (Equation (5.14)) grows like n^2 for Jacobi and Gauss-Seidel and like n for optimal SOR: 4518, 2377 and 155 sweeps for $n = 40$ (**Table 5.1**).
- A small change is not a small error: the remaining error is up to $\rho/(1-\rho)$ times the last change (Equation (5.15)), and the measurements confirm the bound almost to the digit.
- The Java and Python programs agree byte for byte, and SciPy's `spsolve` and conjugate gradient confirm the griddle solution to 10^{-12} and 10^{-9}.

5.13 Exercises

1. Continue the Jacobi table of the four-point griddle by hand for sweeps 7 and 8, starting from the values after sweep 6 printed in Section 5.6. By what factor does the change shrink, and how far is T_1 from 131.25 after sweep 8?

2. For the four-point griddle, write down $T_J = 1/4\,(L+U)$ as a 4×4 matrix and find its eigenvalues. (Hint: try the vectors $(1,1,1,1)$, $(1,-1,-1,1)$, $(1,0,0,-1)$ and $(0,1,-1,0)$.) Use Equation (5.13) to find ρ_{GS} and ω_{opt}, and compare with Section 5.4.3.
3. Is the system $5x_1 + x_2 + 2x_3 = 13$, $x_1 + 4x_2 + x_3 = 12$, $2x_1 + x_2 + 6x_3 = 22$ strictly diagonally dominant? Carry out three Gauss-Seidel sweeps by hand from $x^{(0)} = 0$ and compare with three Jacobi sweeps. The exact solution is $(1,2,3)$.
4. Pak Joni turns the main burner up so that the left edge is at 220 °C. Without iterating, use the fact that the solution of Equation (5.2) depends linearly on the right-hand side to find the new four temperatures. (Hint: the change in the answer solves the same system with right-hand side $(20,0,20,0)$.)
5. The 20×20 griddle has been solved for a left edge of 200 °C. The burner is turned up to 210 °C. Modify `griddle.py` so that SOR with ω_{opt} starts once from zero and once from the old solution (with the new left edge written into the border), and count the sweeps to a change below 10^{-6} in both cases. Explain the result with Equation (5.14).
6. The right edge of the griddle is replaced by a thick wooden handle that lets no heat through. Explain how to change `new_grid` and `sor_sweep` so that the right edge is insulated, as in Section 5.9, and predict, without computing, whether the right half of the plate becomes hotter or cooler.
7. Using Equation (5.14) and the n-dependence of the three spectral radii, estimate how many sweeps Jacobi, Gauss-Seidel and optimal SOR would need for a 160×160 griddle, starting from the counts for $n = 40$ in **Table 5.1**.

5.14 Solutions to the Exercises

1. After sweep 6 the values are 129.3359, 94.3359, 146.8359, 111.8359. Sweep 7: $T_1 = (280 + 94.3359 + 146.8359)/4 = 521.1719/4 = 130.2930$; $T_2 = (140 + 129.3359 + 111.8359)/4 = 381.1719/4 = 95.2930$; $T_3 = (350 + 129.3359 + 111.8359)/4 = 591.1719/4 = 147.7930$; $T_4 = (210 + 94.3359 + 146.8359)/4 = 451.1719/4 = 112.7930$. The change is $130.2930 - 129.3359 = 0.9570$ (up to rounding of the last digit), half of the previous 1.9141. Sweep 8 gives 130.7715, 95.7715, 148.2715, 113.2715 with change 0.4785, again half. After sweep 8, T_1 is $131.25 - 130.7715 = 0.4785$ below the answer: here the error equals the last change, because the error halves each sweep and $\rho/(1-\rho) = 0.5/0.5 = 1$.

2. Row by row, T_J has $1/4$ where two points are neighbors:

$$T_J = \frac{1}{4}\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}.$$

Multiplying: $T_J(1,1,1,1) = 1/4\,(2,2,2,2) = 1/2\,(1,1,1,1)$, eigenvalue $1/2$; $T_J(1,-1,-1,1) = 1/4\,(-2,2,2,-2) = -1/2\,(1,-1,-1,1)$, eigenvalue $-1/2$; $T_J(1,0,0,-1) = 1/4\,(0,0,0,0)$ and $T_J(0,1,-1,0) = 0$, eigenvalue 0 twice. So $\rho_J = 1/2$, which is $\cos(\pi/3)$ from Equation (5.12) with $n = 2$. Then $\rho_{GS} = 1/4$ and $\omega_{\text{opt}} = 2/\left(1+\sqrt{3/4}\right) = 2/1.8660 = 1.0718$, with $\rho(T_{\omega_{\text{opt}}}) = 0.0718$. A numerical eigenvalue computation of T_{GS} and $T_{1.0718}$ gives the same 0.25 and 0.0718.

3. Row 1: $5 > 1 + 2 = 3$; row 2: $4 > 1 + 1 = 2$; row 3: $6 > 2 + 1 = 3$. The matrix is strictly diagonally dominant, so both methods converge. Gauss-Seidel: sweep 1, $x_1 = 13/5 = 2.6$, $x_2 = (12 - 2.6)/4 = 2.35$, $x_3 = (22 - 2\times 2.6 - 2.35)/6 = 14.45/6 = 2.4083$. Sweep 2: $x_1 = (13 -$

$2.35 - 2 \times 2.4083)/5 = 1.1667$, $x_2 = (12 - 1.1667 - 2.4083)/4 = 2.1063$, $x_3 = (22 - 2.3333 - 2.1063)/6 = 2.9267$. Sweep 3: (1.0081,2.0163,2.9946). Jacobi after three sweeps is only at (1.3933,2.2917,3.25) (sweep 1: (2.6,3.0,3.6667); sweep 2: (0.5333,1.4333,2.3)): it overshoots back and forth, while Gauss-Seidel approaches steadily and is about ten times closer.

4. The right-hand side becomes (300,140,370,210), which is the old one plus (20,0,20,0). Because A does not change, the new solution is the old one plus the solution d of $Ad = (20,0,20,0)$. By symmetry $d_1 = d_3$ and $d_2 = d_4$; the first equation gives $4d_1 - d_2 - d_1 = 20$, i.e. $3d_1 - d_2 = 20$, and the second $-d_1 + 4d_2 - d_2 = 0$, i.e. $d_1 = 3d_2$. So $8d_2 = 20$, $d_2 = 2.5$, $d_1 = 7.5$. The new temperatures are $T_1 = 138.75$, $T_2 = 98.75$, $T_3 = 156.25$, $T_4 = 116.25$. Raising the burner by 20 °C raises the near points by 7.5 °C and the far ones by 2.5 °C.

5. Set `griddle.LEFT = 210.0` before calling `new_grid` for the cold start; for the warm start, copy the reference grid computed with 200 °C and write 210.0 into `u[i][0]` for $i = 1, \ldots, n$. With $\omega_{\text{opt}} = 1.741$ the cold start needs 80 sweeps and the warm start 57. By Equation (5.14) the count depends on $\ln(\varepsilon/E_0)$: from zero the first error is about 200 °C, from the old map it is at most 10 °C (the change of the left edge, which spreads inward and is smaller everywhere else). The ratio 20 is worth $\ln 20/\ln(1/0.741) \approx 10$ sweeps by the asymptotic rate, and more in practice because the warm start also avoids SOR's initial transient. On bigger grids and with smaller changes the saving grows.

6. In `new_grid`, the right border column no longer holds a fixed temperature. In `sor_sweep`, before the double loop, set `u[i][n + 1] = u[i][n]` for every row i (the value just outside equals the value just inside, so no heat crosses the edge); the matrix row of a point next to the right edge then has 3 instead of 4 on its effective diagonal and is no longer strictly dominant, but the whole system is still irreducibly dominant because the other three edges are fixed, and SOR still converges. The right half becomes hotter: before, heat flowed out through the right edge at 60 °C; now it cannot leave there, so it must leave through the top and bottom edges, and the plate near the handle warms toward the temperatures of the other edges.

7. Jacobi and Gauss-Seidel sweeps grow like n^2 (more precisely like $(n+1)^2$, with a logarithmic factor from E_0 that we ignore), and optimal SOR like $n + 1$. From $n = 40$ to $n = 160$, $(161/41)^2 \approx 15.4$ and $161/41 \approx 3.93$. Estimates: Jacobi $4518 \times 15.4 \approx 70{,}000$ sweeps, Gauss-Seidel $2377 \times 15.4 \approx 37{,}000$, optimal SOR $155 \times 3.93 \approx 610$. With 25,600 unknowns and about five operations per unknown, Jacobi would need about 9×10^9 operations, SOR about 8×10^7, more than a hundred times fewer.

References

[1] S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers*, 8th ed. New York, NY, USA: McGraw-Hill Education, 2021.

[2] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical Analysis*, 10th ed. Boston, MA, USA: Cengage Learning, 2016.

[3] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed. Baltimore, MD, USA: Johns Hopkins University Press, 2013.

[4] D. M. Young, *Iterative Solution of Large Linear Systems*. New York, NY, USA: Academic Press, 1971 (reprinted by Dover, 2003).

[5] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: SIAM, 2003.

[6] Python Software Foundation, "What's New In Python 3.12: sum()," accessed Sep. 2026. [Online]. Available: https://docs.python.org/3/whatsnew/3.12.html

[7] SciPy Developers, “SciPy 1.18.0 Release Notes,” Jun. 19, 2026, and “scipy 1.18.1” on the Python Package Index, Aug. 21, 2026. [Online]. Available: https://docs.scipy.org/doc/scipy/release/1.18.0-notes.html and https://pypi.org/project/scipy/

[8] NumPy Developers, “NumPy 2.5.0 Release Notes,” Jun. 21, 2026, and “numpy 2.5.3” on the Python Package Index, Sep. 6, 2026. [Online]. Available: https://numpy.org/doc/2.5/release/2.5.0-notes.html and https://pypi.org/project/numpy/

[9] SciPy Developers, “scipy.sparse.linalg.cg,” SciPy v1.18 Manual, accessed Sep. 2026. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.cg.html

[10] R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd ed. Philadelphia, PA, USA: SIAM, 1994.

Chapter 6

Matrix Decompositions: LU, Cholesky and QR

Gaussian elimination, as Chapter 4 did it, throws its work away: once the answer is out, the multipliers that made it possible are forgotten, and a new right-hand side means starting again from the top. This chapter keeps that work. It writes a matrix as a product of simpler matrices, a *decomposition*, and then reuses the pieces as often as it likes. The running example is a fruit and durian orchard that must turn an extension officer's nutrient recommendations into grams of fertilizer, block after block and season after season. By the end of the chapter you will be able to factor a matrix as $PA = LU$ with partial pivoting and solve for many right-hand sides with two triangular substitutions each; compute a determinant and an inverse from the factors; recognize a symmetric positive definite matrix and factor it by Cholesky at half the cost; set up a least squares problem when there are more requirements than unknowns and solve it both by the normal equations and by Householder QR; explain, with numbers, why QR is the safer of the two; build all of it in Java and Python; and check it against SciPy.

6.1 Pak Harun's Fertilizer Question

Pak Harun and his family look after a fruit and durian orchard on a hillside. The trees are grouped in four blocks: young trees planted three years ago, mature trees just after harvest, mature trees before flowering, and trees with fruit swelling on the branches. Twice a year an agricultural extension officer (*penyuluh pertanian*) looks at the trees and at a soil test and writes a recommendation for each block, in grams of the three main nutrients per tree: nitrogen (N), phosphate (written P_2O_5) and potash (K_2O). For the young trees this season it says 150 g of N, 60 g of phosphate and 70 g of potash per tree.

The farm shop in the village does not sell nutrients; it sells bags of fertilizer, each with a *grade* printed on it. A bag labeled NPK 12-12-17 contains 12% N, 12% P_2O_5 and 17% K_2O by weight; NPK 20-10-10 is richer in nitrogen; urea, 46-0-0, is almost half nitrogen and nothing else. Pak Harun keeps these three. The question, for every block, is how many grams of each fertilizer to spread under each tree so that the tree gets exactly what the recommendation says.

Written out, the question is a linear system, three equations (one per nutrient) and three unknowns (one per fertilizer). Chapter 4 can solve it. But Pak Harun does not have one system, he has four this season, four more next season, and the recommendation changes whenever the officer changes his mind. The matrix of grades stays the same for years; only the right-hand side changes. This chapter shows how to do the expensive part of elimination once and reuse it, and then what to do when the officer adds a fourth nutrient, magnesium, and three fertilizers can no longer meet four targets exactly.

A practical word before the mathematics. The program computes a blend that delivers a given recommendation; it does not decide the recommendation. The dose for a tree depends on its age, the soil and the season, and too much fertilizer burns roots and pollutes streams. The numbers in

this chapter are an example; for your own trees, ask the extension officer or a soil laboratory, and use the program to turn their advice into grams.

6.2 The Idea in Plain Words

6.2.1 The equations

Measure the fertilizers in units of 100 grams, so that one unit of NPK 12-12-17 contains 12 g of N, 12 g of phosphate and 17 g of potash. If x_1, x_2 and x_3 are the amounts (in hundreds of grams) of NPK 12-12-17, NPK 20-10-10 and urea, the nitrogen delivered is $12x_1 + 20x_2 + 46x_3$, and so on. **Table 6.1** collects the grades; each column is one bag, each row one nutrient.

Table 6.1 Grams of each nutrient in 100 g of each fertilizer, and the recommendation for one young tree.

Nutrient	NPK 12-12-17	NPK 20-10-10	Urea 46-0-0	Young trees need
N	12	20	46	150 g
P_2O_5	12	10	0	60 g
K_2O	17	10	0	70 g

The system for the young trees is therefore

$$\begin{pmatrix} 12 & 20 & 46 \\ 12 & 10 & 0 \\ 17 & 10 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 150 \\ 60 \\ 70 \end{pmatrix}, \tag{6.1}$$

$Ax = b$ with the grade matrix A and the recommendation b. The other three blocks have the same A and different b: (180, 90, 110) after harvest, (120, 90, 120) before flowering and (100, 80, 110) during fruit filling.

6.2.2 Elimination remembers: finding L and U by hand

Run Gaussian elimination with partial pivoting on A (Chapter 4), but this time write down every multiplier.

Step 1. The largest entry of column 1 in absolute value is 17, in the potash row, so swap it to the top. The rows are now in the order K, P, N:

$$\begin{pmatrix} 17 & 10 & 0 \\ 12 & 10 & 0 \\ 12 & 20 & 46 \end{pmatrix}.$$

The multiplier for the P row is $12/17 = 0.7059$: subtracting 0.7059 times the top row turns (12,10,0) into $(0,\ 10 - 7.0588,\ 0) = (0,\ 2.9412,\ 0)$. The multiplier for the N row is also $12/17$: (12,20,46) becomes $(0,\ 12.9412,\ 46)$.

Step 2. In column 2, below the top row, the candidates are 2.9412 (P row) and 12.9412 (N row). The larger is in the N row, so swap the last two rows; the order is now K, N, P, and the multipliers travel with their rows. The multiplier for the P row is $2.9412/12.9412 = 0.2273$ (exactly $5/22$), and (0,2.9412,0) becomes $(0,\ 0,\ 0 - 0.2273 \times 46) = (0{,}0{,}-10.4545)$.

What is left is the upper triangular matrix U, and the multipliers, placed where they were used, form a lower triangular matrix L with ones on its diagonal:

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0.7059 & 1 & 0 \\ 0.7059 & 0.2273 & 1 \end{pmatrix}, \qquad U = \begin{pmatrix} 17 & 10 & 0 \\ 0 & 12.9412 & 46 \\ 0 & 0 & -10.4545 \end{pmatrix}.$$

The row swaps are recorded in a *permutation matrix* P, the identity matrix with its rows in the order they ended up in (K, N, P): multiplying by P puts the rows of A in that order. The claim of this chapter is that

$$PA = LU. \tag{6.2}$$

You can check it by multiplying: the second row of LU is $0.7059 \times (17,10,0) + (0,12.9412,46) = (12,20,46)$, which is the N row of A, the second row of PA. Nothing new has been computed. L and U are simply elimination's work, written down instead of thrown away.

6.2.3 Two triangles per block

With the factors in hand, solving $Ax = b$ is two easy steps. Multiply $Ax = b$ by P: $PAx = Pb$, that is $L(Ux) = Pb$. Call $y = Ux$. Then

$$Ly = Pb \quad \text{(forward substitution)}, \qquad Ux = y \quad \text{(back substitution)}. \tag{6.3}$$

For the young trees, Pb puts the recommendation in the order K, N, P: (70,150,60). Forward, top to bottom, because L is lower triangular:

- $y_1 = 70$
- $y_2 = 150 - 0.7059 \times 70 = 150 - 49.4118 = 100.5882$
- $y_3 = 60 - 0.7059 \times 70 - 0.2273 \times 100.5882 = 60 - 49.4118 - 22.8610 = -12.2727$

Backward, bottom to top, because U is upper triangular:

- $x_3 = -12.2727/-10.4545 = 1.1739$
- $x_2 = (100.5882 - 46 \times 1.1739)/12.9412 = (100.5882 - 54.0000)/12.9412 = 3.6$
- $x_1 = (70 - 10 \times 3.6 - 0 \times 1.1739)/17 = 34/17 = 2$

So each young tree gets 200 g of NPK 12-12-17, 360 g of NPK 20-10-10 and 117.4 g of urea. Check the nitrogen: $12 \times 2 + 20 \times 3.6 + 46 \times 1.1739 = 24 + 72 + 54.0 = 150$ g. For the next block only the two substitutions are repeated, a handful of multiplications, while the elimination itself is never done again. **Figure 6.1** pictures the pattern: factor once, then two triangles per right-hand side.

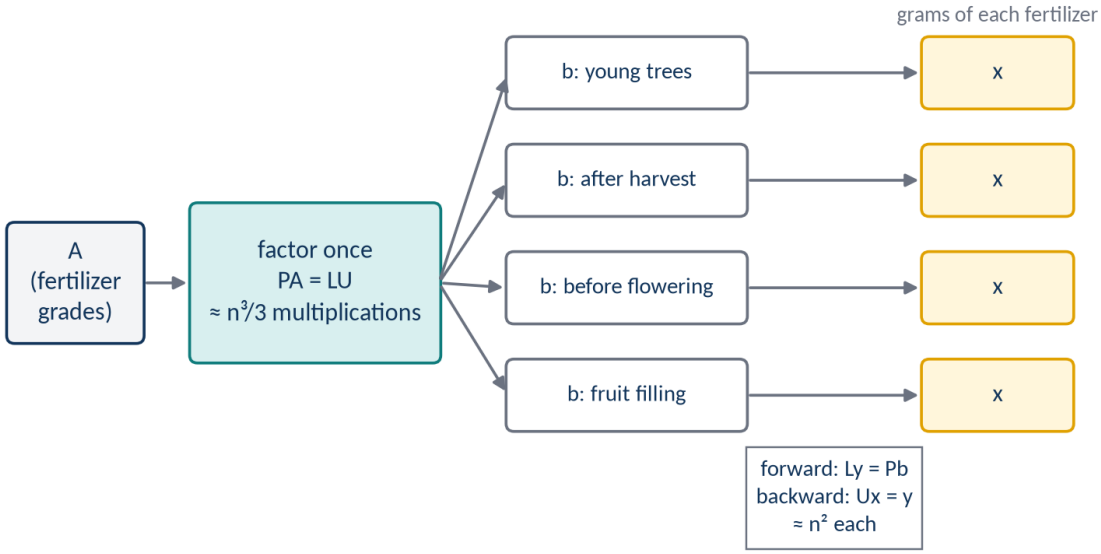


Figure 6.1 Factor once, solve many: the grade matrix is factored as $PA = LU$ once, and each block's recommendation costs only a forward and a back substitution.

6.3 LU Decomposition

6.3.1 Where the factors come from

Why is the product of the multipliers and the last rows equal to A? Look at what elimination does to row i: it subtracts multiples l_{ik} of the pivot rows $u_{k\cdot}$ (the rows of U) for $k = 1,2,\dots,i-1$, and what remains is row i of U. Reading this backward, the original row is the sum of what was subtracted plus what remained:

$$\text{row } i \text{ of } A = \sum_{k<i} l_{ik}\ (\text{row } k \text{ of } U) + 1 \cdot (\text{row } i \text{ of } U),$$

and that is exactly row i of LU when L has the multipliers below the diagonal and ones on it. With row swaps the same argument applies to the rows of PA. Comparing entries of $A = LU$ gives the factors directly, without talking about elimination. This is the *Doolittle* form of the algorithm [1], [2]: for $k = 1,\dots,n$,

$$u_{kj} = a_{kj} - \sum_{m<k} l_{km}\,u_{mj} \quad (j \geq k), \qquad l_{ik} = \frac{1}{u_{kk}}\left(a_{ik} - \sum_{m<k} l_{im}\,u_{mk}\right) \quad (i > k). \tag{6.4}$$

The first formula computes row k of U from what is known, the second column k of L. Each needs $u_{kk} \neq 0$, which is where pivoting comes in: before computing column k of L, choose among the rows $i \geq k$ the one with the largest $|a_{ik} - \sum l_{im}u_{mk}|$ and swap it into position k. **Pseudocode 6.1** does exactly this in place: the matrix is overwritten by its factors, U on and above the diagonal and the multipliers of L below it (the ones on L's diagonal are not stored), and a vector `perm` records the swaps.

Pseudocode 6.1 LU decomposition with partial pivoting, stored in place, and the two triangular solves.

```
LU-DECOMPOSE(A)                      // A is n x n, overwritten
 1  perm = (1, 2, ..., n); sign = 1
 2  for k = 1 to n - 1
 3      p = the row i >= k with the largest |A[i][k]|
 4      if A[p][k] == 0: error "singular"
 5      if p != k: swap rows k and p of A and of perm; sign = -sign
 6      for i = k + 1 to n
 7          A[i][k] = A[i][k] / A[k][k]            // multiplier l_ik
 8          for j = k + 1 to n
 9              A[i][j] = A[i][j] - A[i][k] * A[k][j]
10  return A, perm, sign

LU-SOLVE(A, perm, b)
11  for i = 1 to n: y[i] = b[perm[i]] - sum_{j<i} A[i][j] * y[j]
12  for i = n downto 1: x[i] = (y[i] - sum_{j>i} A[i][j] * x[j]) / A[i][i]
13  return x
```

DEFINITION. An *LU decomposition* of a square matrix A is a factorization $PA = LU$ with a permutation matrix P, a unit lower triangular matrix L (ones on the diagonal) and an upper triangular matrix U. With partial pivoting it exists for every square matrix, and it is unique when A is nonsingular and the pivoting rule is fixed [3].

6.3.2 What it costs

Count multiplications, the traditional measure (additions come in equal number). Step k of the factorization updates an $(n-k)\times(n-k)$ block, about $(n-k)^2$ multiplications, so the whole factorization costs

$$\sum_{k=1}^{n-1}(n-k)^2 \approx \frac{n^3}{3}, \qquad \text{while each solve costs} \quad \approx n^2. \tag{6.5}$$

The forward substitution needs about $n^2/2$ multiplications and the back substitution another $n^2/2$. Solving K systems with the same matrix by elimination from scratch costs $K(n^3/3+n^2)$; with LU it costs $n^3/3 + Kn^2$. **Figure 6.2** plots both for $n = 100$ and up to 100 right-hand sides: the saving is a factor of 26 at $K = 100$, and it grows with n, to 77 for $n = 1000$ (the program of Section 6.6 prints these counts).

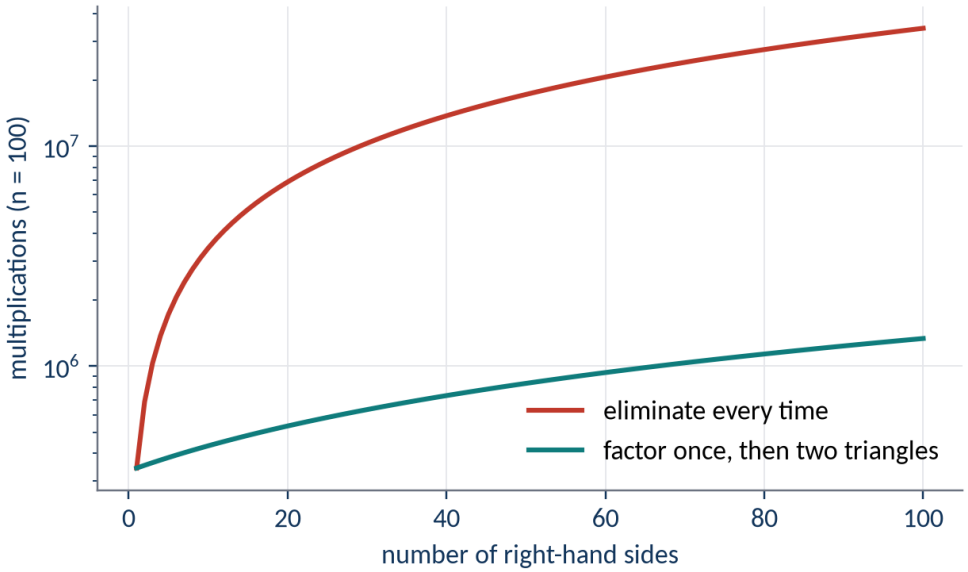


Figure 6.2 Multiplications needed for K right-hand sides with a 100×100 matrix: eliminating every time grows like $Kn^3/3$, factoring once like $n^3/3 + Kn^2$.

6.3.3 Determinant and inverse for free

The determinant of a product is the product of the determinants. A triangular matrix has its determinant on the diagonal, $\det L = 1$, and a permutation has determinant $+1$ or -1 depending on whether it made an even or odd number of swaps. From Equation (6.2),

$$\det A = (\pm 1)\, u_{11} u_{22} \cdots u_{nn}. \tag{6.6}$$

For the grade matrix: two swaps, so the sign is $+1$, and $\det A = 17 \times 12.9412 \times (-10.4545) = -2300$ (exactly, since $12.9412 = 220/17$ and $-10.4545 = -115/11$). A nonzero determinant confirms that every recommendation has exactly one blend. Computing a determinant by cofactor expansion, as some courses teach, costs about $n!$ operations; by Equation (6.6) it costs $n^3/3$. For $n = 20$ that is 2.4×10^{18} against 2,700.

The inverse A^{-1} is the matrix whose column j solves $Ax = e_j$, where e_j is column j of the identity. With the factors, that is n pairs of triangular solves, about n^3 multiplications in total. You rarely need it: to solve $Ax = b$, the two triangles are cheaper and more accurate than forming A^{-1} and multiplying. But its columns can have a meaning of their own, as Exercise 3 shows for the orchard.

6.3.4 How accurate is it?

Chapter 4 showed that partial pivoting keeps the multipliers at most 1 in absolute value, so errors are not amplified by large multipliers, and that the answer's accuracy is governed by the *condition number* $\kappa(A)$: roughly, a computed solution loses $\log_{10}\kappa(A)$ of the sixteen digits a `double` carries. Wilkinson's backward error analysis states it precisely: LU with partial pivoting computes the exact factors of a matrix $A + \delta A$ with $\| \delta A \|$ a modest multiple of $\varepsilon_{\text{mach}} \| A \|$ in all but contrived cases [4]. For the grade matrix $\kappa(A) \approx 27$, and the program finds $\| PA - LU \| = 0$ to the last bit: the entries are small integers and every rounding happens to cancel.

6.3.5 What goes wrong without pivoting

It is worth seeing once why the swaps of Section 6.2.2 are not optional. Take the 2×2 system $\varepsilon x_1 + x_2 = 1$, $x_1 + x_2 = 2$ with $\varepsilon = 10^{-17}$, whose solution is extremely close to $x_1 = x_2 = 1$.

Without pivoting, the first pivot is ε and the multiplier is $l_{21} = 1/\varepsilon = 10^{17}$. The new diagonal entry is $u_{22} = 1 - 10^{17}$, which in double precision rounds to -10^{17}: the 1 is lost, because a `double` keeps only about 16 digits. Forward substitution gives $y_2 = 2 - 10^{17}$, which rounds to -10^{17} as well, so $x_2 = 1$, correctly. But then $x_1 = (1 - x_2)/\varepsilon = 0/10^{-17} = 0$, which is completely wrong. The information about x_1 was destroyed the moment the huge multiplier swamped the second row. With pivoting the rows are swapped first, the pivot is 1, the multiplier is ε, nothing is swamped, and both unknowns come out as 1. The grade matrix of this chapter is harmless without pivoting too, but a program that factors whatever matrix it is given must pivot, and **Pseudocode 6.1** does.

6.4 Cholesky: Half the Work for Symmetric Positive Definite Matrices

6.4.1 A fourth nutrient

After a leaf analysis the extension officer adds a line to the recommendation for the young trees: 6 g of magnesium oxide (MgO) per tree, because the older leaves are yellowing between the veins. Pak Harun's NPK 12-12-17 is sold as 12-12-17-2, with 2% MgO; the other two fertilizers contain none. Now there are four requirements and still three fertilizers:

$$\begin{pmatrix} 12 & 20 & 46 \\ 12 & 10 & 0 \\ 17 & 10 & 0 \\ 2 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \approx \begin{pmatrix} 150 \\ 60 \\ 70 \\ 6 \end{pmatrix}. \tag{6.7}$$

The first three rows alone force $x_1 = 2$ (Section 6.2.3), and then the last row delivers 4 g of MgO, not 6. No blend of these three bags meets all four targets. The system is *overdetermined*, and the honest question is no longer "which blend meets the targets" but "which blend comes closest". Measuring closeness by the sum of squared misses gives the **least squares problem**

$$\min_{x} \; \| \, b - Ax \, \|_2^2 = \min_{x} \sum_{i=1}^{m} \left(b_i - \sum_{j=1}^{n} a_{ij} \, x_j \right)^2, \tag{6.8}$$

here with $m = 4$ requirements and $n = 3$ unknowns. Squares are chosen because they make the problem smooth and linear to solve, they punish big misses more than small ones, and they lead to the geometry of **Figure 6.3**; Chapter 8 is devoted to them.

6.4.2 The normal equations

Every blend x delivers a vector Ax, and all these vectors together form the *column space* of A, a flat three-dimensional subspace inside the four-dimensional space of nutrient vectors. The target b sticks out of it. The closest point to b in a flat space is the foot of the perpendicular: the best blend $\hat{x}$ makes the residual $r = b - A\hat{x}$ perpendicular to every column of A, as **Figure 6.3** shows. Perpendicular to column j means the dot product is zero, $\sum_i a_{ij} \, r_i = 0$, and for all columns at once $A^{\mathsf{T}} r = 0$. Substituting r:

$$A^{\mathsf{T}} A \, \hat{x} = A^{\mathsf{T}} b. \tag{6.9}$$

These are the **normal equations** ("normal" in its old sense of perpendicular). The same equations come out of calculus: setting the derivative of Equation (6.8) with respect to each x_j to zero gives $-2\sum_i a_{ij}\,(b_i - (Ax)_i) = 0$, which is row j of Equation (6.9).

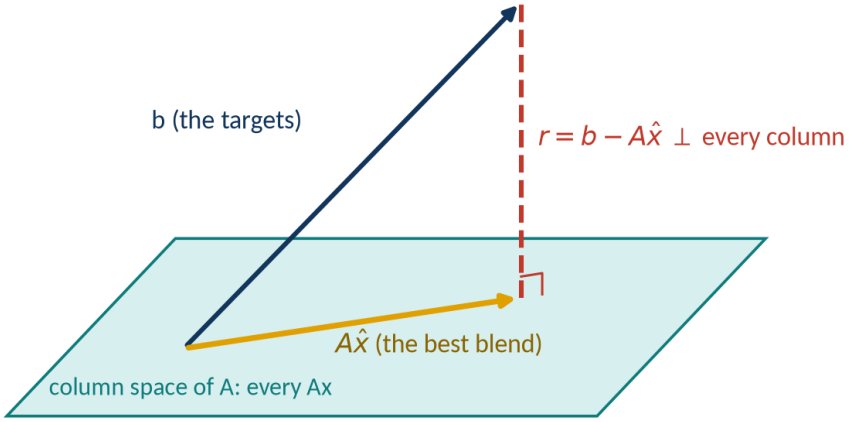


Figure 6.3 The least squares blend: $A\hat{x}$ is the point of the column space closest to the target b, and the miss $r = b - A\hat{x}$ is perpendicular to every column.

For the orchard, $A^{\mathsf{T}}A$ is a 3×3 matrix. Its entry (1,1) is the column of NPK 12-12-17-2 dotted with itself, $12^2 + 12^2 + 17^2 + 2^2 = 581$; entry (1,2) is $12 \cdot 20 + 12 \cdot 10 + 17 \cdot 10 + 2 \cdot 0 = 530$; and so on:

$$A^{\mathsf{T}}A = \begin{pmatrix} 581 & 530 & 552 \\ 530 & 600 & 920 \\ 552 & 920 & 2116 \end{pmatrix}, \qquad A^{\mathsf{T}}b = \begin{pmatrix} 3722 \\ 4300 \\ 6900 \end{pmatrix}.$$

6.4.3 Symmetric positive definite matrices

$A^{\mathsf{T}}A$ is special in two ways. It is *symmetric*, because entry (i, j) and entry (j, i) are the same dot product. And it is *positive definite*: for any nonzero vector z, $z^{\mathsf{T}}(A^{\mathsf{T}}A)z = (Az)^{\mathsf{T}}(Az) = \| Az \|^2 > 0$, as long as the columns of A are independent (no fertilizer is a mix of the others), so that $Az \neq 0$.

> **DEFINITION.** A square matrix S is *symmetric positive definite* (SPD) if $S^{\mathsf{T}} = S$ and $z^{\mathsf{T}}Sz > 0$ for every nonzero vector z. Such matrices arise from sums of squares: energies of springs, variances, the normal equations, and the griddle of Chapter 5.

For an SPD matrix, elimination never needs pivoting, and U turns out to be L^{T} up to the diagonal. Splitting the diagonal evenly between the two factors gives the **Cholesky decomposition** $S = LL^{\mathsf{T}}$, with L lower triangular and a positive diagonal. It was worked out by André-Louis Cholesky, a French artillery officer and surveyor, for the normal equations of map-making, and published after his death in the First World War by a colleague [5]. The formulas come from comparing entries of $S = LL^{\mathsf{T}}$ column by column: for $j = 1, \ldots, n$,

$$l_{jj} = \sqrt{s_{jj} - \sum_{k<j} l_{jk}^2}\,, \qquad l_{ij} = \frac{1}{l_{jj}}\left(s_{ij} - \sum_{k<j} l_{ik}\, l_{jk} \right) \quad (i > j). \tag{6.10}$$

Only half of the matrix is touched, so the cost is about $n^3/6$ multiplications and n square roots, half of LU. The number under the square root is always positive when S is SPD, and this is a free test: if it turns out zero or negative, S is not positive definite (or rounding has made it look so), and the algorithm stops. Solving $Sx = c$ is then the two triangles $Ly = c$ and $L^{\mathsf{T}}x = y$.

By hand for the orchard: $l_{11} = \sqrt{581} = 24.1039$; $l_{21} = 530/24.1039 = 21.9881$; $l_{31} = 552/24.1039 = 22.9008$; $l_{22} = \sqrt{600 - 21.9881^2} = \sqrt{600 - 483.477} = \sqrt{116.523} = 10.7946$; $l_{32} = (920 - 21.9881 \times 22.9008)/10.7946 = (920 - 503.545)/10.7946 = 38.5799$; $l_{33} = \sqrt{2116 - 22.9008^2 - 38.5799^2} = \sqrt{2116 - 524.448 - 1488.409} = \sqrt{103.143} = 10.1559$. The two triangular solves then give the best blend, which Section 6.7 prints: 224.2 g of NPK 12-12-17-2, 324.8 g of NPK 20-10-10 and 126.4 g of urea.

6.4.4 Why Cholesky needs no pivoting

LU needed pivoting because a small pivot produces large multipliers, and large multipliers amplify rounding. Cholesky never meets that danger on an SPD matrix, for two reasons [4]. First, every diagonal entry of an SPD matrix is positive ($s_{jj} = e_j^{\mathsf{T}} S e_j > 0$), and after one step of the factorization the part of the matrix that remains to be factored is again SPD, so every pivot under the square root is positive. Second, the entries of L cannot grow: row j of Equation (6.10) says $l_{j1}^2 + \cdots + l_{jj}^2 = s_{jj}$, so no entry of row j of L can exceed $\sqrt{s_{jj}}$ in absolute value. For the orchard, row 3 of L is (22.9008,38.5799,10.1559) and $22.9008^2 + 38.5799^2 + 10.1559^2 = 2116.0$, exactly s_{33}. Bounded entries mean bounded rounding, which is why Cholesky is both the cheapest and one of the most stable factorizations known. The one thing it cannot fix is the conditioning of the matrix itself, which is the subject of the next section.

6.5 QR: Least Squares Without Squaring

6.5.1 What the normal equations cost in accuracy

The normal equations are quick, but they have a hidden price. The condition number of $A^{\mathsf{T}}A$ is the square of that of A:

$$\kappa(A^{\mathsf{T}}A) = \kappa(A)^2. \tag{6.11}$$

(For the 2-norm this follows from the singular values: those of $A^{\mathsf{T}}A$ are the squares of those of A [6].) If $\kappa(A) = 10^6$, the least squares problem itself deserves an answer good to about $16 - 6 = 10$ digits, but the normal equations have $\kappa = 10^{12}$ and deliver only about 4. For the orchard this does not matter ($\kappa(A) \approx 24$, $\kappa(A^{\mathsf{T}}A) \approx 560$). For fitting a polynomial of degree 9 to data, which Chapter 8 needs, it matters enormously: Section 6.7 shows the normal equations losing seven digits there, and failing outright at degree 12.

6.5.2 Orthogonal matrices do not stretch

A square matrix Q is *orthogonal* if its columns are unit vectors perpendicular to one another, which is the same as $Q^{\mathsf{T}}Q = I$. Multiplying by Q is a rotation or a reflection: it does not change the length of any vector, $\| Qz \| = \| z \|$, because $\| Qz \|^2 = z^{\mathsf{T}}Q^{\mathsf{T}}Qz = z^{\mathsf{T}}z$. Suppose we could write $A = QR$ with an $m \times m$ orthogonal Q and an $m \times n$ matrix R whose top $n \times n$ block R_1 is upper triangular and whose remaining $m - n$ rows are zero. Multiply the residual by Q^{T}, which does not change its length, and split $c = Q^{\mathsf{T}}b$ into its top n entries c_1 and the rest c_2:

$$\| b - Ax \|^2 = \| Q^\top b - Rx \|^2 = \| c_1 - R_1 x \|^2 + \| c_2 \|^2. \tag{6.12}$$

The second term does not depend on x at all. The first becomes zero by solving the triangular system $R_1 x = c_1$ with back substitution. So the least squares solution is one back substitution, and the size of the unavoidable miss is $\| c_2 \|$, for free. No product $A^\top A$ is ever formed, nothing is squared, and the accuracy is governed by $\kappa(A)$ instead of $\kappa(A)^2$ [6].

6.5.3 Householder reflections

How do we find Q? Alston Householder's idea (1958) builds it from reflections [7]. Given a vector a (the first column of A), we want an orthogonal matrix that maps it to a multiple of $e_1 = (1,0,\dots,0)$, zeroing everything below the first entry, just as the first step of elimination does, but without stretching anything. Since the map preserves length, the target must be αe_1 with $|\alpha| = \| a \|$. The mirror that swaps a and αe_1 is perpendicular to $v = a - \alpha e_1$, and the reflection in that mirror is

$$H = I - \frac{2\, vv^\top}{v^\top v}. \tag{6.13}$$

H is symmetric and orthogonal ($H^2 = I$: reflecting twice gives back the original), and $Ha = \alpha e_1$. Two practical points. First, which sign for α? Choosing $\alpha = -\text{sign}(a_1) \| a \|$ makes the first entry of $v = a - \alpha e_1$ equal to $a_1 + \text{sign}(a_1) \| a \|$, a sum of two numbers of the same sign, so there is no cancellation (Chapter 1) even when a is nearly parallel to e_1. Second, H is never formed as a matrix: applying it to a column z costs one dot product, $Hz = z - (2v^\top z / v^\top v)\, v$.

Apply the first reflection to all columns of A and to b; column 1 becomes $(\alpha, 0, \dots, 0)$. Then repeat on the rows and columns below and to the right, with a reflection that leaves the first row alone, and so on for all n columns. What remains of A is R, what remains of b is $c = Q^\top b$, and Q itself, the product of the reflections, is never needed. The work is about $mn^2 - 1/3\, n^3$ multiplications, twice as much as forming and solving the normal equations when m is much larger than n: the price of not squaring the condition number [3], [6].

6.5.4 A check that connects the two methods

Since $A^\top A = R^\top Q^\top Q R = R_1^\top R_1$, the triangular factor of QR and the Cholesky factor of the normal equations are the same matrix up to the signs of its rows: $R_1 = \pm L^\top$. Section 6.8 confirms it: NumPy's R has diagonal -24.1039, 10.7946, -10.1559, and our Cholesky factor has diagonal 24.1039, 10.7946, 10.1559. The two methods compute the same thing along different roads; the road through $A^\top A$ is shorter and bumpier.

6.5.5 Gram-Schmidt, the other road to QR

Many textbooks build Q differently, by the *Gram-Schmidt* process: take the columns of A one at a time, subtract from each its components along the columns already accepted, and normalize what remains. This produces the "thin" factorization, $A = \hat{Q} R_1$ with an $m \times n$ matrix $\hat{Q}$ of orthonormal columns, and it is how most people first understand QR. In floating-point arithmetic, however, the classical version loses orthogonality when the columns are nearly dependent: the computed columns of $\hat{Q}$ are then far from perpendicular, and the least squares answer suffers. The *modified* Gram-Schmidt process, which subtracts the components one at a time from the running

remainder, is much better, but Householder reflections are orthogonal by construction whatever the columns look like, which is why LAPACK, and therefore NumPy and SciPy, use them [3], [6].

Table 6.2 puts the three decompositions side by side.

Table 6.2 The three decompositions of this chapter, what they need, what they cost and what they are for.

Decomposition	For which matrices	Cost (multiplications)	Used for
$PA = LU$	any square	$n^3/3$	square systems, many right-hand sides, determinant, inverse
$S = LL^{\mathsf{T}}$ (Cholesky)	symmetric positive definite	$n^3/6$	SPD systems, normal equations, a test for SPD
$A = QR$ (Householder)	any $m \times n$, $m \geq n$	$mn^2 - n^3/3$	least squares without squaring κ

6.6 Building It in Java and Python

Two programs again. `lu_blend` factors the grade matrix once, prints P, L and U, checks $PA = LU$, computes the determinant, solves for all four blocks and prints the operation counts of Equation (6.5). `least_blend` solves the four-nutrient problem Equation (6.7) by the normal equations with Cholesky and by Householder QR, shows which nutrients are missed, and then repeats the comparison on polynomial columns that become ill-conditioned as the degree grows.

6.6.1 LU in Python and Java

Listing 6.1 follows **Pseudocode 6.1** line by line. `lu_decompose` returns the packed factors, the permutation and the sign; `lu_solve` is the pair of substitutions Equation (6.3); `main` unpacks L and U only for printing. The work counts use the leading terms $n^3/3$ and n^2.

Listing 6.1 `lu_blend.py`: LU decomposition with partial pivoting, the determinant, and the blend for every block from one factorization.

```
"""Fertilizer blends for four orchard blocks: factor A = LU once (with
partial pivoting), then solve for every block with two cheap triangles."""

NAMES = ["NPK 12-12-17", "NPK 20-10-10", "Urea 46-0-0"]
# rows: grams of N, P2O5, K2O in 100 g of each fertilizer (one per column)
A = [[12.0, 20.0, 46.0],
     [12.0, 10.0, 0.0],
     [17.0, 10.0, 0.0]]
BLOCKS = [("young trees", [150.0, 60.0, 70.0]),
          ("after harvest", [180.0, 90.0, 110.0]),
          ("before flowering", [120.0, 90.0, 120.0]),
          ("fruit filling", [100.0, 80.0, 110.0])]



def total(values):
    """Plain left-to-right sum. Python 3.12+ sum() compensates rounding,
    which would make our digits differ from the Java version."""
    s = 0.0
    for v in values:
        s += v
    return s

def lu_decompose(a):
    """Return (lu, perm, sign): L below the diagonal (unit diagonal not
    stored), U on and above it, perm[i] = original row now in row i."""
    n = len(a)
    lu = [row[:] for row in a]
    perm = list(range(n))
    sign = 1.0
    for k in range(n - 1):
        p = max(range(k, n), key=lambda i: abs(lu[i][k]))   # partial pivot
        if lu[p][k] == 0.0:
            raise ValueError("matrix is singular")
        if p != k:
            lu[k], lu[p] = lu[p], lu[k]
            perm[k], perm[p] = perm[p], perm[k]
            sign = -sign
        for i in range(k + 1, n):
            m = lu[i][k] / lu[k][k]
            lu[i][k] = m                    # keep the multiplier: it is L
            for j in range(k + 1, n):
                lu[i][j] -= m * lu[k][j]
    return lu, perm, sign
```

Listing 6.1 (Continued) *lu_blend.py: LU decomposition with partial pivoting, the determinant, and the blend for every block from one factorization.*

```
def lu_solve(lu, perm, b):
    """Solve Ly = Pb (forward), then Ux = y (backward)."""
    n = len(lu)
    y = [0.0] * n
    for i in range(n):
        y[i] = b[perm[i]] - total(lu[i][j] * y[j] for j in range(i))
    x = [0.0] * n
    for i in range(n - 1, -1, -1):
        s = y[i] - total(lu[i][j] * x[j] for j in range(i + 1, n))
        x[i] = s / lu[i][i]
    return x


def show(title, m):
    print(title)
    for row in m:
        print("  " + "".join(f"{v:10.4f}" for v in row))


def main():
    n = len(A)
    lu, perm, sign = lu_decompose(A)
    p = [[1.0 if j == perm[i] else 0.0 for j in range(n)] for i in range(n)]
    low = [[lu[i][j] if j < i else (1.0 if i == j else 0.0)
            for j in range(n)] for i in range(n)]
    up = [[lu[i][j] if j >= i else 0.0 for j in range(n)] for i in range(n)]
    show("P", p)
    show("L", low)
    show("U", up)
    err = max(abs(total(low[i][k] * up[k][j] for k in range(n))
                  - A[perm[i]][j]) for i in range(n) for j in range(n))
    print(f"max |PA - LU| = {err:.1e}")
    det = sign
    for i in range(n):
        det *= lu[i][i]
    print(f"det A = {det:.4f}")
    print()
    print("grams per tree         " + "".join(f"{s:>14s}" for s in NAMES))
    for name, b in BLOCKS:
        x = lu_solve(lu, perm, b)
        print(f"{name:23s}" + "".join(f"{100 * v:14.1f}" for v in x))
    print()
    print("work for 100 right-hand sides (multiplications, leading terms)")
    for size in (3, 100, 1000):
        again = 100 * (size ** 3 / 3 + size ** 2)
        once = size ** 3 / 3 + 100 * size ** 2
```

Listing 6.1 (Continued) *lu_blend.py: LU decomposition with partial pivoting, the determinant, and the blend for every block from one factorization.*

```
        print(f"  n = {size:4d}: eliminate each time {again:14.0f},"
              f" factor once {once:14.0f}")


if __name__ == "__main__":
    main()
```

Listing 6.2 is the Java version; the factors live in static fields so that `luSolve` can be called once per block.

Listing 6.2 `LuBlend.java`: the same factor-once, solve-many program in Java.

```
import java.math.BigDecimal;
import java.math.MathContext;
import java.math.RoundingMode;

/** Fertilizer blends for four orchard blocks: factor A = LU once (with
 *  partial pivoting), then solve for every block with two cheap triangles. */
public class LuBlend {
    static final String[] NAMES = {"NPK 12-12-17", "NPK 20-10-10", "Urea 46-0-0"};
    // rows: grams of N, P2O5, K2O in 100 g of each fertilizer (one per column)
    static final double[][] A = {{12, 20, 46}, {12, 10, 0}, {17, 10, 0}};
    static final String[] BLOCK_NAMES = {"young trees", "after harvest",
                                         "before flowering", "fruit filling"};
    static final double[][] BLOCKS = {{150, 60, 70}, {180, 90, 110},
                                      {120, 90, 120}, {100, 80, 110}};

    /** LU packed in one matrix: L below the diagonal (unit diagonal not
     *  stored), U on and above it; perm[i] = original row now in row i. */
    static double[][] lu;
    static int[] perm;
    static double sign;

    static void luDecompose(double[][] a) {
        int n = a.length;
        lu = new double[n][];
        for (int i = 0; i < n; i++) lu[i] = a[i].clone();
        perm = new int[n];
        for (int i = 0; i < n; i++) perm[i] = i;
        sign = 1.0;
        for (int k = 0; k < n - 1; k++) {
            int p = k;                                   // partial pivot
            for (int i = k + 1; i < n; i++) {
                if (Math.abs(lu[i][k]) > Math.abs(lu[p][k])) p = i;
            }
            if (lu[p][k] == 0.0) throw new ArithmeticException("singular");
            if (p != k) {
                double[] t = lu[k]; lu[k] = lu[p]; lu[p] = t;
                int q = perm[k]; perm[k] = perm[p]; perm[p] = q;
                sign = -sign;
            }
            for (int i = k + 1; i < n; i++) {
                double m = lu[i][k] / lu[k][k];
                lu[i][k] = m;                    // keep the multiplier: it is L
                for (int j = k + 1; j < n; j++) lu[i][j] -= m * lu[k][j];
            }
        }
    }
```

Listing 6.2 (Continued) *LuBlend.java: the same factor-once, solve-many program in Java.*

```
    /** Solve Ly = Pb (forward), then Ux = y (backward). */
    static double[] luSolve(double[] b) {
        int n = lu.length;
        double[] y = new double[n];
        for (int i = 0; i < n; i++) {
            double s = 0;
            for (int j = 0; j < i; j++) s += lu[i][j] * y[j];
            y[i] = b[perm[i]] - s;
        }
        double[] x = new double[n];
        for (int i = n - 1; i >= 0; i--) {
            double s = 0;
            for (int j = i + 1; j < n; j++) s += lu[i][j] * x[j];
            x[i] = (y[i] - s) / lu[i][i];
        }
        return x;
    }

    static void show(String title, double[][] m) {
        System.out.println(title);
        for (double[] row : m) {
            StringBuilder line = new StringBuilder("  ");
            for (double v : row) line.append(f(v, 4, 10));
            System.out.println(line);
        }
    }

    public static void main(String[] args) {
        int n = A.length;
        luDecompose(A);
        double[][] p = new double[n][n], low = new double[n][n],
                   up = new double[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                p[i][j] = j == perm[i] ? 1.0 : 0.0;
                low[i][j] = j < i ? lu[i][j] : (i == j ? 1.0 : 0.0);
                up[i][j] = j >= i ? lu[i][j] : 0.0;
            }
        }
        show("P", p);
        show("L", low);
        show("U", up);
        double err = 0;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
```

Listing 6.2 (Continued) *LuBlend.java: the same factor-once, solve-many program in Java.*

```
                double s = 0;
                for (int k = 0; k < n; k++) s += low[i][k] * up[k][j];
                err = Math.max(err, Math.abs(s - A[perm[i]][j]));
            }
        }
        System.out.println("max |PA - LU| = " + e(err, 1));
        double det = sign;
        for (int i = 0; i < n; i++) det *= lu[i][i];
        System.out.println("det A = " + f(det, 4));
        System.out.println();
        StringBuilder head = new StringBuilder("grams per tree         ");
        for (String s : NAMES) head.append(pad(s, 14));
        System.out.println(head);
        for (int b = 0; b < BLOCKS.length; b++) {
            double[] x = luSolve(BLOCKS[b]);
            StringBuilder line = new StringBuilder(padRight(BLOCK_NAMES[b], 23));
            for (double v : x) line.append(f(100 * v, 1, 14));
            System.out.println(line);
        }
        System.out.println();
        System.out.println("work for 100 right-hand sides"
                           + " (multiplications, leading terms)");
        for (long size : new long[] {3, 100, 1000}) {
            double again = 100 * ((double) (size * size * size) / 3
                                  + size * size);
            double once = (double) (size * size * size) / 3 + 100 * size * size;
            System.out.println("  n = " + pad("" + size, 4)
                    + ": eliminate each time " + f(again, 0, 14)
                    + ", factor once " + f(once, 0, 14));
        }
    }

    // ---- number formatting that matches Python's f"{x:w.df}" and ".de"
    static String f(double x, int d) {
        String s = new BigDecimal(x).setScale(d, RoundingMode.HALF_EVEN)
                                    .toPlainString();
        boolean neg = x < 0 || (x == 0 && 1 / x < 0);
        return neg && !s.startsWith("-") ? "-" + s : s;
    }

    static String f(double x, int d, int w) { return pad(f(x, d), w); }
```

Listing 6.2 (Continued) *LuBlend.java: the same factor-once, solve-many program in Java.*

```
    static String e(double x, int d) {
        if (x == 0) return "0." + "0".repeat(d) + "e+00";
        BigDecimal b = new BigDecimal(x).round(
                new MathContext(d + 1, RoundingMode.HALF_EVEN));
        int exp = b.precision() - b.scale() - 1;
        String digits = b.unscaledValue().abs().toString();
        digits = (digits + "0".repeat(d + 1)).substring(0, d + 1);
        String ex = (exp < 0 ? "-" : "+") + (Math.abs(exp) < 10 ? "0" : "")
                  + Math.abs(exp);
        return (x < 0 ? "-" : "") + digits.charAt(0) + "." + digits.substring(1)
               + "e" + ex;
    }

    static String pad(String s, int w) {
        return " ".repeat(Math.max(0, w - s.length())) + s;
    }

    static String padRight(String s, int w) {
        return s + " ".repeat(Math.max(0, w - s.length()));
    }
}
```

6.6.2 Cholesky and QR in Python and Java

Listing 6.3 has one function per formula: `normal_equations` builds $A^{\mathsf{T}}A$ and $A^{\mathsf{T}}b$, `cholesky` is Equation (6.10) and raises an error when a pivot is not positive, `cholesky_solve` does the two triangles, and `householder_lstsq` applies the reflections Equation (6.13) column by column to a copy of A and to b, then back-substitutes and returns $\| c_2 \|$ from Equation (6.12). The function `vandermonde` builds the columns $1, t, t^2, \ldots, t^d$ at 20 equally spaced points, the columns of polynomial fitting, with powers formed by repeated multiplication so that both languages compute identical numbers.

Listing 6.3 `least_blend.py`: least squares by the normal equations with Cholesky and by Householder QR, for the orchard and for polynomial columns.

```
"""Four nutrient targets, three fertilizers: the best blend in the least
squares sense, by normal equations + Cholesky and by Householder QR."""
import math

NUTRIENTS = ["N", "P2O5", "K2O", "MgO"]
# grams of each nutrient in 100 g of NPK 12-12-17-2, NPK 20-10-10, urea
A = [[12.0, 20.0, 46.0],
     [12.0, 10.0, 0.0],
     [17.0, 10.0, 0.0],
     [2.0, 0.0, 0.0]]
B = [150.0, 60.0, 70.0, 6.0]      # grams per young tree



def total(values):
    """Plain left-to-right sum. Python 3.12+ sum() compensates rounding,
    which would make our digits differ from the Java version."""
    s = 0.0
    for v in values:
        s += v
    return s

def normal_equations(a, b):
    """Return A^T A and A^T b."""
    m, n = len(a), len(a[0])
    ata = [[total(a[k][i] * a[k][j] for k in range(m)) for j in range(n)]
           for i in range(n)]
    atb = [total(a[k][i] * b[k] for k in range(m)) for i in range(n)]
    return ata, atb


def cholesky(s):
    """Lower triangular L with L L^T = s; s must be symmetric positive
    definite, otherwise a pivot is not positive and we stop."""
    n = len(s)
    low = [[0.0] * n for _ in range(n)]
    for j in range(n):
        d = s[j][j] - total(low[j][k] * low[j][k] for k in range(j))
        if d <= 0.0:
            raise ValueError(f"pivot {j + 1} is {d:.3e}: not positive")
        low[j][j] = math.sqrt(d)
        for i in range(j + 1, n):
            t = s[i][j] - total(low[i][k] * low[j][k] for k in range(j))
            low[i][j] = t / low[j][j]
    return low
```

Listing 6.3 (Continued) *least_blend.py: least squares by the normal equations with Cholesky and by Householder QR, for the orchard and for polynomial columns.*

```
def cholesky_solve(low, b):
    """Solve L y = b, then L^T x = y."""
    n = len(low)
    y = [0.0] * n
    for i in range(n):
        y[i] = (b[i] - total(low[i][k] * y[k] for k in range(i))) / low[i][i]
    x = [0.0] * n
    for i in range(n - 1, -1, -1):
        s = total(low[k][i] * x[k] for k in range(i + 1, n))
        x[i] = (y[i] - s) / low[i][i]
    return x


def householder_lstsq(a, b):
    """Least squares by Householder QR: returns x and the residual norm."""
    m, n = len(a), len(a[0])
    r = [row[:] for row in a]
    c = b[:]                                    # becomes Q^T b
    for k in range(n):
        norm = math.sqrt(total(r[i][k] * r[i][k] for i in range(k, m)))
        alpha = -norm if r[k][k] >= 0.0 else norm   # avoid cancellation
        v = [0.0] * m
        for i in range(k, m):
            v[i] = r[i][k]
        v[k] -= alpha
        vv = total(v[i] * v[i] for i in range(k, m))
        if vv == 0.0:
            continue
        for j in range(k, n):
            f = 2.0 * total(v[i] * r[i][j] for i in range(k, m)) / vv
            for i in range(k, m):
                r[i][j] -= f * v[i]
        f = 2.0 * total(v[i] * c[i] for i in range(k, m)) / vv
        for i in range(k, m):
            c[i] -= f * v[i]
    x = [0.0] * n
    for i in range(n - 1, -1, -1):
        s = total(r[i][j] * x[j] for j in range(i + 1, n))
        x[i] = (c[i] - s) / r[i][i]
    rest = math.sqrt(total(c[i] * c[i] for i in range(n, m)))
    return x, rest
```

Listing 6.3 (Continued) *least_blend.py: least squares by the normal equations with Cholesky and by Householder QR, for the orchard and for polynomial columns.*

```
def vandermonde(m, degree):
    """m equally spaced points t in [0, 1], columns 1, t, ..., t^degree."""
    rows = []
    for i in range(m):
        t = i / (m - 1)
        row = [1.0]
        for _ in range(degree):
            row.append(row[-1] * t)          # powers by repeated products
        rows.append(row)
    return rows


def main():
    ata, atb = normal_equations(A, B)
    print("A^T A and A^T b")
    for row, v in zip(ata, atb):
        print("  " + "".join(f"{x:9.1f}" for x in row) + f"   | {v:9.1f}")
    low = cholesky(ata)
    print("Cholesky factor L")
    for row in low:
        print("  " + "".join(f"{x:9.4f}" for x in row))
    x1 = cholesky_solve(low, atb)
    x2, rest = householder_lstsq(A, B)
    print("grams per tree       normal+Cholesky  Householder QR")
    for name, u, v in zip(["NPK 12-12-17-2", "NPK 20-10-10", "Urea"], x1, x2):
        print(f"  {name:18s}{100 * u:15.3f}{100 * v:16.3f}")
    print("nutrient  target  delivered  miss")
    for i, name in enumerate(NUTRIENTS):
        got = total(A[i][j] * x2[j] for j in range(3))
        print(f"  {name:6s}{B[i]:8.1f}{got:11.2f}{got - B[i]:+7.2f}")
    print(f"residual norm {rest:.4f}")
    print()
    print("Polynomial columns t^0..t^d at 20 points, true x = all ones")
    print(" d   normal+Cholesky error   Householder QR error")
    for d in (3, 6, 9, 12):
        a = vandermonde(20, d)
        b = [total(row) for row in a]
        xq, _ = householder_lstsq(a, b)
        eq = max(abs(v - 1.0) for v in xq)
        try:
            s, sb = normal_equations(a, b)
            xn = cholesky_solve(cholesky(s), sb)
            en = f"{max(abs(v - 1.0) for v in xn):22.2e}"
        except ValueError:
            en = f"{'fails (pivot <= 0)':>22s}"
        print(f"{d:2d}{en}{eq:23.2e}")
```

Listing 6.3 (Continued) *least_blend.py: least squares by the normal equations with Cholesky and by Householder QR, for the orchard and for polynomial columns.*

```
if __name__ == "__main__":
    main()
```

Listing 6.4 is the Java twin. A Java method returns one value, so `householderLstsq` returns x and leaves the residual norm $\| c_2 \|$ in the static field `lastRest`, and `normalEquations` writes $A^{\mathsf{T}}b$ into an array passed by the caller. The Cholesky failure is signaled with an `ArithmeticException`, which `main` catches to print the same message as the Python version.

Listing 6.4 `LeastBlend.java`: the same two least squares methods in Java.

```
import java.math.BigDecimal;
import java.math.MathContext;
import java.math.RoundingMode;

/** Four nutrient targets, three fertilizers: the best blend in the least
 *  squares sense, by normal equations + Cholesky and by Householder QR. */
public class LeastBlend {
    static final String[] NUTRIENTS = {"N", "P2O5", "K2O", "MgO"};
    // grams of each nutrient in 100 g of NPK 12-12-17-2, NPK 20-10-10, urea
    static final double[][] A = {{12, 20, 46}, {12, 10, 0}, {17, 10, 0},
                                 {2, 0, 0}};
    static final double[] B = {150, 60, 70, 6};     // grams per young tree

    /** A^T A (returned) and A^T b (written into atb). */
    static double[][] normalEquations(double[][] a, double[] b, double[] atb) {
        int m = a.length, n = a[0].length;
        double[][] ata = new double[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                double s = 0;
                for (int k = 0; k < m; k++) s += a[k][i] * a[k][j];
                ata[i][j] = s;
            }
            double s = 0;
            for (int k = 0; k < m; k++) s += a[k][i] * b[k];
            atb[i] = s;
        }
        return ata;
    }

    /** Lower triangular L with L L^T = s; s must be symmetric positive
     *  definite, otherwise a pivot is not positive and we stop. */
    static double[][] cholesky(double[][] s) {
        int n = s.length;
        double[][] low = new double[n][n];
        for (int j = 0; j < n; j++) {
            double sum = 0;
            for (int k = 0; k < j; k++) sum += low[j][k] * low[j][k];
            double d = s[j][j] - sum;
            if (d <= 0.0) {
                throw new ArithmeticException("pivot " + (j + 1)
                                              + " is not positive");
            }
            low[j][j] = Math.sqrt(d);
            for (int i = j + 1; i < n; i++) {
                double t = 0;
```

Listing 6.4 (Continued) *LeastBlend.java: the same two least squares methods in Java.*

```
                for (int k = 0; k < j; k++) t += low[i][k] * low[j][k];
                low[i][j] = (s[i][j] - t) / low[j][j];
            }
        }
        return low;
    }

    /** Solve L y = b, then L^T x = y. */
    static double[] choleskySolve(double[][] low, double[] b) {
        int n = low.length;
        double[] y = new double[n];
        for (int i = 0; i < n; i++) {
            double s = 0;
            for (int k = 0; k < i; k++) s += low[i][k] * y[k];
            y[i] = (b[i] - s) / low[i][i];
        }
        double[] x = new double[n];
        for (int i = n - 1; i >= 0; i--) {
            double s = 0;
            for (int k = i + 1; k < n; k++) s += low[k][i] * x[k];
            x[i] = (y[i] - s) / low[i][i];
        }
        return x;
    }

    /** Least squares by Householder QR: returns x; the residual norm is
     *  left in lastRest. */
    static double lastRest;

    static double[] householderLstsq(double[][] a, double[] b) {
        int m = a.length, n = a[0].length;
        double[][] r = new double[m][];
        for (int i = 0; i < m; i++) r[i] = a[i].clone();
        double[] c = b.clone();                    // becomes Q^T b
        for (int k = 0; k < n; k++) {
            double ss = 0;
            for (int i = k; i < m; i++) ss += r[i][k] * r[i][k];
            double norm = Math.sqrt(ss);
            double alpha = r[k][k] >= 0.0 ? -norm : norm; // avoid cancellation
            double[] v = new double[m];
            for (int i = k; i < m; i++) v[i] = r[i][k];
            v[k] -= alpha;
            double vv = 0;
            for (int i = k; i < m; i++) vv += v[i] * v[i];
            if (vv == 0.0) continue;
            for (int j = k; j < n; j++) {
```

Listing 6.4 (Continued) *LeastBlend.java: the same two least squares methods in Java.*

```
                double s = 0;
                for (int i = k; i < m; i++) s += v[i] * r[i][j];
                double f = 2.0 * s / vv;
                for (int i = k; i < m; i++) r[i][j] -= f * v[i];
            }
            double s = 0;
            for (int i = k; i < m; i++) s += v[i] * c[i];
            double f = 2.0 * s / vv;
            for (int i = k; i < m; i++) c[i] -= f * v[i];
        }
        double[] x = new double[n];
        for (int i = n - 1; i >= 0; i--) {
            double s = 0;
            for (int j = i + 1; j < n; j++) s += r[i][j] * x[j];
            x[i] = (c[i] - s) / r[i][i];
        }
        double rest = 0;
        for (int i = n; i < m; i++) rest += c[i] * c[i];
        lastRest = Math.sqrt(rest);
        return x;
    }

    /** m equally spaced points t in [0, 1], columns 1, t, ..., t^degree. */
    static double[][] vandermonde(int m, int degree) {
        double[][] rows = new double[m][degree + 1];
        for (int i = 0; i < m; i++) {
            double t = (double) i / (m - 1);
            rows[i][0] = 1.0;
            for (int j = 1; j <= degree; j++) {
                rows[i][j] = rows[i][j - 1] * t;  // powers by repeated products
            }
        }
        return rows;
    }

    public static void main(String[] args) {
        double[] atb = new double[3];
        double[][] ata = normalEquations(A, B, atb);
        System.out.println("A^T A and A^T b");
        for (int i = 0; i < 3; i++) {
            StringBuilder line = new StringBuilder("  ");
            for (double v : ata[i]) line.append(f(v, 1, 9));
            System.out.println(line + "   | " + f(atb[i], 1, 9));
        }
        double[][] low = cholesky(ata);
        System.out.println("Cholesky factor L");
```

Listing 6.4 (Continued) *LeastBlend.java: the same two least squares methods in Java.*

```
        for (double[] row : low) {
            StringBuilder line = new StringBuilder("  ");
            for (double v : row) line.append(f(v, 4, 9));
            System.out.println(line);
        }
        double[] x1 = choleskySolve(low, atb);
        double[] x2 = householderLstsq(A, B);
        double rest = lastRest;
        System.out.println("grams per tree       normal+Cholesky  Householder QR");
        String[] names = {"NPK 12-12-17-2", "NPK 20-10-10", "Urea"};
        for (int j = 0; j < 3; j++) {
            System.out.println("  " + padRight(names[j], 18) + f(100 * x1[j], 3, 15)
                               + f(100 * x2[j], 3, 16));
        }
        System.out.println("nutrient  target  delivered  miss");
        for (int i = 0; i < 4; i++) {
            double got = 0;
            for (int j = 0; j < 3; j++) got += A[i][j] * x2[j];
            String miss = f(got - B[i], 2);
            if (!miss.startsWith("-")) miss = "+" + miss;
            System.out.println("  " + padRight(NUTRIENTS[i], 6) + f(B[i], 1, 8)
                               + f(got, 2, 11) + pad(miss, 7));
        }
        System.out.println("residual norm " + f(rest, 4));
        System.out.println();
        System.out.println("Polynomial columns t^0..t^d at 20 points,"
                           + " true x = all ones");
        System.out.println(" d   normal+Cholesky error   Householder QR error");
        for (int d : new int[] {3, 6, 9, 12}) {
            double[][] a = vandermonde(20, d);
            double[] b = new double[20];
            for (int i = 0; i < 20; i++) {
                double s = 0;
                for (double v : a[i]) s += v;
                b[i] = s;
            }
            double eq = maxError(householderLstsq(a, b));
            String en;
            try {
                double[] sb = new double[d + 1];
                double[][] s = normalEquations(a, b, sb);
                en = pad(e(maxError(choleskySolve(cholesky(s), sb)), 2), 22);
            } catch (ArithmeticException err) {
                en = pad("fails (pivot <= 0)", 22);
            }
            System.out.println(pad("" + d, 2) + en + pad(e(eq, 2), 23));
```

Listing 6.4 (Continued) *LeastBlend.java: the same two least squares methods in Java.*

```
        }
    }

    static double maxError(double[] x) {
        double worst = 0;
        for (double v : x) worst = Math.max(worst, Math.abs(v - 1.0));
        return worst;
    }

    // ---- number formatting that matches Python's f"{x:w.df}" and ".de"
    static String f(double x, int d) {
        String s = new BigDecimal(x).setScale(d, RoundingMode.HALF_EVEN)
                                    .toPlainString();
        boolean neg = x < 0 || (x == 0 && 1 / x < 0);
        return neg && !s.startsWith("-") ? "-" + s : s;
    }

    static String f(double x, int d, int w) { return pad(f(x, d), w); }

    static String e(double x, int d) {
        if (x == 0) return "0." + "0".repeat(d) + "e+00";
        BigDecimal b = new BigDecimal(x).round(
                new MathContext(d + 1, RoundingMode.HALF_EVEN));
        int exp = b.precision() - b.scale() - 1;
        String digits = b.unscaledValue().abs().toString();
        digits = (digits + "0".repeat(d + 1)).substring(0, d + 1);
        String ex = (exp < 0 ? "-" : "+") + (Math.abs(exp) < 10 ? "0" : "")
                    + Math.abs(exp);
        return (x < 0 ? "-" : "") + digits.charAt(0) + "." + digits.substring(1)
               + "e" + ex;
    }

    static String pad(String s, int w) {
        return " ".repeat(Math.max(0, w - s.length())) + s;
    }

    static String padRight(String s, int w) {
        return s + " ".repeat(Math.max(0, w - s.length()));
    }
}
```

6.7 Running It

Compile the Java files with `javac -d out *.java` in the `java` folder, then run `java -cp out LuBlend` and `java -cp out LeastBlend`; in the `python` folder run `python3 lu_blend.py` and `python3 least_blend.py`. Both languages print exactly the same text. The LU program first:

```
P
      0.0000    0.0000    1.0000
      1.0000    0.0000    0.0000
      0.0000    1.0000    0.0000
L
      1.0000    0.0000    0.0000
      0.7059    1.0000    0.0000
      0.7059    0.2273    1.0000
U
     17.0000   10.0000    0.0000
      0.0000   12.9412   46.0000
      0.0000    0.0000  -10.4545
max |PA - LU| = 0.0e+00
det A = -2300.0000

grams per tree           NPK 12-12-17  NPK 20-10-10   Urea 46-0-0
young trees                     200.0         360.0         117.4
after harvest                   400.0         420.0         104.3
before flowering                600.0         180.0          26.1
fruit filling                   600.0          80.0          26.1

work for 100 right-hand sides (multiplications, leading terms)
  n =    3: eliminate each time           1800, factor once            909
  n =  100: eliminate each time       34333333, factor once        1333333
  n = 1000: eliminate each time    33433333333, factor once      433333333
```

P, L and U are the ones we found by hand in Section 6.2.2, and $PA - LU$ is exactly zero. The determinant is -2300. The table of blends has the young trees' 200, 360 and 117.4 g from Section 6.2.3 and three more blocks for the price of six substitutions. Notice the "before flowering" and "fruit filling" rows: they need the same amount of urea (26.1 g) and the same NPK 12-12-17 (600 g), and differ only in NPK 20-10-10. Their recommendations differ by (20,10,10), which is exactly one unit of NPK 20-10-10, so the difference of the blends is 100 g of that bag. Linear systems reward this kind of reading. The work table shows the same idea at scale: for 100 right-hand sides and $n = 1000$, 33 billion multiplications against 0.43 billion.

The least squares program:

```
A^T A and A^T b
      581.0    530.0    552.0   |    3722.0
      530.0    600.0    920.0   |    4300.0
      552.0    920.0   2116.0   |    6900.0
Cholesky factor L
    24.1039   0.0000   0.0000
    21.9881  10.7946   0.0000
    22.9008  38.5799  10.1559
grams per tree       normal+Cholesky  Householder QR
  NPK 12-12-17-2            224.242         224.242
  NPK 20-10-10              324.848         324.848
  Urea                      126.350         126.350
nutrient  target  delivered  miss
  N        150.0     150.00  +0.00
  P2O5      60.0      59.39  -0.61
  K2O       70.0      70.61  +0.61
  MgO        6.0       4.48  -1.52
residual norm 1.7408

Polynomial columns t^0..t^d at 20 points, true x = all ones
 d   normal+Cholesky error   Householder QR error
 3             8.14e-13              7.77e-16
 6             4.14e-09              8.35e-13
 9             1.45e-03              4.05e-10
12    fails (pivot <= 0)              2.81e-08
```

The $A^{\mathsf{T}}A$ and L printed are the ones computed by hand in Section 6.4. Both methods give the same blend to the three decimals shown: 224.242 g of NPK 12-12-17-2, 324.848 g of NPK 20-10-10 and 126.350 g of urea. The table of misses repays attention. Nitrogen is met exactly, because urea contains nothing but nitrogen: whatever the other two bags do, some amount of urea fixes the N row without disturbing the others, so the best compromise never misses there. Phosphate and potash are missed by 0.61 g each, in opposite directions, and magnesium by 1.52 g: the blend delivers 4.48 g instead of 6. Least squares has spread the unavoidable miss where it costs least in squared grams. Whether 1.5 g of magnesium matters more than 0.6 g of phosphate is an agronomic question, not a numerical one; Exercise 4 shows how to tell the program by weighting the rows.

The second table is the accuracy lesson. The exact solution of every polynomial problem is known (all ones), so the errors are real errors. At degree 3 both methods are excellent. At degree 6 the normal equations have lost three more digits than QR; at degree 9 they are wrong in the third decimal (1.45×10^{-3}) while QR still has ten correct digits (4.05×10^{-10}); at degree 12 rounding has made $A^{\mathsf{T}}A$ numerically indefinite, a Cholesky pivot comes out negative, and the normal equations give no answer at all, while QR still delivers eight digits. **Figure 6.4** shows the whole range of degrees. The comparison script of the next section measures $\kappa(A) = 3.79 \times 10^{6}$ at degree 9, so Equation (6.11) predicts an error around $\kappa(A)^2 \varepsilon_{\text{mach}} = 1.44 \times 10^{13} \times 1.1 \times 10^{-16} \approx 1.6 \times 10^{-3}$ for the normal equations and $\kappa(A)\,\varepsilon_{\text{mach}} \approx 4 \times 10^{-10}$ for QR. Both predictions hit the measured errors.

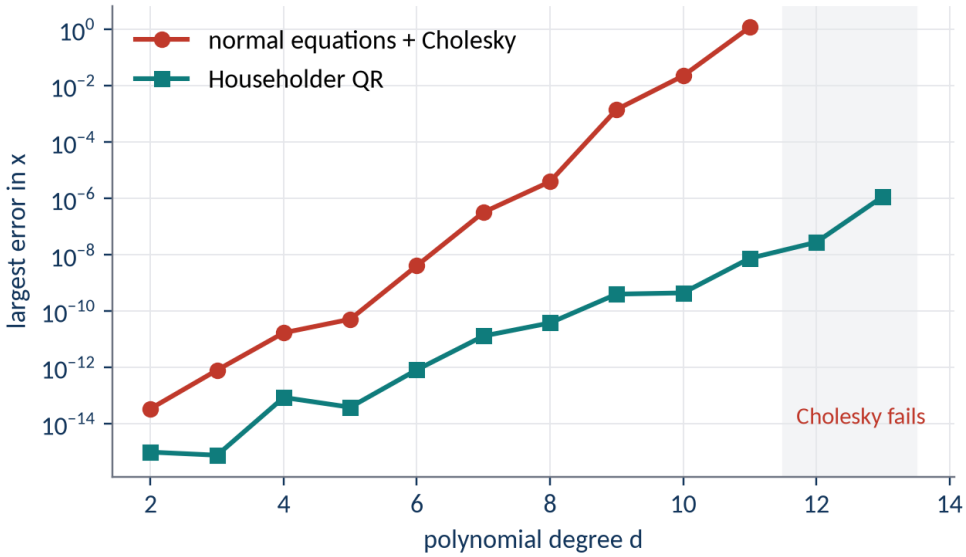


Figure 6.4 Largest error of the least squares solution for polynomial columns of degree d at 20 points: the normal equations lose about twice as many digits as Householder QR, and from degree 12 Cholesky fails.

6.8 Checking Against the Library

SciPy 1.18, the current release in September 2026, rewrote `scipy.linalg.lu` and `det` in C++ with support for batches of matrices and sped up `cholesky` [8], [9]; the interfaces we use are unchanged. **Listing 6.5** asks SciPy for the same factorizations. Note one difference of convention: `scipy.linalg.lu` returns $A = PLU$, with the permutation on the other side, so its P is the transpose of ours [10].

Listing 6.5 `compare_scipy.py`: our factors and solutions compared with SciPy's `lu`, `lu_factor`/`lu_solve`, `cho_factor`/`cho_solve` and NumPy's `lstsq` and `qr`.

```
"""Check the blends against SciPy's LU, Cholesky and NumPy's least squares."""
import numpy as np
import scipy
from scipy.linalg import cho_factor, cho_solve, lu, lu_factor, lu_solve

import least_blend
import lu_blend
from lu_blend import A, BLOCKS


def main():
    print(f"NumPy {np.__version__}, SciPy {scipy.__version__}")
    a = np.array(A)
    p, low, up = lu(a)                    # SciPy returns A = P L U
    print("scipy.linalg.lu: U diagonal", np.round(np.diag(up), 4))
    factor = lu_factor(a)
    ours, perm, _ = lu_blend.lu_decompose(A)
    worst = 0.0
    for _, b in BLOCKS:
        x = lu_solve(factor, np.array(b))
        mine = lu_blend.lu_solve(ours, perm, b)
        worst = max(worst, np.max(np.abs(x - np.array(mine))))
    print(f"lu_solve vs ours, all four blocks: max difference {worst:.1e}")

    m = np.array(least_blend.A)
    b = np.array(least_blend.B)
    x, res, rank, sv = np.linalg.lstsq(m, b)
    print("numpy.linalg.lstsq grams:", np.round(100 * x, 3),
          f" residual norm {np.sqrt(res[0]):.4f}")
    c = cho_factor(m.T @ m, lower=True)
    print("cho_solve grams:         ", np.round(100 * cho_solve(c, m.T @ b), 3))
    q, r = np.linalg.qr(m)
    print("numpy.linalg.qr R diagonal:", np.round(np.diag(r), 4))

    v = np.vander(np.linspace(0.0, 1.0, 20), 10, increasing=True)
    print(f"degree 9 columns: cond(A) = {np.linalg.cond(v):.2e},"
          f" cond(A^T A) = {np.linalg.cond(v.T @ v):.2e}")


if __name__ == "__main__":
    main()
```

```
NumPy 2.5.3, SciPy 1.18.1
scipy.linalg.lu: U diagonal [ 17.      12.9412 -10.4545]
lu_solve vs ours, all four blocks: max difference 6.2e-15
numpy.linalg.lstsq grams: [224.242 324.848 126.35 ]  residual norm 1.7408
cho_solve grams:          [224.242 324.848 126.35 ]
numpy.linalg.qr R diagonal: [-24.1039  10.7946 -10.1559]
degree 9 columns: cond(A) = 3.79e+06, cond(A^T A) = 1.44e+13
```

The diagonal of U is the same, all four blocks agree with `lu_solve` to 6×10^{-15}, and `numpy.linalg.lstsq` (which uses the singular value decomposition, a close relative of QR) gives the same blend and the same residual norm 1.7408. NumPy's R has the same diagonal as our Cholesky factor up to signs, as Section 6.5.4 predicted, and the condition numbers at degree 9, 3.79×10^{6} and 1.44×10^{13}, confirm Equation (6.11). In practice, call these library routines: they use blocked algorithms that keep the processor's cache busy and run hundreds of times faster than our loops on large matrices [3]. Our programs exist so that you know what the library does and can tell when its answer deserves trust.

6.9 Making It Better

- **Negative grams.** Nothing in LU or QR knows that fertilizer cannot be negative. The four blocks of this season happen to give positive blends, but a recommendation of (150, 100, 120) gives -4.35 g of urea. The right tool is *nonnegative least squares*, which finds the best blend among those with $x \geq 0$; SciPy has it as `scipy.optimize.nnls`, based on the algorithm of Lawson and Hanson [11]. Chapter 13 treats constrained optimization more generally.
- **Tridiagonal and banded matrices.** When only a few diagonals are nonzero, LU fills in nothing outside the band, and the cost falls from $n^3/3$ to about nb^2 for bandwidth b. The tridiagonal case, with $b = 1$, is the *Thomas algorithm*, and Chapter 7 uses it for splines at a cost of $O(n)$.
- **Iterative refinement.** After solving $Ax = b$, compute the residual $r = b - Ax$ (in higher precision if possible), solve $A\,d = r$ with the same factors, which is cheap, and correct $x \leftarrow x + d$. One or two rounds recover most of the digits that rounding took, because the factors are reused [4].
- **Sparse LU.** For large sparse matrices, such as the griddle of Chapter 5, the order in which unknowns are eliminated decides how many zeros fill in. Sparse direct solvers, such as SuperLU behind SciPy's `spsolve`, first reorder the unknowns to keep the fill-in small and only then factor [3].
- **Updating instead of refactoring.** When one fertilizer is replaced by another, only one column of A changes. There are formulas that update LU or QR for such a change in $O(n^2)$ instead of refactoring in $O(n^3)$ [3].

6.10 Using It for Real

To use `lu_blend.py` for your own orchard, rice field or vegetable garden, change three things.

1. **The grades.** Copy the grade from each bag into a column of `A`: the percentages of N, P_2O_5 and K_2O, which are also the grams per 100 g. If you add a fourth nutrient (MgO, sulfur, calcium), add a row, and switch to `least_blend.py` unless you also add a fourth fertilizer.
2. **The recommendations.** Put each block's recommendation, in grams per tree (or per square meter), into `BLOCKS`. Get it from the extension officer or a soil laboratory; a program cannot see a yellow leaf.
3. **Check the answer before spreading.** A negative amount means the recommendation cannot be met with these bags; the determinant close to zero, or a warning from `lu_decompose`, means two of your bags are nearly the same mix and the blend is fragile. Round to what you can weigh (a 10-gram scoop), and recompute the nutrients delivered by the rounded amounts, as the "delivered" column of `least_blend.py` does.

Here is what step 3 looks like for the young trees. The exact blend is 200 g, 360 g and 117.4 g. With a 10-gram scoop, Pak Harun's helper will measure 200 g, 360 g and 120 g. The nutrients delivered by the rounded amounts are N: $12 \times 2 + 20 \times 3.6 + 46 \times 1.2 = 24 + 72 + 55.2 = 151.2$ g, phosphate $12 \times 2 + 10 \times 3.6 = 60$ g and potash $17 \times 2 + 10 \times 3.6 = 70$ g. Rounding the urea has added 1.2 g of nitrogen, less than 1%, which no tree will notice. For a block of 120 young trees the shopping list is then 120×200 g $= 24$ kg of NPK 12-12-17, 43.2 kg of NPK 20-10-10 and 14.4 kg of urea: in 50 kg bags, one bag of each, with most of the urea bag left over for the next season. Doing the arithmetic in this order, exact blend, then rounding, then checking what the rounded blend delivers, catches the cases where rounding matters, such as a blend with only a few grams of one fertilizer.

The same program answers questions in any shop that mixes: a feed mill combining maize, rice bran and soybean meal to meet protein, energy and fiber targets; a small bakery that mixes flours to reach a protein content; a laundry that dilutes two detergent concentrates to a target strength. Whenever the recipe book (the matrix) stays and the order (the right-hand side) changes, factor once.

6.11 Numerical Methods in Practice: Profitina

Profitina's AI Visibility Engine turns the signals it measures on a website, such as how extractable the content is, what structured data the pages carry and how technically sound the site is, into a readiness score, and it explains each signal on its own. Turning many measurements into one number is a weighted aggregation, and the weights must be chosen so that the score agrees with what is observed: a site that answer engines cite often should score high. When there are more observed sites than weights, which is always the case, choosing the weights is an overdetermined least squares problem of exactly the form Equation (6.8), one row per observed site and one column per signal. The lessons of this chapter apply without change. Signals that measure nearly the same thing make the columns nearly dependent, $\kappa(A)$ large and $\kappa(A^{\mathsf{T}}A)$ enormous; a method that forms the normal equations can then produce weights that are pure rounding noise, and a QR-based solver, which libraries use by default, is the safe road. And when the same signal model is evaluated for many brands at every scheduled re-scan, the expensive factorization belongs outside the loop and only the cheap substitutions inside it, as in **Figure 6.1**.

Profitina describes what its score measures and why, not the proprietary details of how it is weighted, and it presents the score as advice: the brand owner decides what to change. The numerical care described here is what makes such advice stable from one week to the next instead of jumping because of rounding.

6.12 Common Mistakes, and How to Avoid Them

- **Computing the inverse to solve a system.** `x = inv(A) @ b` costs three times as much as `solve` and is less accurate. Factor and substitute; form A^{-1} only when its entries themselves are wanted.
- **Forgetting the permutation.** With pivoting the factors satisfy $PA = LU$, not $A = LU$. Solving $Ly = b$ instead of $Ly = Pb$ gives a wrong answer with no warning. Different libraries put P on different sides (SciPy returns $A = PLU$); read the documentation.
- **Refactoring for every right-hand side.** If the matrix has not changed, neither have its factors. Keep them.
- **Using Cholesky on a matrix that is not SPD.** A symmetric matrix can be indefinite; Cholesky will then hit a negative number under the square root. That failure is a diagnosis, not a bug; do not "fix" it by taking absolute values.
- **Solving least squares with the normal equations by habit.** They square the condition number (Equation (6.11)). For well-conditioned problems they are fine and fast; for polynomial fits, nearly dependent columns or data with very different scales, use QR (`numpy.linalg.lstsq` or `scipy.linalg.lstsq`).
- **Reading a least squares answer as exact.** The blend of Section 6.7 misses magnesium by 25%. Always print what the answer actually delivers next to what was asked, and decide whether the miss is acceptable.

- **Reading the determinant as a measure of trouble.** A tiny determinant does not mean a matrix is nearly singular: multiplying every entry of a 10×10 identity matrix by 0.1 gives determinant 10^{-10} and a perfectly conditioned matrix. Measuring the fertilizers in kilograms instead of hundreds of grams would shrink $\det A$ by a factor of 10^3 without changing anything real. Use the condition number, which does not depend on units, to judge whether a system is fragile.
- **Mixing units in the rows.** If one row is in grams and another in kilograms, least squares treats a 1-unit miss in both as equally bad. Put all rows in comparable units, or weight them on purpose (Exercise 4).

6.13 Chapter Summary

- Gaussian elimination with partial pivoting, with its multipliers recorded, is the decomposition $PA = LU$ (Equation (6.2), **Pseudocode 6.1**); the entries follow directly from the Doolittle formulas (Equation (6.4)).
- Factoring costs $n^3/3$ multiplications; each right-hand side then costs two triangular substitutions (Equation (6.3)), about n^2 (Equation (6.5), **Figure 6.2**). Four blocks of the orchard (Equation (6.1)) share one factorization (**Figure 6.1**).
- The determinant is the signed product of U's diagonal (Equation (6.6)); the inverse is n solves, and is rarely needed.
- Pivoting is not optional: without it a tiny pivot produces a huge multiplier that wipes out the information in the other rows (Section 6.3.5). Cholesky needs no pivoting on SPD matrices, because its entries are bounded by the square roots of the diagonal (Section 6.4.4).
- More requirements than unknowns (Equation (6.7)) lead to least squares (Equation (6.8)). The best x makes the residual perpendicular to the columns (**Figure 6.3**), which gives the normal equations (Equation (6.9)).
- $A^{\mathsf{T}}A$ is symmetric positive definite and can be factored by Cholesky (Equation (6.10)) at half the cost of LU; a non-positive pivot proves the matrix is not SPD.
- The normal equations square the condition number (Equation (6.11)). Householder QR (Equation (6.13)) reduces least squares to one triangular solve (Equation (6.12)) without squaring; at degree 9 it keeps ten digits where the normal equations keep three, and at degree 12 it works where Cholesky fails (**Figure 6.4**). **Table 6.2** summarizes the three decompositions.
- The Java and Python programs agree byte for byte, and SciPy and NumPy reproduce every number.

6.14 Exercises

1. Using the factors P, L and U of Section 6.2.2, solve by hand for the "before flowering" block, $b = (120,90,120)$. Check your answer against the program.
2. A new farm shop sells only NPK 12-12-17 in two bag sizes and urea, so Pak Harun's grade matrix becomes $\begin{pmatrix} 12 & 12 & 46 \\ 12 & 12 & 0 \\ 17 & 17 & 0 \end{pmatrix}$. At which step does **Pseudocode 6.1** stop, and why? What does its determinant tell you about the blends?
3. Compute A^{-1} for the grade matrix of Equation (6.1) with the LU factors (three solves). Interpret its second column: how would you give a tree one extra gram of phosphate and nothing else, and why is it impossible in practice?

4. In the four-nutrient problem, a miss of 1.5 g of MgO is 25% of the target, while a miss of 0.6 g of phosphate is 1%. Divide each row of A and b by that row's target, so that the misses become relative, and solve the weighted problem. How do the blend and the relative misses change?
5. Find the Cholesky factor of $S = \begin{pmatrix} 4 & 2 & 2 \\ 2 & 5 & 3 \\ 2 & 3 & 6 \end{pmatrix}$ by hand. Then try the same formulas on $\begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$ and explain what happens.
6. Construct the Householder reflection H that maps $a = (3,4)$ onto a multiple of e_1, using the sign rule of Section 6.5.3. Write H as a 2×2 matrix, check that $H^{\mathsf{T}}H = I$ and compute Ha.
7. A laptop does about 10^9 multiplications per second in a simple loop. Estimate the time to solve 100 systems with the same 1000×1000 matrix by elimination each time and by LU once, using the counts printed by `lu_blend`.

6.15 Solutions to the Exercises

1. Pb reorders b as K, N, P: (120,120,90). Forward: $y_1 = 120$; $y_2 = 120 - 0.7059 \times 120 = 120 - 84.7059 = 35.2941$; $y_3 = 90 - 0.7059 \times 120 - 0.2273 \times 35.2941 = 90 - 84.7059 - 8.0214 = -2.7273$. Backward: $x_3 = -2.7273/-10.4545 = 0.2609$; $x_2 = (35.2941 - 46 \times 0.2609)/12.9412 = (35.2941 - 12.0000)/12.9412 = 1.8$; $x_1 = (120 - 10 \times 1.8)/17 = 102/17 = 6$. So 600 g of NPK 12-12-17, 180 g of NPK 20-10-10 and 26.1 g of urea, as the program prints.

2. Step 1 picks the pivot 17 (the K row) and eliminates: both remaining rows get $12 - (12/17) \times 17 = 0$ in column 2, because columns 1 and 2 are identical. Step 2 then finds only zeros in column 2 on and below the diagonal, and line 4 stops with "singular" (the Python program raises `ValueError: matrix is singular`). The determinant is 0, because two columns are equal. It means the recommendation cannot pin down the blend: if some blend works, then moving grams from one bag size to the other gives another blend that works equally well, and if the recommendation is not in the column space, no blend works at all. Two bags with the same grade are one fertilizer, not two.

3. Solving $Ax = e_1$, e_2 and e_3 with the factors gives (in hundreds of grams)

$$A^{-1} = \begin{pmatrix} 0 & -0.2 & 0.2 \\ 0 & 0.34 & -0.24 \\ 0.02174 & -0.09565 & 0.05217 \end{pmatrix}.$$

Column 1 says that one extra gram of nitrogen alone costs 2.17 g of urea (0.02174 hundreds of grams), naturally, since urea is pure nitrogen at 46%. Column 2 says that one extra gram of phosphate alone would need -20 g of NPK 12-12-17, $+34$ g of NPK 20-10-10 and -9.57 g of urea. The negative amounts mean taking fertilizer away from an existing blend: possible only if the blend contains at least that much. With these three bags phosphate never comes alone, so raising it without touching N and K requires trading bags against each other. This is what A^{-1} is good for: not for solving, but for reading off sensitivities.

4. Dividing row i by b_i makes each residual the relative miss. The weighted solution is 297.42 g of NPK 12-12-17-2, 222.46 g of NPK 20-10-10 and 151.78 g of urea. The delivered nutrients are 150.00 g N, 57.94 g phosphate, 72.81 g potash and 5.95 g MgO, relative misses of 0%, -3.4%, $+4.0\%$ and -0.9%, compared with 0%, -1.0%, $+0.9\%$ and -25.3% unweighted. More NPK 12-12-17-2 brings the magnesium almost to target, and the price is paid in phosphate and potash. Which weighting is right is the officer's call; the program only does what the weights say.

5. $l_{11} = \sqrt{4} = 2,\ l_{21} = 2/2 = 1,\ l_{31} = 2/2 = 1;\ l_{22} = \sqrt{5-1} = 2,\ l_{32} = (3 - 1 \times 1)/2 = 1;\ l_{33} = \sqrt{6-1-1} = 2$. So $L = \begin{pmatrix} 2 & 0 & 0 \\ 1 & 2 & 0 \\ 1 & 1 & 2 \end{pmatrix}$, and multiplying LL^{T} gives back S. For $\begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$: $l_{11} = 1$, $l_{21} = 2$, and $l_{22} = \sqrt{1-4} = \sqrt{-3}$, not a real number. The matrix is symmetric but not positive definite: $z = (1, -1)$ gives $z^{\mathrm{T}}Sz = 1 - 2 - 2 + 1 = -2 < 0$. The failure of Cholesky is the proof.

6. $\| a \| = 5$ and $a_1 = 3 > 0$, so $\alpha = -5$ and $v = a - \alpha e_1 = (3 + 5, 4) = (8, 4)$, $v^{\mathrm{T}}v = 80$. Then $H = I - \frac{2}{80}\begin{pmatrix} 64 & 32 \\ 32 & 16 \end{pmatrix} = \begin{pmatrix} 1 - 1.6 & -0.8 \\ -0.8 & 1 - 0.4 \end{pmatrix} = \begin{pmatrix} -0.6 & -0.8 \\ -0.8 & 0.6 \end{pmatrix}$. $H^{\mathrm{T}}H = \begin{pmatrix} 0.36 + 0.64 & 0.48 - 0.48 \\ 0.48 - 0.48 & 0.64 + 0.36 \end{pmatrix} = I$, and $Ha = (-1.8 - 3.2,\ -2.4 + 2.4) = (-5, 0)$, as intended. Had we chosen $\alpha = +5$, v would be $(-2, 4)$, which is fine here, but for $a = (1, 10^{-9})$ that choice would compute $v_1 = 1 - \| a \|$ as the difference of two nearly equal numbers and lose all its digits.

7. `lu_blend` prints 3.34×10^{10} multiplications for eliminating 100 times and 4.33×10^{8} for factoring once. At 10^{9} per second that is about 33 seconds against 0.43 seconds. (A library LU runs much faster than a simple loop, but the ratio between the two approaches stays the same.)

References

[1] S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers*, 8th ed. New York, NY, USA: McGraw-Hill Education, 2021.

[2] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical Analysis*, 10th ed. Boston, MA, USA: Cengage Learning, 2016.

[3] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed. Baltimore, MD, USA: Johns Hopkins University Press, 2013.

[4] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: SIAM, 2002.

[5] Commandant Benoît, “Note sur une méthode de résolution des équations normales provenant de l’application de la méthode des moindres carrés à un système d’équations linéaires en nombre inférieur à celui des inconnues (Procédé du Commandant Cholesky),” *Bulletin Géodésique*, vol. 2, pp. 67–77, 1924.

[6] L. N. Trefethen and D. Bau, *Numerical Linear Algebra*, 25th anniversary ed. Philadelphia, PA, USA: SIAM, 2022.

[7] A. S. Householder, “Unitary triangularization of a nonsymmetric matrix,” *Journal of the ACM*, vol. 5, no. 4, pp. 339–342, 1958.

[8] SciPy Developers, “SciPy 1.18.0 Release Notes,” Jun. 19, 2026, and “scipy 1.18.1” on the Python Package Index, Aug. 21, 2026. [Online]. Available: https://docs.scipy.org/doc/scipy/release/1.18.0-notes.html and https://pypi.org/project/scipy/

[9] NumPy Developers, “NumPy 2.5.0 Release Notes,” Jun. 21, 2026, and “numpy 2.5.3” on the Python Package Index, Sep. 6, 2026. [Online]. Available: https://numpy.org/doc/2.5/release/2.5.0-notes.html and https://pypi.org/project/numpy/

[10] SciPy Developers, “scipy.linalg.lu” and “scipy.linalg.lu_factor,” SciPy v1.18 Manual, accessed Sep. 2026. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lu.html

[11] C. L. Lawson and R. J. Hanson, *Solving Least Squares Problems*. Philadelphia, PA, USA: SIAM, 1995 (reprint of the 1974 edition).

Chapter 7

Interpolation: Lagrange, Newton and Splines

Tables are how the world hands us functions. A tank maker prints liters against dipstick depth, a weather service gives temperatures every three hours, a lender lists instalments for a few loan amounts, a battery maker lists voltage against charge. The value we need almost never sits on a line of the table. *Interpolation* is the art of reading between the lines: building a function that passes exactly through the tabulated points and using it in between. This chapter reads the dipstick table of a lying cylindrical tank, where the exact answer is known and every error can be measured. By the end of the chapter you will be able to interpolate linearly and explain its error; build the unique interpolating polynomial in the Lagrange form and in Newton's divided-difference form and evaluate it cheaply; state and use the error formula of polynomial interpolation; recognize Runge's phenomenon and avoid it with Chebyshev points; derive the cubic spline, set up its tridiagonal system with natural or clamped ends and solve it in linear time; program all of it in Java and Python; compare with NumPy and SciPy; and choose, for a given table, the method that is actually best, which is not always the most sophisticated one.

7.1 Pak Darto's Dipstick

Pak Darto runs a small fuel kiosk at the edge of his village. The fuel waits in a steel tank lying on its side, 200 cm long and 120 cm across, and the only way to know how much is inside is a wooden dipstick: push it down to the bottom, pull it out, and read how deep the wet mark goes. Every evening he writes the depth in a notebook, converts it to liters and compares it with what the pump says it sold. When the two disagree by more than a little, something is wrong: a leak, a faulty pump, a tired helper who forgot to write down a sale.

The conversion is the problem. A tank lying on its side is not a bucket: the first 10 cm of depth hold very little, because the bottom is narrow, while 10 cm in the middle of the tank hold a lot. The tank maker supplied a printed table with the volume at every 10 cm, and a painted chart on the kiosk wall repeats every second line of it, every 20 cm. The dipstick, however, says 47 cm, or 13 cm, or 108 cm. For years Pak Darto has drawn a straight line between the two nearest marks in his head. His nephew, who studies at a vocational school in town, suspects that this is where some of the "missing" liters come from, and he is right: near the bottom and the top of the tank, the straight-line reading from the wall chart is wrong by more than 30 liters.

This tank is a perfect teacher, because unlike most tables its exact function is known. A cylinder of radius r and length L lying on its side, filled to depth h, holds the area of a circular segment times the length:

$$V(h) = L\left[r^2 \arccos\left(\frac{r-h}{r}\right) - (r-h)\sqrt{2rh-h^2}\right]. \tag{7.1}$$

With $r = 60$ cm, $L = 200$ cm and the result divided by 1000 to turn cubic centimeters into liters, this reproduces the tank maker's table: 247.80 L at 20 cm, 1130.97 L at 60 cm (half full), 2261.95

L full. We will pretend not to know Equation (7.1) while we interpolate, and then use it to measure exactly how wrong each method is. **Figure 7.1** shows the tank.

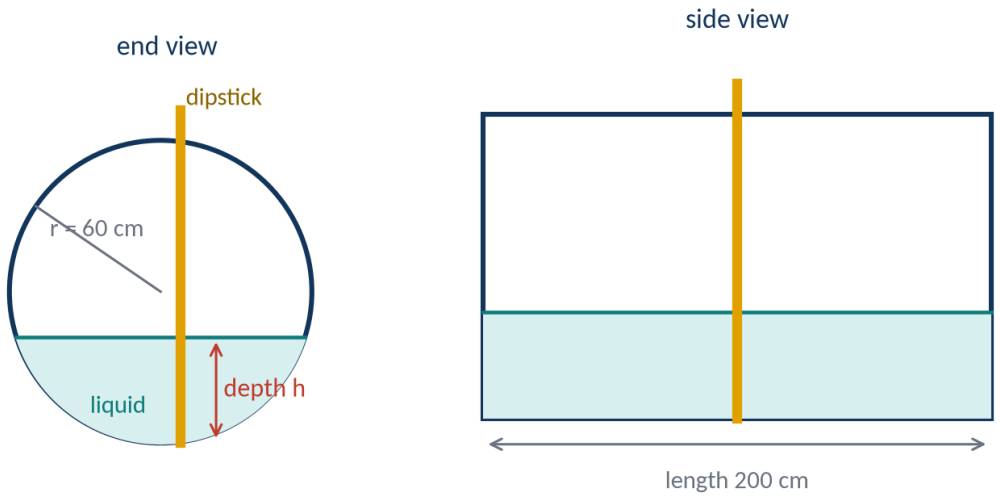


Figure 7.1 The lying tank of the fuel kiosk: the dipstick measures the depth h, and the volume is the area of the circular segment times the length.

7.2 The Idea in Plain Words

7.2.1 A straight line between two marks

The wall chart gives depths $x_0, x_1, \ldots, x_n$ (0, 20, ..., 120 cm) and volumes $y_0, y_1, \ldots, y_n$. For a depth x between two marks x_i and x_{i+1}, the straight line through the two points gives

$$p(x) = y_i + \frac{x - x_i}{x_{i+1} - x_i}\,(y_{i+1} - y_i). \tag{7.2}$$

The fraction says how far along the interval x lies, from 0 at x_i to 1 at x_{i+1}, and the volume moves the same fraction of the way from y_i to y_{i+1}. At 50 cm, halfway between 40 cm (660.02 L) and 60 cm (1130.97 L), this gives $660.02 + 0.5 \times 470.95 = 895.50$ L. The tank maker's table says 892.09 L at 50 cm, so the straight line is 3.4 L too high. At 10 cm it gives $0 + 0.5 \times 247.80 = 123.90$ L against the true 90.03 L: 33.9 L, more than a third too much.

Why so different? Because the volume curve bends, and it bends most near the bottom and the top. Between two marks, a straight line cuts across the bend. Near the bottom the curve is convex (it curves upward, like a smile), the chord lies above it, and the straight line reads too high; near the top the curve is concave and the line reads too low. In the middle of the tank the curve is almost straight, and so is the line.

7.2.2 A curve through more marks

If one straight piece cannot follow the bend, use a curve that passes through more marks. A parabola through three points bends once. Through the marks at 0, 20 and 40 cm it gives, at 10 cm, 103.35 L (we compute this by hand in Section 7.3.2): still 13 L too high, but much better than 123.90. A cubic through the four marks 0, 20, 40, 60 cm gives 96.74 L, and the polynomial of degree 6 through all seven marks of the wall chart gives 93.62 L, only 3.6 L off. So far, more points mean a better answer.

That hope is not always fulfilled. Section 7.3.6 shows a harmless-looking function for which the polynomial through more and more equally spaced points gets *worse*, wildly so, near the ends of the table. And the most popular modern answer, the cubic spline, turns out to be worse than the degree-6 polynomial for this particular tank, for a reason that teaches something about the tank. Before judging, we need the tools.

7.3 Polynomial Interpolation

7.3.1 One polynomial, and only one

> **DEFINITION.** Given $n+1$ points $(x_0, y_0), \dots, (x_n, y_n)$ with distinct x_i (the *nodes*), a polynomial p *interpolates* them if $p(x_i) = y_i$ for every i. **Theorem** [1], [2]: there is exactly one interpolating polynomial of degree at most n.

Why at most one. If p and q both interpolate, their difference $p - q$ has degree at most n and is zero at the $n+1$ nodes. A nonzero polynomial of degree at most n has at most n roots, so $p - q$ is the zero polynomial.

Why at least one. Writing $p(x) = c_0 + c_1 x + \cdots + c_n x^n$ and demanding $p(x_i) = y_i$ gives $n+1$ linear equations for the $n+1$ coefficients, with the *Vandermonde matrix* (rows $1, x_i, x_i^2, \dots, x_i^n$). By the uniqueness just shown, the homogeneous system (all $y_i = 0$) has only the zero solution, so the matrix is nonsingular and the system has a solution. Solving it is a poor way to find p, however: Chapter 6 saw how ill-conditioned the Vandermonde matrix becomes. The next two forms produce the same polynomial without solving anything.

7.3.2 The Lagrange form

The idea of Joseph-Louis Lagrange (1795) is to build, for each node x_i, a polynomial ℓ_i that is 1 at x_i and 0 at every other node:

$$\ell_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}, \qquad p(x) = \sum_{i=0}^{n} y_i \, \ell_i(x). \tag{7.3}$$

Each ℓ_i has degree n (a product of n linear factors); the numerator vanishes at every other node, and the denominator makes the value 1 at x_i. So $p(x_k) = \sum_i y_i \, \ell_i(x_k) = y_k$, as required. The ℓ_i are called the *Lagrange basis polynomials*.

By hand, with the marks 0, 20 and 40 cm at $x = 10$:

- $\ell_0(10) = \frac{(10-20)(10-40)}{(0-20)(0-40)} = \frac{(-10)(-30)}{800} = 0.375$
- $\ell_1(10) = \frac{(10-0)(10-40)}{(20-0)(20-40)} = \frac{(10)(-30)}{-400} = 0.75$
- $\ell_2(10) = \frac{(10-0)(10-20)}{(40-0)(40-20)} = \frac{(10)(-10)}{800} = -0.125$

and $p(10) = 0.375 \times 0 + 0.75 \times 247.80 - 0.125 \times 660.02 = 185.85 - 82.50 = 103.35$ L. Notice that the three weights add up to 1 (as they always do, because interpolating the constant function 1 must give 1), and that one of them is negative: the parabola "borrows" from the far mark to bend.

The Lagrange form is easy to write and to reason with, but each evaluation costs $O(n^2)$ operations, and adding a node means recomputing every basis polynomial. The Newton form fixes both.

7.3.3 The Newton form and divided differences

Isaac Newton wrote the interpolating polynomial as a sum of terms that each vanish at one more node:

$$\begin{aligned} p(x) &= f[x_0] + f[x_0,x_1](x-x_0) + f[x_0,x_1,x_2](x-x_0)(x-x_1) + \cdots \\ &\quad + f[x_0,\dots,x_n]\prod_{j=0}^{n-1}(x-x_j). \end{aligned} \tag{7.4}$$

The coefficients $f[\dots]$ are **divided differences**, defined recursively: $f[x_i] = y_i$, and

$$f[x_i,\dots,x_{i+k}] = \frac{f[x_{i+1},\dots,x_{i+k}] - f[x_i,\dots,x_{i+k-1}]}{x_{i+k} - x_i}. \tag{7.5}$$

The first divided difference $f[x_i, x_{i+1}]$ is the slope of the chord between two neighbors; the second is a difference of slopes divided by a width, a measure of curvature (for a smooth function, $f[x_0,x_1,x_2] \approx f''/2$); in general $f[x_0,\dots,x_k] = f^{(k)}(\xi)/k!$ for some ξ between the nodes [2]. Why does Equation (7.4) interpolate? The first $k+1$ terms form a polynomial that interpolates $x_0,\dots,x_k$ (this is proved by induction on k using Equation (7.5)), and every later term contains the factor $(x-x_0)\cdots(x-x_k)$ and vanishes at those nodes.

The divided differences are computed in a triangular table, column by column. For the first four marks of the wall chart:

Table 7.1 The divided-difference table for the first four marks; the top entry of each column is a coefficient of the Newton form.

x_i	$f[x_i]$	First	Second	Third
0	0.00	$247.80/20 = 12.39$	$(20.611 - 12.39)/40 = 0.205525$	$(0.0734125 - 0.205525)/60 = -0.00220188$
20	247.80	$412.22/20 = 20.611$	$(23.5475 - 20.611)/40 = 0.0734125$	
40	660.02	$470.95/20 = 23.5475$		
60	1130.97			

The Newton coefficients are the top row: 0, 12.39, 0.205525, −0.00220188. The cubic through these four marks is

$$p_3(x) = 0 + 12.39\,x + 0.205525\,x(x-20) - 0.00220188\,x(x-20)(x-40),$$

and at $x = 10$: $123.90 + 0.205525 \times 10 \times (-10) - 0.00220188 \times 10 \times (-10) \times (-30) = 123.90 - 20.55 - 6.61 = 96.74$ L. Read the three terms as a story: the chord says 123.90, the curvature correction pulls 20.55 L off, and the change in curvature another 6.61 L. Adding a fifth mark adds one more term and leaves these alone, which is the second advantage of the Newton form.

7.3.4 Evaluating cheaply: nested multiplication

Written as in Equation (7.4), the degree-n polynomial needs $O(n^2)$ multiplications per evaluation. Factor it like Horner's rule for ordinary polynomials:

$$p(x) = c_0 + (x - x_0)(c_1 + (x - x_1)(c_2 + \cdots + (x - x_{n-1})\, c_n)), \tag{7.6}$$

with $c_k = f[x_0, \ldots, x_k]$. Evaluate from the inside out: start with c_n, and n times multiply by $(x - x_k)$ and add c_k. That is n multiplications per evaluation, after an $O(n^2)$ table computed once.

7.3.5 How wrong is an interpolating polynomial?

Knowing the answer is exact at the nodes says nothing about in between. The fundamental error formula does [1], [2]: if f has $n + 1$ continuous derivatives and p interpolates it at $x_0, \ldots, x_n$, then for each x there is a ξ in the smallest interval containing x and the nodes such that

$$f(x) - p(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^{n} (x - x_i). \tag{7.7}$$

Why. Fix x (not a node) and define $g(t) = f(t) - p(t) - K \prod_i (t - x_i)$, with the constant K chosen so that $g(x) = 0$. Then g vanishes at $n + 2$ points: the $n + 1$ nodes and x. By Rolle's theorem g' vanishes at $n + 1$ points between them, g'' at n, and so on, until $g^{(n+1)}$ vanishes at some ξ. But $p^{(n+1)} = 0$ (degree n) and the product's $(n + 1)$st derivative is $(n + 1)!$, so $0 = f^{(n+1)}(\xi) - K(n + 1)!$, which gives K and the formula.

For linear interpolation ($n = 1$) on an interval of width h, the product $(x - x_i)(x - x_{i+1})$ is largest in the middle, where it equals $h^2/4$ in absolute value, so the error is at most $\frac{h^2}{8} \max|f''|$. Check this on the tank at 50 cm. Differentiating Equation (7.1) gives $V'(h) = 2L\sqrt{2rh - h^2}/1000$ liters per centimeter (the width of the liquid surface times the length, which is what one extra centimeter of depth adds) and $V''(h) = 2L(r - h)/\left(1000\sqrt{2rh - h^2}\right)$. At the midpoint of an interval the error is exactly $\frac{h^2}{8} V''(\xi)$ for some ξ in the interval. Between 40 and 60 cm, V'' is about 0.068 near 50 cm, which gives $\frac{400}{8} \times 0.068 = 3.4$ L, and the error measured at 50 cm is indeed 3.4 L; the worst-case bound, with the largest V'' on the interval (0.141, at 40 cm), is 7.1 L. Near the bottom V'' grows without limit ($\sqrt{2rh - h^2} \to 0$ as $h \to 0$), and so does the bound, which is why the straight line fails there.

7.3.6 Runge's phenomenon and Chebyshev points

Equation (7.7) has two parts: the derivative factor, which depends on f, and the node polynomial $\prod(x - x_i)$, which depends only on where the nodes are. For equally spaced nodes on $[-1,1]$ the node polynomial is small in the middle and huge near the ends, and for some functions the high derivatives grow faster than $(n + 1)!$. Carl Runge found the classic example in 1901 [3]:

$$f(x) = \frac{1}{1 + 25x^2}, \qquad -1 \le x \le 1. \tag{7.8}$$

It is a smooth bell, nothing unusual about it. Yet its interpolating polynomial through $n + 1$ equally spaced points *diverges* near ± 1 as n grows: the program of Section 7.6 finds a largest error of 0.43 for $n = 5$, 1.92 for $n = 10$ and 59.8 for $n = 20$. **Figure 7.2** shows the degree-10

polynomial swinging up to almost 2 near the ends, where the function is below 0.04. The cause lies off the real line: f has poles at $x = \pm i/5$, close to the interval, and this limits how well any polynomial can follow it with badly placed nodes [4].

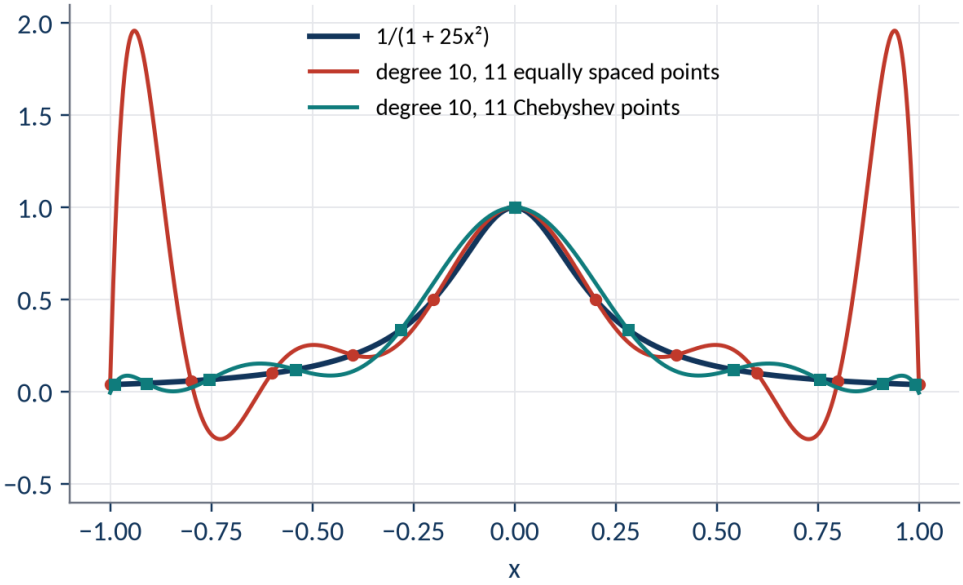


Figure 7.2 Runge's function and its degree-10 interpolants: through 11 equally spaced points the polynomial swings wildly near the ends; through 11 Chebyshev points it follows the function closely.

The cure is to move the nodes. The **Chebyshev points**

$$x_i = \cos\frac{(2i+1)\pi}{2n+2}, \qquad i = 0,1,\dots,n, \tag{7.9}$$

are the projections onto the axis of equally spaced points on a half circle, so they crowd toward the ends. They are exactly the nodes that make the largest value of $|\prod(x - x_i)|$ on $[-1,1]$ as small as possible, namely 2^{-n} [2], [4]. With them the Runge errors fall to 0.109 for $n = 10$ and 0.015 for $n = 20$, and they keep falling as n grows. How big is the difference? For $n = 10$ on $[-1,1]$, the largest value of $|\prod(x - x_i)|$ is about 8.5×10^{-3} for equally spaced nodes, reached in the outermost intervals, and exactly $2^{-10} \approx 9.8 \times 10^{-4}$ for Chebyshev nodes, spread evenly over the whole interval: almost nine times smaller, and the ratio grows quickly with n. For any interval $[a, b]$, map the points with $x \mapsto \frac{a+b}{2} + \frac{b-a}{2}x$. The lesson is general: if you can choose where to measure, do not choose equal spacing for a high-degree polynomial; crowd the measurements toward the ends.

> **KEY INSIGHT:** More points do not automatically mean a better polynomial. The error Equation (7.7) is a product of what the function does (its high derivatives) and where the points are (the node polynomial). Equal spacing makes the node polynomial blow up near the ends; Chebyshev spacing keeps it uniformly small.

7.3.7 When the slopes are known too: Hermite interpolation

Sometimes a table gives more than values. For the tank we know the slope $V'(h)$ as well, the width of the liquid surface times the length, and at the bottom it is zero. A polynomial that matches both the values and the slopes at the nodes is a *Hermite interpolant*, after Charles Hermite. On one interval $[x_0, x_1]$ of width h with values y_0, y_1 and slopes s_0, s_1 it is the unique cubic with those four properties, and at the midpoint it takes the simple form

$$p\left(\frac{x_0 + x_1}{2}\right) = \frac{y_0 + y_1}{2} + \frac{h}{8}\,(s_0 - s_1). \tag{7.10}$$

The first term is the straight line; the second bends it using the difference of the end slopes. For the first interval of the tank, $y_0 = 0$, $y_1 = 247.80$, $s_0 = 0$ and $s_1 = V'(20) = 0.4\sqrt{2400 - 400} = 17.89$ L/cm, so $p(10) = 123.90 + \frac{20}{8}(0 - 17.89) = 123.90 - 44.72 = 79.18$ L. The true volume is 90.03 L: the straight line was 33.9 L too high, the Hermite cubic is 10.9 L too low. Knowing the slopes has cut the error by two thirds on the worst interval, using only the two neighboring marks. Piecewise Hermite cubics, with slopes either known or estimated from the data, are the basis of the shape-preserving PCHIP interpolant of Section 7.8.

7.4 Cubic Splines

7.4.1 Piecewise, but smooth

When the points are given, as in a table, and you cannot move them, there is another escape from high degrees: use a *different low-degree polynomial on each interval*, and glue the pieces together smoothly. Straight pieces give Equation (7.2), with kinks at the nodes. Cubic pieces have enough freedom to hide the joins: the **cubic spline** is a function S that is a cubic polynomial on each interval $[x_i, x_{i+1}]$, interpolates the data, and has continuous first and second derivatives everywhere. The name comes from the thin flexible strips of wood (splines) that draftsmen and shipbuilders bent through fixed pins to draw smooth curves; a bent strip takes the shape that minimizes its bending energy, and the natural cubic spline minimizes $\int S''(x)^2\,dx$ among all smooth interpolants, the mathematical version of the same property [5].

7.4.2 Deriving the tridiagonal system

Count the conditions. With n intervals there are $4n$ cubic coefficients. Interpolation at both ends of each piece gives $2n$ conditions; continuity of S' and S'' at the $n - 1$ inner nodes gives $2(n - 1)$ more. That is $4n - 2$: two conditions are missing, and they are supplied at the two ends of the table.

The neat way to set it up uses the unknown second derivatives $M_i = S''(x_i)$ at the nodes. On $[x_i, x_{i+1}]$, of width $h_i = x_{i+1} - x_i$, S is cubic, so S'' is linear, going from M_i to M_{i+1}. Integrating twice and choosing the two constants of integration so that $S(x_i) = y_i$ and $S(x_{i+1}) = y_{i+1}$ gives, with $a = x_{i+1} - x$ and $b = x - x_i$,

$$S(x) = \frac{M_i a^3 + M_{i+1} b^3}{6h_i} + \left(\frac{y_i}{h_i} - \frac{M_i h_i}{6}\right) a + \left(\frac{y_{i+1}}{h_i} - \frac{M_{i+1} h_i}{6}\right) b. \tag{7.11}$$

This piece interpolates and has second derivatives M_i and M_{i+1} at its ends, so S and S'' are continuous by construction. What remains is continuity of S' at each inner node x_i. Write $\Delta_i = (y_{i+1} - y_i)/h_i$ for the chord slope of interval i. Differentiating Equation (7.11) (remember that a decreases as x grows) and putting $x = x_i$, where $a = h_i$ and $b = 0$, gives the slope at the left end of interval i: $S'(x_i^+) = \Delta_i - h_i(2M_i + M_{i+1})/6$. Doing the same on interval $i - 1$ at its right end, where $a = 0$ and $b = h_{i-1}$, gives $S'(x_i^-) = \Delta_{i-1} + h_{i-1}(M_{i-1} + 2M_i)/6$. The two slopes must be equal. Multiplying the equation by 6 and collecting the curvatures on the left:

$$h_{i-1}M_{i-1} + 2(h_{i-1} + h_i)\,M_i + h_i M_{i+1} = 6\,(\Delta_i - \Delta_{i-1}), \qquad i = 1, \dots, n-1. \tag{7.12}$$

Each equation involves only three neighboring unknowns: the system is **tridiagonal**. Its diagonal $2(h_{i-1} + h_i)$ is larger than the sum of the off-diagonals $h_{i-1} + h_i$, so it is strictly diagonally dominant (Chapter 5): it has a unique solution, and elimination needs no pivoting.

7.4.3 The two end conditions

- **Natural spline:** $M_0 = M_n = 0$, no bending at the ends, like a free draftsman's strip. It is the default in many textbooks, but it is an assumption about the data, and often a wrong one.
- **Clamped spline:** the end slopes $S'(x_0) = s_0$ and $S'(x_n) = s_n$ are given, which adds the equations $2h_0M_0 + h_0M_1 = 6(\Delta_0 - s_0)$ and $h_{n-1}M_{n-1} + 2h_{n-1}M_n = 6(s_n - \Delta_{n-1})$. Use it when the slopes are known. For the tank they are: the dipstick at the very bottom or the very top touches a horizontal tangent of the circle, and $V'(0) = V'(120) = 0$.
- **Not-a-knot:** the third derivative is continuous at x_1 and x_{n-1}, so the first two and the last two pieces are each one cubic. This is SciPy's default [6].

7.4.4 The Thomas algorithm

A tridiagonal system is solved by Gaussian elimination without pivoting that only touches the three diagonals. With lower diagonal l_i, diagonal d_i, upper diagonal u_i and right-hand side r_i:

$$d_i \leftarrow d_i - \frac{l_i}{d_{i-1}}\,u_{i-1}, \quad r_i \leftarrow r_i - \frac{l_i}{d_{i-1}}\,r_{i-1} \quad (i = 1, \dots, n); \qquad M_n = \frac{r_n}{d_n}, \quad M_i = \frac{r_i - u_i M_{i+1}}{d_i}. \tag{7.13}$$

This is the LU decomposition of Chapter 6 for a matrix with bandwidth 1, and it costs about $8n$ operations instead of $n^3/3$: a spline through a million points costs a few milliseconds. The algorithm carries the name of the physicist Llewellyn Thomas, who described it in 1949 [7], and it is safe without pivoting whenever the matrix is diagonally dominant, as ours is [1].

7.4.5 How accurate is a spline?

For a function with four continuous derivatives, the clamped cubic spline with largest interval h satisfies $\max|f - S| \le \frac{5}{384}h^4 \max\left|f^{(4)}\right|$ [2], [5]. Halving the spacing divides the error by 16, and the constant $5/384$ is small. This is why splines are the default for smooth data. The hypothesis matters, though: the bound assumes $f^{(4)}$ is bounded. For the tank it is not: V'' is already infinite at both ends. Section 7.6 shows what that does.

7.4.6 One spline reading by hand

Once the curvatures are known, reading the spline is arithmetic. The program of Section 7.6 finds, for the natural spline through the seven wall-chart marks, $M_0 = 0$ and $M_1 = 0.5990$. At $x = 10$ cm we are in the first interval, $x_0 = 0$, $x_1 = 20$, $h_0 = 20$, and $a = x_1 - x = 10$, $b = x - x_0 = 10$. Equation (7.11) gives

$$S(10) = \frac{0 \cdot 10^3 + 0.5990 \cdot 10^3}{6 \cdot 20} + \left(\frac{0}{20} - 0\right) \cdot 10 + \left(\frac{247.80}{20} - \frac{0.5990 \cdot 20}{6}\right) \cdot 10$$
$$= 4.99 + 0 + (12.39 - 2.00) \times 10 = 108.93 \text{ L}.$$

The last bracket is the chord slope 12.39 corrected by the curvature. Compared with the straight line (123.90 L) the spline has bent the reading down by 15 L toward the true 90.03 L, but the natural end condition $M_0 = 0$ has stopped it from bending enough: near the bottom the true curvature is not zero but infinite.

7.4.7 Reading the table backward

Sometimes the question runs the other way. A tanker delivers 1000 L; to which depth should the fuel rise? This is *inverse interpolation*: swap the roles of the two columns and interpolate depth as a function of volume. It works whenever the table is monotone, as a volume table always is, because then every volume belongs to exactly one depth. Suppose the dipstick read 40 cm (660.02 L) before a delivery of 1000 L. Afterward the tank holds 1660.02 L, which lies between the marks at 80 cm (1601.93 L) and 100 cm (2014.15 L), and the straight line read backward gives $80 + 20 \times (1660.02 - 1601.93)/(2014.15 - 1601.93) = 80 + 20 \times 0.1409 = 82.8$ cm. The exact depth from Equation (7.1) is 82.6 cm, so a dipstick reading far from 82.6 cm after the delivery means the delivery was short. The alternative is to solve $p(x) = 1660.02$ for x with a root finder from Chapters 2 and 3, using the interpolant p of the forward table. The two answers differ slightly, because interpolating the inverse function is not the same as inverting the interpolant, but both are good near the middle of the tank, where the curve is nearly straight.

7.5 Building It in Java and Python

Two programs: `interp` reads the tank maker's table from `data/dipstick.csv`, takes every second line as the wall chart (7 marks, 20 cm apart), and tests linear, Lagrange and Newton interpolation on the 10 cm lines it did not use, then against Equation (7.1) on a 1 mm grid, and finally runs Runge's experiment. `spline` builds natural and clamped splines from the same marks and from all 13 lines of the table. The tank formula Equation (7.1) is used only to measure errors, never to interpolate. Storing the table as a file, rounded to two decimals as a printed table would be, keeps both languages working from exactly the same numbers.

Listing 7.1 is the Python version of the first program. `lagrange` is Equation (7.3) written as two loops; `divided_differences` builds the table of Equation (7.5) column by column; `newton_eval` is the nested form Equation (7.6).

Listing 7.1 `interp.py`: linear, Lagrange and Newton interpolation of the dipstick table, errors against the exact tank formula, and Runge's experiment with equally spaced and Chebyshev points.

```
"""Reading between the marks of a dipstick table: linear, Lagrange and
Newton interpolation for a horizontal tank, and Runge's warning."""
import math

R, LENGTH = 60.0, 200.0          # tank radius and length in cm


def tank_liters(h):
    """Exact volume of liquid (liters) at depth h cm in a lying cylinder."""
    c = R - h
    area = R * R * math.acos(c / R) - c * math.sqrt(2.0 * R * h - h * h)
    return area * LENGTH / 1000.0


def lagrange(xs, ys, x):
    """Sum of y_i times the basis polynomial that is 1 at x_i, 0 elsewhere."""
    total = 0.0
    for i in range(len(xs)):
        basis = 1.0
        for j in range(len(xs)):
            if j != i:
                basis *= (x - xs[j]) / (xs[i] - xs[j])
        total += ys[i] * basis
    return total


def divided_differences(xs, ys):
    """Full table: table[k][i] = f[x_i, ..., x_(i+k)]."""
    table = [list(ys)]
    for k in range(1, len(xs)):
        prev = table[-1]
        table.append([(prev[i + 1] - prev[i]) / (xs[i + k] - xs[i])
                      for i in range(len(xs) - k)])
    return table


def newton_eval(xs, coef, x):
    """Nested (Horner-like) evaluation of the Newton form."""
    result = coef[-1]
    for k in range(len(coef) - 2, -1, -1):
        result = result * (x - xs[k]) + coef[k]
    return result
```

Listing 7.1 (Continued) *interp.py: linear, Lagrange and Newton interpolation of the dipstick table, errors against the exact tank formula, and Runge's experiment with equally spaced and Chebyshev points.*

```
def linear(xs, ys, x):
    for i in range(len(xs) - 1):
        if xs[i] <= x <= xs[i + 1]:
            t = (x - xs[i]) / (xs[i + 1] - xs[i])
            return ys[i] + t * (ys[i + 1] - ys[i])
    raise ValueError("x outside the table")


def runge(x):
    return 1.0 / (1.0 + 25.0 * x * x)


def read_table(path):
    """Rows of (depth in cm, liters) from the tank maker's CSV table."""
    rows = []
    with open(path, encoding="utf-8") as f:
        next(f)                                     # skip the header
        for line in f:
            h, v = line.strip().split(",")
            rows.append((float(h), float(v)))
    return rows


def main():
    table10 = read_table("../data/dipstick.csv")
    xs = [h for h, _ in table10 if h % 20 == 0]    # the wall chart: 20 cm
    ys = [v for h, v in table10 if h % 20 == 0]
    dd = divided_differences(xs, ys)
    print("Newton coefficients f[x0..xk] from the 20 cm marks")
    for k, col in enumerate(dd):
        print(f"  k = {k}: {col[0]: .6e}")
    coef = [col[0] for col in dd]
    print()
    print("depth   table   linear   error  Lagrange   Newton   error")
    for h, v in table10:
        if h % 20 == 0:
            continue                                # a node: no test there
        li = linear(xs, ys, h)
        la = lagrange(xs, ys, h)
        ne = newton_eval(xs, coef, h)
        print(f"{h:5.0f}{v:9.2f}{li:9.2f}{li - v:8.2f}"
              f"{la:10.2f}{ne:9.2f}{ne - v:8.2f}")
    worst_lin = worst_pol = 0.0
    for k in range(1201):
        h = 0.1 * k
        ex = tank_liters(h)
```

Listing 7.1 (Continued) *interp.py: linear, Lagrange and Newton interpolation of the dipstick table, errors against the exact tank formula, and Runge's experiment with equally spaced and Chebyshev points.*

```
        worst_lin = max(worst_lin, abs(linear(xs, ys, h) - ex))
        worst_pol = max(worst_pol, abs(newton_eval(xs, coef, h) - ex))
    print(f"largest error on 0..120 cm: linear {worst_lin:.2f} L,"
          f" degree 6 {worst_pol:.2f} L")
    print()
    print("Runge's function 1/(1+25x^2) on [-1, 1]: largest error")
    print("   n   equally spaced   Chebyshev points")
    for n in (5, 10, 15, 20):
        eq = [-1.0 + 2.0 * i / n for i in range(n + 1)]
        ch = [math.cos((2 * i + 1) * math.pi / (2 * n + 2))
              for i in range(n + 1)]
        errs = []
        for nodes in (eq, ch):
            vals = [runge(t) for t in nodes]
            worst = 0.0
            for k in range(2001):
                t = -1.0 + k / 1000.0
                worst = max(worst, abs(lagrange(nodes, vals, t) - runge(t)))
            errs.append(worst)
        print(f"{n:4d}{errs[0]:17.4f}{errs[1]:19.4f}")


if __name__ == "__main__":
    main()
```

Listing 7.2 is the Java version. `readTable` is shared with the spline program, so both are compiled together.

Listing 7.2 `Interp.java`: the same interpolation program in Java.

```
import java.io.IOException;
import java.math.BigDecimal;
import java.math.MathContext;
import java.math.RoundingMode;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;

/** Reading between the marks of a dipstick table: linear, Lagrange and
 *  Newton interpolation for a horizontal tank, and Runge's warning. */
public class Interp {
    static final double R = 60.0, LENGTH = 200.0;   // tank radius, length, cm

    /** Exact volume of liquid (liters) at depth h cm in a lying cylinder. */
    static double tankLiters(double h) {
        double c = R - h;
        double area = R * R * Math.acos(c / R) - c * Math.sqrt(2.0 * R * h - h * h);
        return area * LENGTH / 1000.0;
    }

    /** Sum of y_i times the basis polynomial that is 1 at x_i, 0 elsewhere. */
    static double lagrange(double[] xs, double[] ys, double x) {
        double total = 0.0;
        for (int i = 0; i < xs.length; i++) {
            double basis = 1.0;
            for (int j = 0; j < xs.length; j++) {
                if (j != i) basis *= (x - xs[j]) / (xs[i] - xs[j]);
            }
            total += ys[i] * basis;
        }
        return total;
    }

    /** Full table: table[k][i] = f[x_i, ..., x_(i+k)]. */
    static double[][] dividedDifferences(double[] xs, double[] ys) {
        int n = xs.length;
        double[][] table = new double[n][];
        table[0] = ys.clone();
        for (int k = 1; k < n; k++) {
            table[k] = new double[n - k];
            for (int i = 0; i < n - k; i++) {
                table[k][i] = (table[k - 1][i + 1] - table[k - 1][i])
                              / (xs[i + k] - xs[i]);
            }
        }
```

Listing 7.2 (Continued) *Interp.java: the same interpolation program in Java.*

```
        return table;
    }

    /** Nested (Horner-like) evaluation of the Newton form. */
    static double newtonEval(double[] xs, double[] coef, double x) {
        double result = coef[coef.length - 1];
        for (int k = coef.length - 2; k >= 0; k--) {
            result = result * (x - xs[k]) + coef[k];
        }
        return result;
    }

    static double linear(double[] xs, double[] ys, double x) {
        for (int i = 0; i < xs.length - 1; i++) {
            if (xs[i] <= x && x <= xs[i + 1]) {
                double t = (x - xs[i]) / (xs[i + 1] - xs[i]);
                return ys[i] + t * (ys[i + 1] - ys[i]);
            }
        }
        throw new IllegalArgumentException("x outside the table");
    }

    static double runge(double x) { return 1.0 / (1.0 + 25.0 * x * x); }

    /** Rows of (depth in cm, liters) from the tank maker's CSV table. */
    static double[][] readTable(String path) throws IOException {
        List<String> lines = Files.readAllLines(Path.of(path));
        List<double[]> rows = new ArrayList<>();
        for (String line : lines.subList(1, lines.size())) {  // skip header
            String[] p = line.strip().split(",");
            rows.add(new double[] {Double.parseDouble(p[0]),
                                   Double.parseDouble(p[1])});
        }
        return rows.toArray(new double[0][]);
    }
```

Listing 7.2 (Continued) *Interp.java: the same interpolation program in Java.*

```
    public static void main(String[] args) throws IOException {
        double[][] table10 = readTable("../data/dipstick.csv");
        List<double[]> marks = new ArrayList<>();
        for (double[] row : table10) {
            if (row[0] % 20 == 0) marks.add(row);    // the wall chart: 20 cm
        }
        double[] xs = new double[marks.size()], ys = new double[marks.size()];
        for (int i = 0; i < xs.length; i++) {
            xs[i] = marks.get(i)[0];
            ys[i] = marks.get(i)[1];
        }
        double[][] dd = dividedDifferences(xs, ys);
        System.out.println("Newton coefficients f[x0..xk] from the 20 cm marks");
        double[] coef = new double[dd.length];
        for (int k = 0; k < dd.length; k++) {
            coef[k] = dd[k][0];
            String s = e(coef[k], 6);
            System.out.println("  k = " + k + ": "
                               + (s.startsWith("-") ? s : " " + s));
        }
        System.out.println();
        System.out.println("depth   table   linear   error  Lagrange"
                           + "   Newton   error");
        for (double[] row : table10) {
            double h = row[0], v = row[1];
            if (h % 20 == 0) continue;               // a node: no test there
            double li = linear(xs, ys, h);
            double la = lagrange(xs, ys, h);
            double ne = newtonEval(xs, coef, h);
            System.out.println(f(h, 0, 5) + f(v, 2, 9) + f(li, 2, 9)
                    + f(li - v, 2, 8) + f(la, 2, 10) + f(ne, 2, 9)
                    + f(ne - v, 2, 8));
        }
        double worstLin = 0, worstPol = 0;
        for (int k = 0; k <= 1200; k++) {
            double h = 0.1 * k;
            double ex = tankLiters(h);
            worstLin = Math.max(worstLin, Math.abs(linear(xs, ys, h) - ex));
            worstPol = Math.max(worstPol, Math.abs(newtonEval(xs, coef, h) - ex));
        }
        System.out.println("largest error on 0..120 cm: linear " + f(worstLin, 2)
                           + " L, degree 6 " + f(worstPol, 2) + " L");
        System.out.println();
        System.out.println("Runge's function 1/(1+25x^2) on [-1, 1]:"
                           + " largest error");
        System.out.println("   n   equally spaced   Chebyshev points");
```

Listing 7.2 (Continued) *Interp.java: the same interpolation program in Java.*

```
        for (int n : new int[] {5, 10, 15, 20}) {
            double[] eq = new double[n + 1], ch = new double[n + 1];
            for (int i = 0; i <= n; i++) {
                eq[i] = -1.0 + 2.0 * i / n;
                ch[i] = Math.cos((2 * i + 1) * Math.PI / (2 * n + 2));
            }
            StringBuilder line = new StringBuilder(pad("" + n, 4));
            int width = 17;
            for (double[] nodes : new double[][] {eq, ch}) {
                double[] vals = new double[nodes.length];
                for (int i = 0; i < nodes.length; i++) vals[i] = runge(nodes[i]);
                double worst = 0;
                for (int k = 0; k <= 2000; k++) {
                    double t = -1.0 + k / 1000.0;
                    worst = Math.max(worst,
                            Math.abs(lagrange(nodes, vals, t) - runge(t)));
                }
                line.append(f(worst, 4, width));
                width = 19;
            }
            System.out.println(line);
        }
    }

    // ---- number formatting that matches Python's f"{x:w.df}" and ".de"
    static String f(double x, int d) {
        String s = new BigDecimal(x).setScale(d, RoundingMode.HALF_EVEN)
                                    .toPlainString();
        boolean neg = x < 0 || (x == 0 && 1 / x < 0);
        return neg && !s.startsWith("-") ? "-" + s : s;
    }

    static String f(double x, int d, int w) { return pad(f(x, d), w); }

    static String e(double x, int d) {
        if (x == 0) return "0." + "0".repeat(d) + "e+00";
        BigDecimal b = new BigDecimal(x).round(
                new MathContext(d + 1, RoundingMode.HALF_EVEN));
        int exp = b.precision() - b.scale() - 1;
        String digits = b.unscaledValue().abs().toString();
        digits = (digits + "0".repeat(d + 1)).substring(0, d + 1);
        String ex = (exp < 0 ? "-" : "+") + (Math.abs(exp) < 10 ? "0" : "")
                    + Math.abs(exp);
        return (x < 0 ? "-" : "") + digits.charAt(0) + "." + digits.substring(1)
               + "e" + ex;
    }
```

Listing 7.2 (Continued) *Interp.java: the same interpolation program in Java.*

```
    static String pad(String s, int w) {
        return " ".repeat(Math.max(0, w - s.length())) + s;
    }

    static String padRight(String s, int w) {
        return s + " ".repeat(Math.max(0, w - s.length()));
    }
}
```

Listing 7.3 builds the spline. `spline_curvatures` fills the three diagonals of Equation (7.12), adds the natural or clamped end rows, and runs the Thomas algorithm Equation (7.13); `spline_eval` finds the interval and applies Equation (7.11). A natural spline is written as two trivial rows "$1 \cdot M_0 = 0$" and "$1 \cdot M_n = 0$", so both kinds of end use the same solver.

Listing 7.3 `spline.py`: natural and clamped cubic splines by the tridiagonal system of curvatures and the Thomas algorithm.

```
"""Natural and clamped cubic splines through the dipstick table: one small
tridiagonal system for the curvatures, then a cubic between two marks."""
from interp import read_table, tank_liters


def spline_curvatures(xs, ys, slopes=None):
    """Second derivatives M_0..M_n of the cubic spline. slopes=None gives
    the natural spline (M_0 = M_n = 0); slopes=(s0, sn) clamps the end
    slopes. One tridiagonal system, solved by the Thomas algorithm."""
    n = len(xs) - 1
    h = [xs[i + 1] - xs[i] for i in range(n)]
    low, diag, up, rhs = [0.0] * (n + 1), [0.0] * (n + 1), \
        [0.0] * (n + 1), [0.0] * (n + 1)
    for i in range(1, n):
        low[i], diag[i], up[i] = h[i - 1], 2.0 * (h[i - 1] + h[i]), h[i]
        rhs[i] = 6.0 * ((ys[i + 1] - ys[i]) / h[i]
                        - (ys[i] - ys[i - 1]) / h[i - 1])
    if slopes is None:
        diag[0] = diag[n] = 1.0                    # rows say M_0 = M_n = 0
    else:
        diag[0], up[0] = 2.0 * h[0], h[0]
        rhs[0] = 6.0 * ((ys[1] - ys[0]) / h[0] - slopes[0])
        low[n], diag[n] = h[n - 1], 2.0 * h[n - 1]
        rhs[n] = 6.0 * (slopes[1] - (ys[n] - ys[n - 1]) / h[n - 1])
    for i in range(1, n + 1):                      # eliminate below diagonal
        f = low[i] / diag[i - 1]
        diag[i] -= f * up[i - 1]
        rhs[i] -= f * rhs[i - 1]
    m = [0.0] * (n + 1)
    m[n] = rhs[n] / diag[n]
    for i in range(n - 1, -1, -1):                 # back-substitute
        m[i] = (rhs[i] - up[i] * m[i + 1]) / diag[i]
    return m


def spline_eval(xs, ys, m, x):
    """Value of the cubic piece on the interval that contains x."""
    i = 0
    while i < len(xs) - 2 and x > xs[i + 1]:
        i += 1
    h = xs[i + 1] - xs[i]
    a = xs[i + 1] - x
    b = x - xs[i]
    return (m[i] * a * a * a / (6.0 * h) + m[i + 1] * b * b * b / (6.0 * h)
            + (ys[i] / h - m[i] * h / 6.0) * a
            + (ys[i + 1] / h - m[i + 1] * h / 6.0) * b)
```

Listing 7.3 (Continued) *spline.py: natural and clamped cubic splines by the tridiagonal system of curvatures and the Thomas algorithm.*

```
def worst_error(xs, ys, m, lo, hi):
    worst = 0.0
    for k in range(int(round(10 * lo)), int(round(10 * hi)) + 1):
        h = 0.1 * k
        worst = max(worst, abs(spline_eval(xs, ys, m, h) - tank_liters(h)))
    return worst


def main():
    table10 = read_table("../data/dipstick.csv")
    xs = [h for h, _ in table10 if h % 20 == 0]
    ys = [v for h, v in table10 if h % 20 == 0]
    nat = spline_curvatures(xs, ys)
    cla = spline_curvatures(xs, ys, (0.0, 0.0))  # the tank is flat at the ends
    print("curvatures M_i = S''(x_i) at the 20 cm marks")
    print("  natural " + " ".join(f"{v:7.4f}" for v in nat))
    print("  clamped " + " ".join(f"{v:7.4f}" for v in cla))
    print()
    print("depth   table  natural   error  clamped   error")
    for h, v in table10:
        if h % 20 == 0:
            continue
        s1 = spline_eval(xs, ys, nat, h)
        s2 = spline_eval(xs, ys, cla, h)
        print(f"{h:5.0f}{v:9.2f}{s1:9.2f}{s1 - v:8.2f}{s2:9.2f}{s2 - v:8.2f}")
    print()
    x10 = [h for h, _ in table10]
    y10 = [v for _, v in table10]
    print("largest error against the exact tank formula")
    print("  marks  ends       0..120 cm   20..100 cm")
    for name, a, b in (("20 cm", xs, ys), ("10 cm", x10, y10)):
        for kind, sl in (("natural", None), ("clamped", (0.0, 0.0))):
            m = spline_curvatures(a, b, sl)
            print(f"  {name}  {kind}{worst_error(a, b, m, 0, 120):10.2f} L"
                  f"{worst_error(a, b, m, 20, 100):11.2f} L")
    m10 = spline_curvatures(x10, y10, (0.0, 0.0))
    print(f"a reading of 47 cm: {spline_eval(x10, y10, m10, 47.0):.1f} L"
          f" (exact {tank_liters(47.0):.1f} L)")


if __name__ == "__main__":
    main()
```

Listing 7.4 `Spline.java`: the cubic splines in Java.

```
import java.io.IOException;
import java.math.BigDecimal;
import java.math.MathContext;
import java.math.RoundingMode;

/** Natural and clamped cubic splines through the dipstick table: one small
 *  tridiagonal system for the curvatures, then a cubic between two marks. */
public class Spline {
    /** Second derivatives M_0..M_n of the cubic spline. slopes == null gives
     *  the natural spline (M_0 = M_n = 0); slopes = {s0, sn} clamps the end
     *  slopes. One tridiagonal system, solved by the Thomas algorithm. */
    static double[] curvatures(double[] xs, double[] ys, double[] slopes) {
        int n = xs.length - 1;
        double[] h = new double[n];
        for (int i = 0; i < n; i++) h[i] = xs[i + 1] - xs[i];
        double[] low = new double[n + 1], diag = new double[n + 1],
                 up = new double[n + 1], rhs = new double[n + 1];
        for (int i = 1; i < n; i++) {
            low[i] = h[i - 1];
            diag[i] = 2.0 * (h[i - 1] + h[i]);
            up[i] = h[i];
            rhs[i] = 6.0 * ((ys[i + 1] - ys[i]) / h[i]
                            - (ys[i] - ys[i - 1]) / h[i - 1]);
        }
        if (slopes == null) {
            diag[0] = diag[n] = 1.0;                // rows say M_0 = M_n = 0
        } else {
            diag[0] = 2.0 * h[0];
            up[0] = h[0];
            rhs[0] = 6.0 * ((ys[1] - ys[0]) / h[0] - slopes[0]);
            low[n] = h[n - 1];
            diag[n] = 2.0 * h[n - 1];
            rhs[n] = 6.0 * (slopes[1] - (ys[n] - ys[n - 1]) / h[n - 1]);
        }
        for (int i = 1; i <= n; i++) {              // eliminate below diagonal
            double f = low[i] / diag[i - 1];
            diag[i] -= f * up[i - 1];
            rhs[i] -= f * rhs[i - 1];
        }
        double[] m = new double[n + 1];
        m[n] = rhs[n] / diag[n];
        for (int i = n - 1; i >= 0; i--) {          // back-substitute
            m[i] = (rhs[i] - up[i] * m[i + 1]) / diag[i];
        }
        return m;
    }
```

Listing 7.4 (Continued) *Spline.java: the cubic splines in Java.*

```
    /** Value of the cubic piece on the interval that contains x. */
    static double eval(double[] xs, double[] ys, double[] m, double x) {
        int i = 0;
        while (i < xs.length - 2 && x > xs[i + 1]) i++;
        double h = xs[i + 1] - xs[i];
        double a = xs[i + 1] - x;
        double b = x - xs[i];
        return m[i] * a * a * a / (6.0 * h) + m[i + 1] * b * b * b / (6.0 * h)
               + (ys[i] / h - m[i] * h / 6.0) * a
               + (ys[i + 1] / h - m[i + 1] * h / 6.0) * b;
    }

    static double worstError(double[] xs, double[] ys, double[] m,
                             double lo, double hi) {
        double worst = 0;
        for (long k = Math.round(10 * lo); k <= Math.round(10 * hi); k++) {
            double h = 0.1 * k;
            worst = Math.max(worst,
                             Math.abs(eval(xs, ys, m, h) - Interp.tankLiters(h)));
        }
        return worst;
    }

    public static void main(String[] args) throws IOException {
        double[][] table10 = Interp.readTable("../data/dipstick.csv");
        int n20 = 0;
        for (double[] row : table10) if (row[0] % 20 == 0) n20++;
        double[] xs = new double[n20], ys = new double[n20];
        double[] x10 = new double[table10.length], y10 = new double[table10.length];
        for (int i = 0, k = 0; i < table10.length; i++) {
            x10[i] = table10[i][0];
            y10[i] = table10[i][1];
            if (table10[i][0] % 20 == 0) {
                xs[k] = table10[i][0];
                ys[k++] = table10[i][1];
            }
        }
        double[] flat = {0.0, 0.0};                 // the tank is flat at the ends
        double[] nat = curvatures(xs, ys, null);
        double[] cla = curvatures(xs, ys, flat);
        System.out.println("curvatures M_i = S''(x_i) at the 20 cm marks");
        StringBuilder a = new StringBuilder("  natural");
        StringBuilder b = new StringBuilder("  clamped");
        for (int i = 0; i < nat.length; i++) {
            a.append(" ").append(f(nat[i], 4, 7));
```

Listing 7.4 (Continued) *Spline.java: the cubic splines in Java.*

```
            b.append(" ").append(f(cla[i], 4, 7));
        }
        System.out.println(a);
        System.out.println(b);
        System.out.println();
        System.out.println("depth   table  natural   error  clamped   error");
        for (double[] row : table10) {
            double h = row[0], v = row[1];
            if (h % 20 == 0) continue;
            double s1 = eval(xs, ys, nat, h), s2 = eval(xs, ys, cla, h);
            System.out.println(f(h, 0, 5) + f(v, 2, 9) + f(s1, 2, 9)
                    + f(s1 - v, 2, 8) + f(s2, 2, 9) + f(s2 - v, 2, 8));
        }
        System.out.println();
        System.out.println("largest error against the exact tank formula");
        System.out.println("  marks  ends       0..120 cm   20..100 cm");
        String[] names = {"20 cm", "10 cm"};
        double[][][] data = {{xs, ys}, {x10, y10}};
        for (int d = 0; d < 2; d++) {
            for (int kind = 0; kind < 2; kind++) {
                double[] px = data[d][0], py = data[d][1];
                double[] m = curvatures(px, py, kind == 0 ? null : flat);
                System.out.println("  " + names[d] + "  "
                        + (kind == 0 ? "natural" : "clamped")
                        + f(worstError(px, py, m, 0, 120), 2, 10) + " L"
                        + f(worstError(px, py, m, 20, 100), 2, 11) + " L");
            }
        }
        double[] m10 = curvatures(x10, y10, flat);
        System.out.println("a reading of 47 cm: " + f(eval(x10, y10, m10, 47.0), 1)
                           + " L (exact " + f(Interp.tankLiters(47.0), 1) + " L)");
    }

    // ---- number formatting that matches Python's f"{x:w.df}" and ".de"
    static String f(double x, int d) {
        String s = new BigDecimal(x).setScale(d, RoundingMode.HALF_EVEN)
                                    .toPlainString();
        boolean neg = x < 0 || (x == 0 && 1 / x < 0);
        return neg && !s.startsWith("-") ? "-" + s : s;
    }

    static String f(double x, int d, int w) { return pad(f(x, d), w); }
```

Listing 7.4 (Continued) *Spline.java: the cubic splines in Java.*

```
    static String e(double x, int d) {
        if (x == 0) return "0." + "0".repeat(d) + "e+00";
        BigDecimal b = new BigDecimal(x).round(
                new MathContext(d + 1, RoundingMode.HALF_EVEN));
        int exp = b.precision() - b.scale() - 1;
        String digits = b.unscaledValue().abs().toString();
        digits = (digits + "0".repeat(d + 1)).substring(0, d + 1);
        String ex = (exp < 0 ? "-" : "+") + (Math.abs(exp) < 10 ? "0" : "")
                  + Math.abs(exp);
        return (x < 0 ? "-" : "") + digits.charAt(0) + "." + digits.substring(1)
               + "e" + ex;
    }

    static String pad(String s, int w) {
        return " ".repeat(Math.max(0, w - s.length())) + s;
    }

    static String padRight(String s, int w) {
        return s + " ".repeat(Math.max(0, w - s.length()));
    }
}
```

Listing 7.4 mirrors the Python function for function; the end slopes are passed as an array, or `null` for the natural spline.

7.6 Running It

In the `java` folder, `javac -d out *.java` compiles both programs; `java -cp out Interp` and `java -cp out Spline` run them. In the `python` folder, `python3 interp.py` and `python3 spline.py`. The outputs are identical in both languages.

```
Newton coefficients f[x0..xk] from the 20 cm marks
  k = 0:  0.000000e+00
  k = 1:  1.239000e+01
  k = 2:  2.055250e-01
  k = 3: -2.201875e-03
  k = 4:  1.223177e-05
  k = 5: -1.223958e-07
  k = 6:  2.170139e-12

depth   table   linear   error  Lagrange   Newton   error
   10    90.03   123.90   33.87     93.62    93.62    3.59
   30   442.21   453.91   11.70    441.62   441.62   -0.59
   50   892.09   895.50    3.40    892.37   892.37    0.28
   70  1369.86  1366.45   -3.41   1369.57  1369.57   -0.29
   90  1819.73  1808.04  -11.69   1820.34  1820.34    0.61
  110  2171.92  2138.05  -33.87   2168.32  2168.32   -3.60
largest error on 0..120 cm: linear 34.49 L, degree 6 5.45 L

Runge's function 1/(1+25x^2) on [-1, 1]: largest error
   n   equally spaced   Chebyshev points
   5           0.4327             0.5559
  10           1.9156             0.1092
  15           2.1076             0.0831
  20          59.8223             0.0153
```

The Newton coefficients $k = 0$ to 3 are those of **Table 7.1**. The last one, 2.17×10^{-12}, deserves a remark: the exact volumes are symmetric about the half-full mark, and for exact data this coefficient would be zero; what we see is the rounding of the printed table to two decimals, amplified by the divided differences. The table of errors shows the two methods side by side at the six 10 cm lines the wall chart does not have. The straight line is off by 33.9 L at 10 cm and 110

cm, by 11.7 L at 30 cm and 90 cm, and by 3.4 L in the middle; the degree-6 polynomial is off by 3.6, 0.6 and 0.3 L at the same places. Lagrange and Newton print the same numbers, as they must: they are two ways of writing the one interpolating polynomial. On the fine grid, the worst straight-line error is 34.5 L and the worst polynomial error 5.45 L. **Figure 7.3** shows both error curves over the whole depth: each straight piece has its own arch, tallest near the ends; the polynomial is almost perfect in the middle and wobbles at the ends.

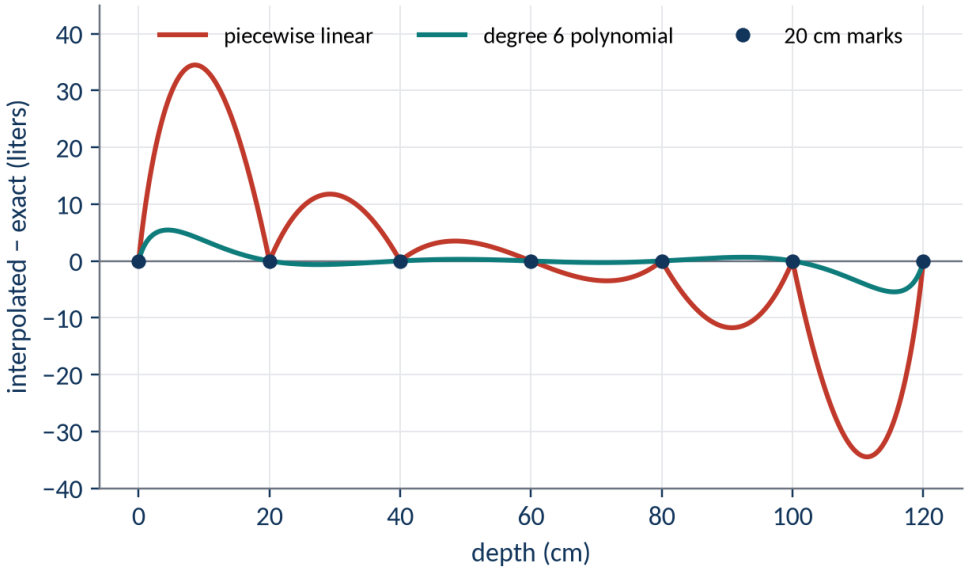


Figure 7.3 Error of piecewise linear and degree-6 polynomial interpolation from the 20 cm marks, measured against the exact tank formula.

Translate the table into Pak Darto's evening check. On a day when the dipstick goes from 88 cm in the morning to 70 cm in the evening, the exact volumes are 1777.7 L and 1369.9 L, so 407.9 L left the tank; straight lines from the wall chart say 1766.8 L and 1366.5 L, a difference of 400.4 L, only 7.5 L short. Near the bottom it is another story. From 28 cm to 12 cm the exact difference is 283.4 L, while the wall chart with straight lines says 412.7 L minus 148.7 L, which is 264.0 L: a gap of 19.4 L between what the pump says it sold and what the tank seems to have lost, although neither the pump nor the tank is at fault. The error does not cancel between the morning and evening readings, because both lie in the part of the tank where the straight line reads too high, and it reads more too high the closer it is to the bottom. That is exactly the kind of unexplained difference the nephew suspected, and it comes from reading the table, not from the fuel.

The Runge table confirms Section 7.3.6: equally spaced errors grow from 0.43 to 59.8 as n goes from 5 to 20, Chebyshev errors shrink from 0.56 to 0.015. (For $n = 5$ the equally spaced nodes happen to be slightly better; the advantage of Chebyshev points is in the trend, which is what matters when you add points.)

Now the splines:

```
curvatures M_i = S''(x_i) at the 20 cm marks
  natural  0.0000  0.5990  0.0705  0.0001 -0.0706 -0.5989  0.0000
  clamped  1.7989  0.1193  0.1904  0.0001 -0.1905 -0.1192 -1.7989

depth   table  natural   error  clamped   error
   10    90.03   108.93   18.90    75.95  -14.08
   30   442.21   437.17   -5.04   446.17    3.96
   50   892.09   893.73    1.64   890.73   -1.36
   70  1369.86  1368.21   -1.65  1371.21    1.35
   90  1819.73  1824.78    5.05  1815.78   -3.95
  110  2171.92  2153.02  -18.90  2186.00   14.08

largest error against the exact tank formula
  marks  ends       0..120 cm   20..100 cm
  20 cm  natural     21.34 L       5.35 L
  20 cm  clamped     14.45 L       4.26 L
  10 cm  natural      7.67 L       0.50 L
  10 cm  clamped      5.08 L       0.41 L
a reading of 47 cm: 821.4 L (exact 821.4 L)
```

The curvatures of the natural spline are 0 at both ends by decree, while the clamped spline finds large curvatures there (1.80), which is what the tank really has: the true V'' is infinite at the ends. Both splines have $M_3 \approx 0$ at 60 cm, where the curve has its inflection. At 50 cm the natural spline is off by 1.64 L and the clamped one by 1.36 L, both better than the straight line (3.4 L), but both worse than the degree-6 polynomial (0.28 L). Near the ends the splines are off by 14 to 19 L, better than the straight line (34 L) but again worse than the polynomial (3.6 L). **Figure 7.4** compares the error curves.

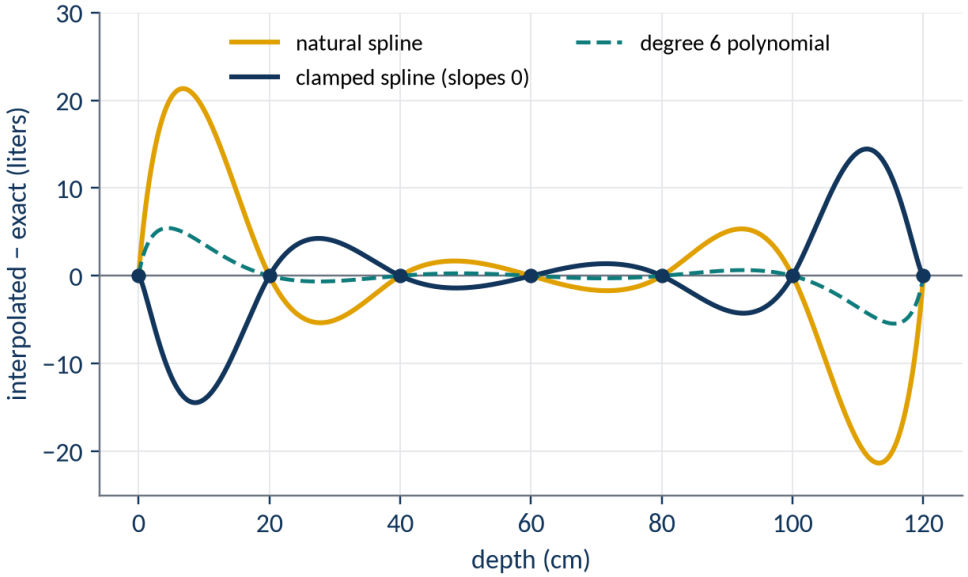


Figure 7.4 Error of the natural spline, the clamped spline (end slopes 0) and the degree-6 polynomial through the 20 cm marks. Near the ends, where the tank's curvature is infinite, the splines suffer most.

Why does the celebrated spline lose to a plain polynomial here? Two reasons, both instructive. First, the tank function is analytic inside the interval and symmetric, and a polynomial of degree 6 fitted to seven exact values captures its overall shape very well; the error bound Equation (7.7) is small in the middle. Second, the spline's accuracy guarantee of Section 7.4.5 assumes a bounded fourth derivative, and near the ends the tank does not even have a bounded second derivative: the volume grows like $h^{3/2}$ near the bottom, and no cubic can follow that. The natural spline makes things worse by forcing $S'' = 0$ exactly where the true curvature is infinite; clamping with the correct slopes helps (21.3 L down to 14.5 L), and halving the spacing to 10 cm helps a lot more

(5.1 L with clamped ends, and only 0.41 L between 20 and 100 cm). With all 13 lines of the table and clamped ends, the reading of 47 cm is 821.4 L, which agrees with the exact formula to the tenth of a liter shown.

> **WARNING:** “Use a spline” is a good default for smooth data, but the word *smooth* carries the weight. Where the data come from a function with a kink, a corner or a square-root singularity, measure the errors before trusting any interpolant, and put more measurements where the function misbehaves.

What would each choice mean for Pak Darto’s evening check? With the wall chart and straight lines, a reading near the bottom or the top can be wrong by up to 34 L, far more than the discrepancies he is trying to explain. With the full 10 cm table and straight lines, the worst error falls to 13.0 L near the ends and 3.6 L between 20 and 100 cm. With the 10 cm table and a clamped spline it falls to 5.1 L near the ends and 0.4 L in the middle, and with the exact formula Equation (7.1) it disappears. A tank that is kept between a quarter and three quarters full, as most working tanks are, lives in the middle range, where every method except the coarse straight line is good enough; the ends are where a careful owner either uses the formula or takes readings with extra care.

7.7 Checking Against the Library

NumPy and SciPy provide every interpolant of this chapter [6], [8]. **Listing 7.5** evaluates our polynomial and splines on the fine grid and compares them with `BarycentricInterpolator` (the Lagrange polynomial in a numerically stable form [4]), `numpy.interp` (piecewise linear) and `CubicSpline` with natural, clamped and default end conditions.

Listing 7.5 `compare_scipy.py`: our interpolants against `numpy.interp`, `BarycentricInterpolator` and `CubicSpline`.

```
"""Check our interpolants against numpy.interp and SciPy's interpolators."""
import numpy as np
import scipy
from scipy.interpolate import BarycentricInterpolator, CubicSpline

import spline
from interp import newton_eval, divided_differences, read_table, tank_liters


def main():
    print(f"NumPy {np.__version__}, SciPy {scipy.__version__}")
    rows = read_table("../data/dipstick.csv")
    xs = [h for h, _ in rows if h % 20 == 0]
    ys = [v for h, v in rows if h % 20 == 0]
    t = np.linspace(0.0, 120.0, 1201)
    exact = np.array([tank_liters(h) for h in t])

    coef = [col[0] for col in divided_differences(xs, ys)]
    ours = np.array([newton_eval(xs, coef, h) for h in t])
    bary = BarycentricInterpolator(xs, ys)(t)
    print(f"degree 6: Newton vs BarycentricInterpolator"
          f" {np.max(np.abs(ours - bary)):.1e}")
    lin = np.interp(t, xs, ys)
    print(f"numpy.interp largest error {np.max(np.abs(lin - exact)):.2f} L")

    for kind, sl in (("natural", None), ("clamped", (0.0, 0.0))):
        m = spline.spline_curvatures(xs, ys, sl)
        mine = np.array([spline.spline_eval(xs, ys, m, h) for h in t])
        bc = "natural" if sl is None else ((1, 0.0), (1, 0.0))
        cs = CubicSpline(xs, ys, bc_type=bc)(t)
        print(f"{kind}: ours vs CubicSpline {np.max(np.abs(mine - cs)):.1e},"
              f" largest error {np.max(np.abs(cs - exact)):.2f} L")
    cs = CubicSpline(xs, ys)(t)                       # default: not-a-knot
    print(f"not-a-knot (SciPy default): largest error"
          f" {np.max(np.abs(cs - exact)):.2f} L")


if __name__ == "__main__":
    main()
```

```
NumPy 2.5.3, SciPy 1.18.1
degree 6: Newton vs BarycentricInterpolator 9.1e-13
numpy.interp largest error 34.49 L
natural: ours vs CubicSpline 4.5e-13, largest error 21.34 L
clamped: ours vs CubicSpline 1.2e-12, largest error 14.45 L
not-a-knot (SciPy default): largest error 7.87 L
```

The degree-6 polynomial agrees with SciPy's barycentric form to 9×10^{-13} L, the straight line with `numpy.interp` exactly (the same 34.49 L worst error), and both splines with `CubicSpline` to about 10^{-12}. SciPy's default *not-a-knot* end condition, which does not pretend to know the curvature at the ends, gives a worst error of 7.87 L: better than both natural (21.34) and clamped (14.45), and still not as good as the polynomial (5.45). In SciPy 1.18 the old Fortran FITPACK routines behind some of the spline functions were ported to C, a modernization that changes nothing for the user [6].

7.8 Making It Better

- **Use what you know.** Pak Darto's tank is a cylinder, and Equation (7.1) gives the volume at any depth without interpolation. Whenever a table comes from a known formula, use the formula. Interpolation is for tables whose origin is unknown or too complicated to model.

- **Put the marks where the function bends.** Chebyshev-like spacing, dense near the ends, suits the tank's square-root behavior. Seven marks at Chebyshev depths (1.5, 13.1, 34.0, 60.0, 86.0, 106.9 and 118.5 cm) give a degree-6 polynomial with a worst error of 3.95 L, against 5.45 L for equal spacing (Exercise 4).
- **Monotone interpolation.** Volumes only grow with depth, but a spline or a high-degree polynomial can dip between two points and make an extra centimeter of fuel "lose" liters. Shape-preserving piecewise cubics (PCHIP, `scipy.interpolate.PchipInterpolator`) choose slopes so that the interpolant is monotone wherever the data are [6].
- **Barycentric evaluation.** The Lagrange form rewritten as $p(x) = \sum_i \frac{w_i}{x-x_i} y_i / \sum_i \frac{w_i}{x-x_i}$, with weights computed once, evaluates in $O(n)$ and is numerically stable; at Chebyshev points it is the method of choice for polynomial interpolation of any degree [4].

7.9 Using It for Real

For your own tank, drum or jerry can with an odd shape, the program works with any table. If you have no table, make one: pour in a known volume at a time (a 20-liter jerry can is convenient), push the dipstick down after each pour and write down the depth. Put the rows in `data/dipstick.csv`. Measure more often near the bottom and the top, where the shape changes fastest. Then compare the program's reading with a few extra pours that you did not put in the table; if the error there is acceptable, the table is good enough. Keep the finished table with the tank's papers, and repeat the pouring test whenever the tank is moved or set down again, because a tank that tilts by a few centimeters from end to end changes every reading, most of all near the bottom. Keep in mind that a fuel tank is a place for care, not experiments: measure with the pump off, never with a light or a phone near the opening, and leave anything unusual (a smell, a stain on the ground, a reading that keeps dropping at night) to the supplier's technician.

The same few lines interpolate any table. A BMKG forecast gives temperatures every three hours; a spline through them gives a smooth curve for the whole day (Exercise 5 shows it may overshoot the highest forecast value slightly). A lender's table gives the monthly instalment for a few loan amounts; linear interpolation between neighboring amounts is usually accurate enough there, because instalments grow almost linearly with the amount. A phone's charge-voltage table lets a small battery monitor show a percentage. In each case: look at the shape of the data, choose linear, a low-degree polynomial or a spline, and measure the error on points you did not use.

7.10 Numerical Methods in Practice: Profitina

Profitina's historical analytics dashboard shows, for each brand, how its AI visibility score has moved week by week, and it offers the history as a CSV export. The score exists only at the scan dates: between two weekly scans there is no measurement. When a chart draws a line between the weekly points, it interpolates, and this chapter says which kind of line is honest. A straight segment claims nothing beyond "somewhere between these two values". A smooth spline through the points looks more professional, but it can overshoot, showing a peak above the highest real score or a dip below the lowest, a value that was never measured. On a dashboard read by a brand owner who is not a numerical analyst, an invented peak is a small lie. The safe choices are straight segments or a monotone, shape-preserving interpolant such as PCHIP, which never creates a maximum or minimum that the data do not have.

The same question appears wherever a quantity is known on a schedule and needed in between. The PROFIT utility coin's emission follows a decaying schedule, and its value in rupiah comes

from a PROFIT to IDR oracle that publishes prices at certain moments. Reading such a schedule or a price between its published points is interpolation, and the choice between a straight line, a polynomial and a spline is a choice about what the reader will be told happened in between.

7.11 Common Mistakes, and How to Avoid Them

- **Raising the degree to improve accuracy.** With equally spaced points, a higher-degree polynomial can be much worse (Runge, Equation (7.8)). Use Chebyshev points if you can choose them, and splines or piecewise low degrees if you cannot.
- **Extrapolating.** Interpolation formulas are designed for points between the nodes. Outside the table the node polynomial in Equation (7.7) grows rapidly and so does the error; a reading of 125 cm from our tank table is meaningless, and should be refused, as `linear` does.
- **Solving the Vandermonde system.** It gives the same polynomial as Lagrange and Newton, but with a badly conditioned matrix. Use the Newton or barycentric form.
- **Trusting natural end conditions.** $S'' = 0$ at the ends is an assumption. If you know the end slopes, clamp; if you know nothing, not-a-knot is usually safer.
- **Forgetting that splines can overshoot.** A spline through monotone data need not be monotone. For quantities that can only grow (cumulative volumes, totals) or must stay in a range (percentages), use PCHIP or straight segments.
- **Unequal intervals in the spline formulas.** Equation (7.12) is written for general h_i. The simplified form $M_{i-1} + 4M_i + M_{i+1}$ holds only for equal spacing; using it on an uneven table gives a wrong spline with no warning.
- **Interpolating noisy measurements.** An interpolant passes through every point, including its errors. If the table comes from rough measurements, such as dipstick readings taken by eye, a curve through every point follows the noise; fit a smooth curve by least squares instead (Chapter 8).
- **Rounding the table, then reading too many digits.** The last Newton coefficient in Section 7.6 is pure rounding noise from a table printed to two decimals. The accuracy of any interpolant is limited by that of its data.

7.12 Chapter Summary

- Interpolation builds a function through tabulated points and reads it in between. The straight line between neighbors (Equation (7.2)) is simple and has error at most $\frac{h^2}{8}\max|f''|$, 34 L near the ends of the tank (Equation (7.1), **Figure 7.1**) and 3.4 L in the middle.
- Through $n + 1$ points there is exactly one polynomial of degree at most n. The Lagrange form (Equation (7.3)) writes it with basis polynomials; the Newton form (Equation (7.4)) with divided differences (Equation (7.5), **Table 7.1**), which extends to new points by one term and evaluates in n multiplications (Equation (7.6)).
- The error of polynomial interpolation is a derivative factor times the node polynomial (Equation (7.7)). Equally spaced nodes make it explode near the ends for Runge's function (Equation (7.8), **Figure 7.2**); Chebyshev points (Equation (7.9)) keep it uniformly small.
- When slopes are known as well as values, Hermite interpolation uses them (Equation (7.10)); at the bottom of the tank it cuts the straight-line error by two thirds.
- A cubic spline is a smooth chain of cubics (Equation (7.11)). Its curvatures solve a strictly diagonally dominant tridiagonal system (Equation (7.12)), solved by the Thomas algorithm

(Equation (7.13)) in linear time. Natural, clamped and not-a-knot ends supply the two missing conditions.

- For the tank, the degree-6 polynomial beats the splines (**Figure 7.3**, **Figure 7.4**), because the tank's curvature is infinite at the ends; clamping the known end slopes and denser marks help most. Measure before you trust.
- The Java and Python programs agree byte for byte, and NumPy and SciPy reproduce every interpolant to about 10^{-12}.

7.13 Exercises

1. Using the wall chart, compute by hand the straight-line reading at 30 cm and the reading of the parabola through the marks at 20, 40 and 60 cm. Compare both with the tank maker's 442.21 L.
2. Extend **Table 7.1** by one row: add the mark at 80 cm (1601.93 L), compute the new divided differences, and check the new top coefficient against the program's value for $k = 4$. Which coefficients of the Newton form change?
3. Use the error bound for linear interpolation to estimate the worst error of the wall chart between 20 and 40 cm, taking $\max|V''|$ on that interval at 20 cm. Compare with the error the program measured at 30 cm.
4. Replace the equally spaced marks by the seven Chebyshev depths on [0,120] from Equation (7.9) (mapped to the interval), with volumes from Equation (7.1). Using the functions of `interp.py`, find the worst error of the degree-6 polynomial on 0 to 120 cm.
5. A BMKG forecast for a town gives these temperatures (an example): 24 °C at 07:00, 29 °C at 10:00, 32 °C at 13:00, 30 °C at 16:00, 27 °C at 19:00 and 25 °C at 22:00. Use `spline_curvatures` and `spline_eval` with natural ends to estimate the temperature at 11:30 and the highest temperature of the day. Compare with linear interpolation. Is the spline's maximum believable?
6. Show that the Newton coefficient $f[x_0, x_1, x_2]$ equals $1/2\, f''$ exactly when f is a quadratic polynomial, and explain why the table's second column for the tank (0.2055 and 0.0734) says that the tank bends more near the bottom than in the middle.
7. Explain, using Equation (7.7), why Chebyshev points improve Runge's function so dramatically but improve the tank only a little (Exercise 4).
8. Use the Hermite midpoint formula Equation (7.10) on the last interval of the wall chart, from 100 cm (2014.15 L) to 120 cm (2261.95 L), with the slopes $V'(100)$ from the formula of Section 7.3.5 and $V'(120) = 0$, to read the volume at 110 cm. Compare with the straight line and with the tank maker's 2171.92 L.

7.14 Solutions to the Exercises

1. Straight line between 20 cm (247.80) and 40 cm (660.02): $247.80 + 0.5 \times 412.22 = 453.91$ L, error $+11.70$ L. The parabola through 20, 40 and 60 cm at $x = 30$: $\ell_0 = \frac{(30-40)(30-60)}{(20-40)(20-60)} = \frac{300}{800} = 0.375$, $\ell_1 = \frac{(30-20)(30-60)}{(40-20)(40-60)} = \frac{-300}{-400} = 0.75$, $\ell_2 = \frac{(30-20)(30-40)}{(60-20)(60-40)} = \frac{-100}{800} = -0.125$. So $p(30) = 0.375 \times 247.80 + 0.75 \times 660.02 - 0.125 \times 1130.97 = 92.925 + 495.015 - 141.371 = 446.57$ L, error $+4.36$ L. The parabola cuts the error by almost two thirds.

2. New entries: first difference $f[60,80] = (1601.93 - 1130.97)/20 = 23.548$; second $f[40,60,80] = (23.548 - 23.5475)/40 = 0.0000125$; third $f[20,\ldots,80] = (0.0000125 - 0.0734125)/60 = -0.00122333$; fourth $f[0,\ldots,80] = (-0.00122333 - (-0.00220188))/80 =$

0.0000122319, which matches the program's 1.223177×10^{-5} to the digits we carried. Only one new coefficient appears; the first four are unchanged, which is the advantage of the Newton form.

3. $V''(h) = 2L(r-h)/\left(1000\sqrt{2rh-h^2}\right) = 0.4 \times 40/\sqrt{2400-400} = 16/44.72 = 0.358$ at $h = 20$. The bound is $\frac{20^2}{8} \times 0.358 = 17.9$ L. The measured error at 30 cm is 11.70 L, below the bound, as it must be; the bound is pessimistic because V'' at 20 cm is larger than its values in the rest of the interval.

4. The Chebyshev depths are $60 - 60\cos\frac{(2i+1)\pi}{14}$ for $i = 0, \ldots, 6$: 1.5, 13.1, 34.0, 60.0, 86.0, 106.9 and 118.5 cm. With the exact volumes at these depths and `lagrange` evaluated on a 1 mm grid, the worst error is 3.95 L, against 5.45 L with the equally spaced marks. It helps, because the nodes move toward the ends where the tank misbehaves, but it does not transform the result.

5. The natural spline gives 31.02 °C at 11:30, linear interpolation 30.5 °C. Evaluating the spline every 0.01 hour, the highest value is 32.01 °C at about 13:11. The spline's maximum lies just above the highest forecast value and a few minutes after it; that is believable, since the real maximum of a day's temperature usually comes in the early afternoon and need not fall exactly on a forecast hour. But a forecast is itself uncertain by a degree or more, so the 0.01 °C of overshoot carries no information. The curvatures are $M = (0, -0.134, -0.797, -0.010, 0.169, 0)$ per hour squared: the curve bends down most strongly around 13:00.

6. If $f(x) = a + bx + cx^2$, then $f[x_0, x_1] = b + c(x_0 + x_1)$ and $f[x_1, x_2] = b + c(x_1 + x_2)$, so $f[x_0, x_1, x_2] = c(x_2 - x_0)/(x_2 - x_0) = c = f''/2$. In general, $f[x_0, x_1, x_2] = f''(\xi)/2$ for some ξ in the interval. The tank's second divided differences 0.2055 (from 0 to 40 cm) and 0.0734 (from 20 to 60 cm) therefore estimate $V''/2$ as about 0.41 and 0.15 liters per cm^2 on those stretches: the bottom bends almost three times more strongly than the middle.

7. The error is $f^{(n+1)}(\xi)/(n+1)!$ times the node polynomial. Chebyshev points minimize the node polynomial: they shrink the second factor uniformly. For Runge's function the trouble was the node polynomial's growth near the ends with equal spacing, combined with derivatives that grow like $(n+1)!\, 5^{n+1}$; Chebyshev points remove the first problem and turn divergence into convergence. For the tank, the trouble is the first factor: the derivatives of V are infinite at the ends, so no choice of nodes can make the error formula small there. Chebyshev spacing helps a little by putting nodes closer to the singular ends, but the cure for the tank is either the exact formula or many more measurements near the ends.

8. $V'(100) = 0.4\sqrt{2 \times 60 \times 100 - 100^2} = 0.4\sqrt{2000} = 17.89$ L/cm and $V'(120) = 0$. The straight line gives $(2014.15 + 2261.95)/2 = 2138.05$ L, 33.87 L too low. Hermite adds $\frac{20}{8}(17.89 - 0) = 44.72$ L and gives 2182.77 L, 10.85 L too high. By the symmetry of the tank this is the mirror image of the reading at 10 cm in Section 7.3.7: the same error sizes, with opposite signs.

References

[1] S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers*, 8th ed. New York, NY, USA: McGraw-Hill Education, 2021.

[2] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical Analysis*, 10th ed. Boston, MA, USA: Cengage Learning, 2016.

[3] C. Runge, "Über empirische Funktionen und die Interpolation zwischen äquidistanten Ordinaten," *Zeitschrift für Mathematik und Physik*, vol. 46, pp. 224–243, 1901.

[4] L. N. Trefethen, *Approximation Theory and Approximation Practice*, extended ed. Philadelphia, PA, USA: SIAM, 2019.

[5] C. de Boor, *A Practical Guide to Splines*, rev. ed. New York, NY, USA: Springer, 2001.

[6] SciPy Developers, "SciPy 1.18.0 Release Notes," Jun. 19, 2026, and "scipy.interpolate.CubicSpline," SciPy v1.18 Manual, accessed Sep. 2026. [Online]. Available: https://docs.scipy.org/doc/scipy/release/1.18.0-notes.html and https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.CubicSpline.html

[7] L. H. Thomas, "Elliptic problems in linear difference equations over a network," Watson Scientific Computing Laboratory, Columbia University, New York, NY, USA, 1949.

[8] NumPy Developers, "NumPy 2.5.0 Release Notes," Jun. 21, 2026, and "numpy 2.5.3" on the Python Package Index, Sep. 6, 2026. [Online]. Available: https://numpy.org/doc/2.5/release/2.5.0-notes.html and https://pypi.org/project/numpy/

Chapter 8

Curve Fitting: Least Squares Regression

Chapter 7 drew curves that pass exactly through every point of a table. That is right when the table is exact, like a tank maker's. Most data are not: they are measurements from real days, and every day has its own small accidents. A curve forced through every one of them follows the accidents instead of the pattern. This chapter asks for less and gets more: a simple curve that passes *near* all the points, chosen so that the sum of the squared misses is as small as possible. The running example is the notebook of an ojek driver who wants to understand his fuel. By the end of the chapter you will be able to fit a straight line by least squares by hand and derive its formulas; measure the quality of a fit with residuals, R^2 and the standard error, and put an honest uncertainty on a slope; fit several predictors at once through the design matrix and the QR method of Chapter 6; fit polynomials and exponential curves by making them linear in their coefficients; recognize overfitting and choose a model by testing it on data it has not seen; build all of it in Java and Python; and check it against NumPy and SciPy.

8.1 Mas Dimas's Notebook

Mas Dimas drives an ojek, a motorcycle taxi, taking rides and food deliveries through a ride-hailing app in a mid-sized city. His motorbike is his office and fuel is his biggest daily cost. For forty working days he has kept a small notebook: every evening he writes down the distance the app says he rode, the number of trips he made, and how many liters of fuel he bought to fill the tank back up. On some days he also notes the average speed the app reports, and every two months he looks at the battery health his phone shows, because the phone that runs the app is his second tool of trade.

He has four questions. How much fuel does one kilometer really cost him, so that he can judge whether a long ride at a fixed fare is worth taking? How much fuel should he plan for tomorrow, when the app's schedule suggests about 120 km and 17 trips? At what average speed does his bike go furthest on a liter? And when will the phone's battery drop to 80% of its original capacity, the level at which many repair shops suggest a replacement?

None of these questions has an exact answer in the notebook. On two days with the same distance he bought different amounts of fuel, because of traffic, wind, a longer wait with the engine running, or a tank that was filled to a slightly different level. The points scatter. What he needs is the *trend* under the scatter, with an honest idea of how far to trust it. **Figure 8.1** shows the forty days: distance on the horizontal axis, liters on the vertical. The points clearly lie close to a straight line. Which line?

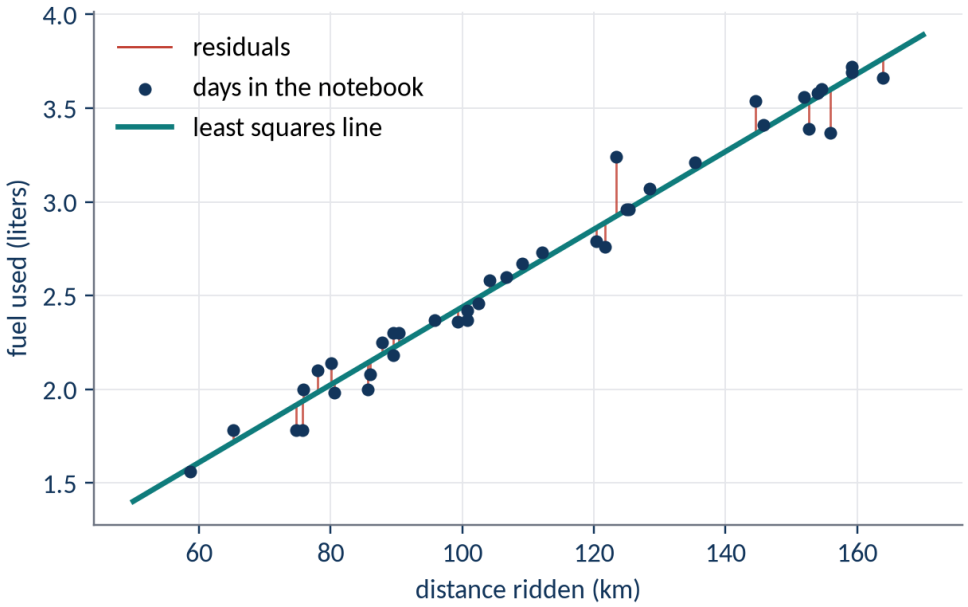


Figure 8.1 Forty days of Mas Dimas's notebook: fuel used against distance ridden, the least squares line, and the residuals, the vertical misses that the line makes as small as possible in the sum of squares.

8.2 The Idea in Plain Words

8.2.1 Misses, and why we square them

Suppose we try the line $y = a + bx$, where x is the distance and y the fuel. On day i the line predicts $a + bx_i$, and the notebook says y_i. The difference

$$e_i = y_i - (a + bx_i)$$

is the *residual* of day i: positive when the line runs below the point, negative when it runs above. A good line makes the residuals small, all of them at once. We need one number that measures "all at once". The sum of the residuals is useless, because positive and negative misses cancel: a line far above half the points and far below the other half can have a sum of zero. The sum of absolute values works but is awkward to minimize (it has corners). Adrien-Marie Legendre in 1805, and Carl Friedrich Gauss, who said he had used it since 1795 and who in 1801 used it to find the lost asteroid Ceres again, chose the **sum of squared residuals** [1], [2]:

$$\mathrm{SSE}(a, b) = \sum_{i=1}^{n} (y_i - a - bx_i)^2. \tag{8.1}$$

Squares are always positive, so nothing cancels; big misses cost much more than small ones, so the line is pulled toward being fair to every point; and, above all, SSE is a smooth bowl-shaped function of a and b whose lowest point is found by solving linear equations. Gauss later showed that when the measurement errors are independent with equal spread, no other straight-line recipe that is linear in the data and unbiased has smaller variance (the Gauss-Markov theorem) [3].

8.2.2 Five days by hand

Take the first five days of the notebook. The recipe, which Section 8.3 derives, is: compute the mean distance $\bar{x}$ and the mean fuel $\bar{y}$, then two sums, S_{xx} of the squared deviations of x and S_{xy} of the products of deviations, and set $b = S_{xy}/S_{xx}$ and $a = \bar{y} - b\bar{x}$. **Table 8.1** lays out the arithmetic. The means are $\bar{x} = 537.8/5 = 107.56$ km and $\bar{y} = 12.96/5 = 2.592$ L.

Table 8.1 The least squares line through the first five days, by hand: deviations from the means, their squares and their products.

Day	x_i (km)	y_i (L)	$x_i - x$	$y_i - y$	$(x_i - x)^2$	$(x_i - x)(y_i - y)$
1	74.7	1.78	−32.86	−0.812	1079.78	26.6823
2	95.8	2.37	−11.76	−0.222	138.30	2.6107
3	154.6	3.60	47.04	1.008	2212.76	47.4163
4	87.8	2.25	−19.76	−0.342	390.46	6.7579
5	124.9	2.96	17.34	0.368	300.68	6.3811
sum	537.8	12.96	0	0	$S_{xx} = 4121.97$	$S_{xy} = 89.848$

So $b = 89.848/4121.97 = 0.021797$ L/km and $a = 2.592 - 0.021797 \times 107.56 = 0.2475$ L. The line says that each kilometer costs about 0.022 L and that a day on the road costs about a quarter of a liter before the first kilometer, which is the fuel burned idling, waiting for orders and starting up. The five residuals are -0.096, 0.034, -0.017, 0.089 and -0.010 L; their sum is zero, as Section 8.3 will prove it must be. With all forty days the program of Section 8.8 finds a slightly different line, $0.3652 + 0.020737x$: five days are a small sample, and the trend they suggest moves when more days arrive. How much it can move is part of the answer, and Section 8.3.4 measures it.

8.3 The Straight Line Precisely

8.3.1 The normal equations for a line

SSE in Equation (8.1) is a quadratic function of the two unknowns a and b. At its minimum both partial derivatives vanish:

$$\frac{\partial\,\mathrm{SSE}}{\partial a} = -2\sum_i (y_i - a - bx_i) = 0, \qquad \frac{\partial\,\mathrm{SSE}}{\partial b} = -2\sum_i x_i\,(y_i - a - bx_i) = 0.$$

Dividing by -2 and rearranging gives two linear equations, the **normal equations** of the straight line:

$$\begin{aligned} n\,a + \left(\sum x_i\right) b &= \sum y_i \\ \left(\sum x_i\right) a + \left(\sum x_i^2\right) b &= \sum x_i y_i \end{aligned} \tag{8.2}$$

They are the 2×2 case of the normal equations $A^{\mathsf{T}}A\,x = A^{\mathsf{T}}b$ of Chapter 6, with a column of ones and a column of distances [4], [5]. Dividing the first equation by n gives $a = \bar{y} - b\bar{x}$: *the line passes through the point of means* $(\bar{x}, \bar{y})$. Substituting this into the second equation and simplifying (subtract $n\bar{x}$ times the first equation from the second) leaves

$$b = \frac{S_{xy}}{S_{xx}}, \quad a = \bar{y} - b\,\bar{x}, \qquad S_{xx} = \sum_i (x_i - \bar{x})^2\,, \quad S_{xy} = \sum_i (x_i - \bar{x})\,(y_i - \bar{y}). \tag{8.3}$$

The centered sums are also the numerically sensible way to compute: the textbook shortcut $S_{xx} = \sum x_i^2 - n\bar{x}^2$ subtracts two large, nearly equal numbers and can lose most of its digits when the x_i are large and close together (the cancellation of Chapter 1). Our programs compute the deviations first.

8.3.2 Two facts about residuals

The first normal equation says $\sum_i e_i = 0$: the residuals of a least squares line with an intercept always add up to zero. The second says $\sum_i x_i\, e_i = 0$: the residuals are uncorrelated with x, perpendicular to the column of distances in the sense of Chapter 6. The program's residuals for all forty days add up to 7×10^{-15}, zero up to rounding. These are not properties of the data; they are properties of the method, and they are also a check: if your residuals do not add up to zero, your line is not the least squares line.

8.3.3 How good is the fit? R^2 and the standard error

Two numbers summarize a fit. The first compares the fit with the crudest possible model, "every day uses the average amount". Its misses are $y_i - \bar{y}$, and their sum of squares is the *total sum of squares* $\text{SST} = \sum (y_i - \bar{y})^2$. The line's misses have sum of squares SSE. The **coefficient of determination** is

$$R^2 = 1 - \frac{\text{SSE}}{\text{SST}}, \tag{8.4}$$

the fraction of the day-to-day variation in fuel that the line explains. For the forty days, SSE $= 0.387$ and SST $= 15.91$, so $R^2 = 0.9757$: distance explains about 97.6% of the variation. R^2 is 1 for a perfect fit and 0 for a line no better than the average.

The second number speaks in the units of the data. The **standard error of the estimate**

$$s_e = \sqrt{\frac{\text{SSE}}{n-2}} \tag{8.5}$$

is a typical size of a residual. The divisor is $n - 2$, not n, because two numbers, a and b, were fitted from the same data, and a fit with n points and 2 parameters has only $n - 2$ degrees of freedom left to show its misses; with two points the line passes through both and SSE is zero whatever the truth. For the notebook $s_e = 0.101$ L: a typical day's fuel differs from the line by about a tenth of a liter.

8.3.4 How sure is the slope?

If the notebook were kept for another forty days, the fitted slope would come out slightly different. Under the standard assumptions (independent errors with equal spread around a true line), the slope's standard error is

$$\mathrm{SE}(b) = \frac{s_e}{\sqrt{S_{xx}}}, \tag{8.6}$$

and the true slope lies within about two standard errors of the fitted one with 95% confidence (for 38 degrees of freedom the exact factor from Student's t distribution is 2.02) [3]. For Mas Dimas, $\mathrm{SE}(b) = 0.101/\sqrt{36092.6} = 0.000531$ L/km, and the slope is 0.020737 ± 0.001062 L/km. Equation (8.6) also says how to get a sharper slope: more days (a smaller s_e is not in your hands, a larger S_{xx} is), and above all days that are spread out, some short and some long, because S_{xx} grows with the spread of x. Forty days that all had 110 km would say almost nothing about the cost per kilometer.

8.3.5 Predicting one new day

The slope's uncertainty is about the *trend*. A question about *one* new day adds the day's own scatter. For a new distance x_0, the least squares prediction is $\hat{y}_0 = a + bx_0$, and a 95% *prediction interval* for the fuel of that single day is [3]

$$\hat{y}_0 \pm t\, s_e \sqrt{1 + \frac{1}{n} + \frac{(x_0 - \bar{x})^2}{S_{xx}}}, \tag{8.7}$$

with $t = 2.02$ for 38 degrees of freedom. The three terms under the root are the scatter of one day (the 1), the uncertainty of the line's height ($1/n$) and the uncertainty of its slope, which grows with the distance of x_0 from the mean. For $x_0 = 150$ km, $1 + 1/40 + (150 - 111.75)^2/36092.6 = 1.0655$, the root is 1.032, and the interval is $3.476 \pm 2.02 \times 0.1009 \times 1.032 = 3.476 \pm 0.211$ L. Almost all of it comes from the day's own scatter; the line itself is well known. Far outside the data, the last term would take over, which is the formula's way of warning against extrapolation.

8.3.6 Looking at the residuals

The numbers of Section 8.3.3 summarize a fit; the residuals themselves tell you whether to believe it. Plot them against x (or against the fitted values). If the model is right, they form a shapeless band around zero of roughly constant width. Three patterns are warnings [4], [3]. A *curve* in the residuals (positive at both ends, negative in the middle, or the reverse) means the data bend and a straight line is the wrong model; Section 8.5 fixes that. A *funnel*, residuals that spread wider as x grows, means the scatter is not the same everywhere, and the uncertainty of Equation (8.6) is then too optimistic for large x; weighting (Section 8.10) or a log transform helps. A *single far point*, like day 17 in Section 8.8, asks for a look at the notebook. For Mas Dimas's forty days the residuals of **Figure 8.1** show none of these patterns: the vertical red segments are short and long above and below the line at every distance, which is what a good straight-line model looks like.

> **KEY INSIGHT:** A fitted number without its uncertainty is half an answer. The slope 0.0207 ± 0.0011 L/km says both "about 0.021 L per km" and "do not base a decision on the fourth digit".

8.4 More Predictors: the Design Matrix

8.4.1 Fuel from distance and trips

Distance is not the only thing that burns fuel. Each trip means stopping, waiting for the customer or the food, and starting again. The notebook has the number of trips, so we can try

$$y \approx \beta_0 + \beta_1 x_1 + \beta_2 x_2,$$

with x_1 the distance and x_2 the trips. Write one row per day, $[1,\ x_{i1},\ x_{i2}]$, and stack the rows into the **design matrix** X (40 rows, 3 columns); stack the fuel into y and the coefficients into β. The model for all days at once is

$$y \approx X\beta, \tag{8.8}$$

and least squares chooses β to minimize $\| y - X\beta \|^2$. This is exactly the problem of Section 6.4 with new names, and all its tools apply: the normal equations

$$X^{\mathsf{T}} X\, \beta = X^{\mathsf{T}} y, \tag{8.9}$$

solved by Cholesky, or, better, Householder QR, which does not square the condition number. Equation (8.2) is the special case with the columns 1 and x. The word "linear" in *linear least squares* refers to the coefficients β, not to the columns: any column that can be computed from the data is allowed, as the next section exploits.

8.4.2 Why QR, even here

The fuel design matrix has a column of ones, a column of distances around 110 and a column of trips around 15. Its condition number is about 480, and that of $X^{\mathsf{T}}X$ about 230,000, its square, as Chapter 6 showed that it must be. Here even the normal equations would keep ten or eleven digits, more than the data deserve. But the habit matters more than this case: add a column of squared distances (values around 12,000) and the condition number of $X^{\mathsf{T}}X$ jumps by several orders of magnitude. QR costs a little more and never squares anything, so our programs always use it. Centering and scaling the columns helps further: with each predictor replaced by its deviation from the mean divided by its spread, the condition number of this design matrix falls from 480 to 2.2.

8.4.3 What the coefficients mean

The program finds liters $= 0.2654 + 0.018568\,\text{km} + 0.02285\,\text{trips}$. The coefficient of the trips says that each extra trip, *at the same distance*, costs about 0.023 L; the coefficient of the distance has dropped from 0.0207 to 0.0186 L/km, because part of what the simple line attributed to kilometers was really due to the trips that come with them. The two predictors are correlated (in this notebook the correlation between distance and trips is 0.66: longer days have more trips), and a simple line cannot tell their effects apart; the multiple fit can, as long as the correlation is not too close to 1. When two columns of X are almost proportional, the matrix $X^{\mathsf{T}}X$ is nearly singular, the coefficients become very uncertain and can swing wildly, and the problem is called *collinear* [3]. R^2 rises from 0.9757 to 0.9894 and the standard error falls from 0.101 to 0.068 L (now computed with $n - 3$ degrees of freedom, since three coefficients were fitted). For tomorrow, 120 km and 17 trips, the model plans 2.88 L.

8.5 Curves That Are Linear in Their Coefficients

8.5.1 Polynomials

Mas Dimas's third question is about speed. At walking pace in traffic a motorbike burns fuel going nowhere; at high speed the air pushes back harder and harder. Fuel economy (km/L) therefore

rises with average speed, peaks, and falls: a curve, not a line. **Figure 8.3** shows 36 days with a recorded average speed. The simplest curve with one peak is a parabola, and a polynomial model of degree d,

$$y \approx \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_d x^d, \tag{8.10}$$

is *linear in the coefficients*: its design matrix has the columns $1, x, x^2, \ldots, x^d$. So polynomial fitting is linear least squares, and the program of Section 8.4 fits it with a different design matrix.

There is one trap, which Chapter 6 measured. With speeds between 15 and 70 km/h, the column x^8 has entries up to $70^8 \approx 5.8 \times 10^{14}$, and the columns of powers are nearly parallel: the condition number is astronomical. The cure is cheap: **center and scale** the variable first. Our programs use $t = (\text{speed} - 45)/25$, which lies roughly between -1 and 1, and fit polynomials in t. The fitted curve is the same function of speed; only its coefficients are written differently, and the computation keeps its digits.

8.5.2 The best speed

For the degree-2 fit $y = c_0 + c_1 t + c_2 t^2$ with $c_2 < 0$, the peak is where the derivative $c_1 + 2c_2 t$ vanishes:

$$t^* = -\frac{c_1}{2c_2}, \qquad \text{speed}^* = 45 + 25\, t^*. \tag{8.11}$$

The program finds $c_0 = 54.653$, $c_1 = 0.535$ and $c_2 = -8.459$, so $t^* = 0.535/16.918 = 0.0316$ and the best average speed is $45 + 25 \times 0.0316 = 45.8$ km/h, where the bike gives about 54.7 km/L. Chapter 13 returns to finding the top of a curve when there is no formula for it.

8.5.3 Exponentials through logarithms

A phone battery loses capacity roughly in proportion to what it has left: a model $H(t) = Ce^{kt}$, with H the health in percent, t the months of use, and $k < 0$. This is not linear in C and k. But its logarithm is:

$$\ln H = \ln C + k\, t, \tag{8.12}$$

a straight line in t with intercept $\ln C$ and slope k. Fit the line to the points $(t_i, \ln H_i)$ with Equation (8.3) and transform back: $C = e^{\text{intercept}}$. For Mas Dimas's thirteen readings (every two months for two years, **Figure 8.4**) this gives $H = 100.03\, e^{-0.00954\, t}$, and the health reaches 80% when $100.03\, e^{-0.00954\, t} = 80$, that is after $t = \ln(80/100.03)/(-0.00954) = 23.4$ months. The same trick fits power laws, $y = Cx^k$ becomes $\ln y = \ln C + k \ln x$, and many growth and decay curves.

The trick has a price: least squares on $\ln H$ minimizes squared *relative* misses rather than squared misses in percent, so it gives the early, larger readings a little less weight than a direct fit would. Section 8.9 fits the curve directly with SciPy's nonlinear `curve_fit` and gets $100.04\, e^{-0.00956\, t}$, practically the same here, because the readings are close to the curve. When the data are noisy, the two can differ, and the direct fit is the one that answers the question as asked.

8.6 Overfitting, and How to Choose a Model

A polynomial of higher degree always fits the data at hand at least as well as a lower one: adding a column to X can only lower SSE, because the old solution is still available (with the new coefficient zero). R^2 therefore rises with every degree, and a degree-35 polynomial through 36 points would reach $R^2 = 1$. That does not make it a better model. It has learned the accidents of those 36 days, and on the 37th day it is likely to be badly wrong. This is **overfitting**.

The honest test is to judge a model on data it has not seen. Split the days into two groups: fit on one (the *training* days) and measure the error on the other (the *held-back* days). The measure is the **root mean square error**

$$\text{RMSE} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2}, \tag{8.13}$$

the typical miss in the units of y, where $\hat{y}_i$ is the model's prediction for day i. The polynomial program holds back every third day (days 3, 6, 9, ..., 36), fits degrees 1 to 8 on the other 24, and measures RMSE on both groups. **Figure 8.2** shows the typical picture of every model-choosing problem: the error on the fitting days keeps falling with the degree; the error on the held-back days falls, reaches a minimum, and then rises. The best model is at the bottom of the second curve, not the first.

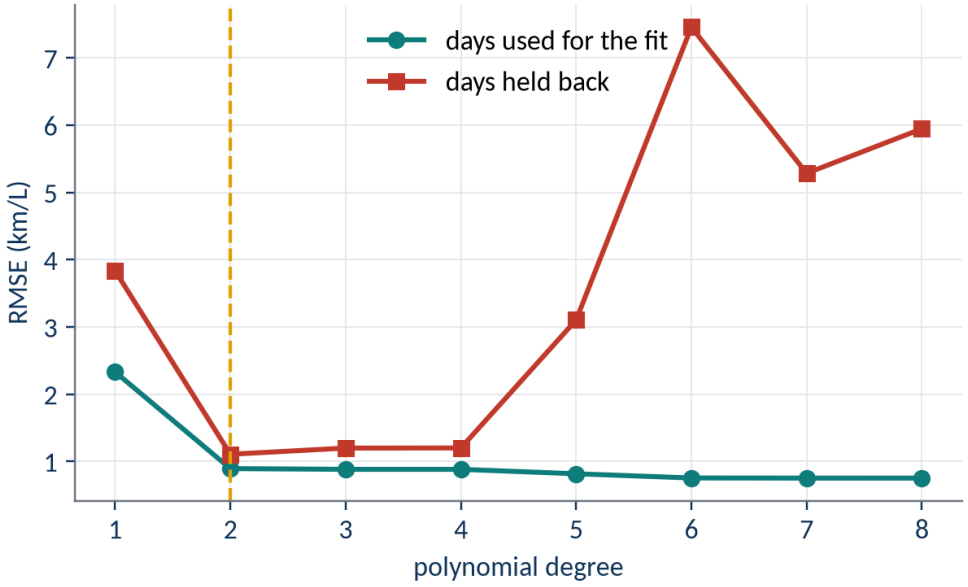


Figure 8.2 RMSE of polynomial fits of degree 1 to 8 on the 24 fitting days and on the 12 held-back days. The fitting error only falls; the held-back error is smallest at degree 2 and grows sharply from degree 5.

Why does the held-back error rise again? A flexible model has two sources of error on new data. Its *bias* is the part of the truth it cannot represent: a straight line cannot bend, however many days it sees, so it misses the peak on every new day. Its *variance* is how much its prediction would change if it had been fitted on a different set of days: a degree-8 polynomial reshapes itself around every accident, so a new day meets a curve that was bent for someone else. Raising the degree lowers the bias and raises the variance, and the held-back error is smallest where their sum is smallest [6]. Degree 2 has almost no bias for a curve with one peak and very little variance.

One split can be lucky or unlucky. *Cross-validation* rotates the held-back group (the first third, then the second, then the third) and averages the held-back errors; Exercise 6 does this by hand for the three thirds of the notebook. The most thorough version holds back one day at a time: fit on 35 days, predict the 36th, and repeat for every day (*leave-one-out* cross-validation). For the speed data it gives a held-back RMSE of 3.13 km/L for degree 1, 1.017 for degree 2, 1.021 for degree 3, 1.049 for degree 4, 1.33 for degree 5 and 3.07 for degree 6: again a minimum at degree 2, with degree 3 a close second that does not earn its extra coefficient. In machine learning, the same idea protects every model from believing its own training data, as Chapter 14 shows.

8.7 Building It in Java and Python

Three files per language. `lsq` holds the shared tools: reading a CSV file, least squares by Householder QR (the function of Chapter 6, now called `lstsq`), prediction and R^2. `line_fit` answers Mas Dimas's first, second and fourth questions: the straight line by Equation (8.3), its R^2, standard error and slope uncertainty, the two-predictor model by QR, and the battery curve by logarithms. `poly_fit` answers the third: polynomials of degree 1 to 8 judged on held-back days, and the best speed from the degree-2 fit.

Listing 8.1 is the Python helper module. The `lstsq` function is the Householder QR of Section 6.5, unchanged except that it returns only the coefficients.

Listing 8.1 `lsq.py`: CSV reading, least squares by Householder QR, prediction and R^2, shared by the programs of this chapter.

```
"""Shared helpers: plain sums, CSV reading and least squares by
Householder QR (the method of Chapter 6)."""
import math


def total(values):
    """Plain left-to-right sum (Python 3.12+ sum() compensates rounding,
    which would make our digits differ from the Java version)."""
    s = 0.0
    for v in values:
        s += v
    return s


def read_csv(path):
    """Header names and the rows as lists of floats."""
    with open(path, encoding="utf-8") as f:
        names = f.readline().strip().split(",")
        rows = [[float(v) for v in line.strip().split(",")]
                for line in f if line.strip()]
    return names, rows


def lstsq(a, b):
    """Minimize ||b - A x|| with Householder reflections; returns x."""
    m, n = len(a), len(a[0])
    r = [row[:] for row in a]
    c = b[:]
    for k in range(n):
        norm = math.sqrt(total(r[i][k] * r[i][k] for i in range(k, m)))
        alpha = -norm if r[k][k] >= 0.0 else norm
        v = [0.0] * m
        for i in range(k, m):
            v[i] = r[i][k]
        v[k] -= alpha
        vv = total(v[i] * v[i] for i in range(k, m))
        if vv == 0.0:
            continue
        for j in range(k, n):
            f = 2.0 * total(v[i] * r[i][j] for i in range(k, m)) / vv
            for i in range(k, m):
                r[i][j] -= f * v[i]
        f = 2.0 * total(v[i] * c[i] for i in range(k, m)) / vv
        for i in range(k, m):
            c[i] -= f * v[i]
    x = [0.0] * n
```

Listing 8.1 (Continued) *lsq.py: CSV reading, least squares by Householder QR, prediction and R^2, shared by the programs of this chapter.*

```
    for i in range(n - 1, -1, -1):
        s = total(r[i][j] * x[j] for j in range(i + 1, n))
        x[i] = (c[i] - s) / r[i][i]
    return x


def predict(a, x):
    return [total(row[j] * x[j] for j in range(len(x))) for row in a]


def r_squared(y, fitted):
    """1 - (residual sum of squares) / (total sum of squares)."""
    mean = total(y) / len(y)
    sse = total((u - v) * (u - v) for u, v in zip(y, fitted))
    sst = total((u - mean) * (u - mean) for u in y)
    return 1.0 - sse / sst
```

Listing 8.2 is the Java helper class. It also carries the number formatting that makes the Java output identical to Python's, and a `signed` helper for numbers printed with an explicit plus sign.

Listing 8.2 `Lsq.java`: the shared helpers in Java.

```
import java.io.IOException;
import java.math.BigDecimal;
import java.math.MathContext;
import java.math.RoundingMode;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;

/** Shared helpers: CSV reading, least squares by Householder QR (the
 *  method of Chapter 6), R^2, and number formatting that matches Python. */
public class Lsq {
    /** The rows of a CSV file with a header line, as numbers. */
    static double[][] readCsv(String path) throws IOException {
        List<String> lines = Files.readAllLines(Path.of(path));
        double[][] rows = new double[lines.size() - 1][];
        for (int i = 1; i < lines.size(); i++) {
            String[] p = lines.get(i).strip().split(",");
            rows[i - 1] = new double[p.length];
            for (int j = 0; j < p.length; j++) {
                rows[i - 1][j] = Double.parseDouble(p[j]);
            }
        }
        return rows;
    }

    /** Minimize ||b - A x|| with Householder reflections; returns x. */
    static double[] lstsq(double[][] a, double[] b) {
        int m = a.length, n = a[0].length;
        double[][] r = new double[m][];
        for (int i = 0; i < m; i++) r[i] = a[i].clone();
        double[] c = b.clone();
        for (int k = 0; k < n; k++) {
            double ss = 0;
            for (int i = k; i < m; i++) ss += r[i][k] * r[i][k];
            double norm = Math.sqrt(ss);
            double alpha = r[k][k] >= 0.0 ? -norm : norm;
            double[] v = new double[m];
            for (int i = k; i < m; i++) v[i] = r[i][k];
            v[k] -= alpha;
            double vv = 0;
            for (int i = k; i < m; i++) vv += v[i] * v[i];
            if (vv == 0.0) continue;
            for (int j = k; j < n; j++) {
                double s = 0;
                for (int i = k; i < m; i++) s += v[i] * r[i][j];
                double f = 2.0 * s / vv;
```

Listing 8.2 (Continued) *Lsq.java: the shared helpers in Java.*

```
                for (int i = k; i < m; i++) r[i][j] -= f * v[i];
            }
            double s = 0;
            for (int i = k; i < m; i++) s += v[i] * c[i];
            double f = 2.0 * s / vv;
            for (int i = k; i < m; i++) c[i] -= f * v[i];
        }
        double[] x = new double[n];
        for (int i = n - 1; i >= 0; i--) {
            double s = 0;
            for (int j = i + 1; j < n; j++) s += r[i][j] * x[j];
            x[i] = (c[i] - s) / r[i][i];
        }
        return x;
    }

    static double[] predict(double[][] a, double[] x) {
        double[] out = new double[a.length];
        for (int i = 0; i < a.length; i++) {
            double s = 0;
            for (int j = 0; j < x.length; j++) s += a[i][j] * x[j];
            out[i] = s;
        }
        return out;
    }

    static double total(double[] v) {
        double s = 0;
        for (double x : v) s += x;
        return s;
    }

    /** 1 - (residual sum of squares) / (total sum of squares). */
    static double rSquared(double[] y, double[] fitted) {
        double mean = total(y) / y.length;
        double sse = 0, sst = 0;
        for (int i = 0; i < y.length; i++) {
            sse += (y[i] - fitted[i]) * (y[i] - fitted[i]);
        }
        for (double u : y) sst += (u - mean) * (u - mean);
        return 1.0 - sse / sst;
    }
```

Listing 8.2 (Continued) *Lsq.java: the shared helpers in Java.*

```
    static String signed(double x, int d) {
        String s = f(x, d);
        return s.startsWith("-") ? s : "+" + s;
    }

    // ---- number formatting that matches Python's f"{x:w.df}" and ".de"
    static String f(double x, int d) {
        String s = new BigDecimal(x).setScale(d, RoundingMode.HALF_EVEN)
                                    .toPlainString();
        boolean neg = x < 0 || (x == 0 && 1 / x < 0);
        return neg && !s.startsWith("-") ? "-" + s : s;
    }

    static String f(double x, int d, int w) { return pad(f(x, d), w); }

    static String e(double x, int d) {
        if (x == 0) return "0." + "0".repeat(d) + "e+00";
        BigDecimal b = new BigDecimal(x).round(
                new MathContext(d + 1, RoundingMode.HALF_EVEN));
        int exp = b.precision() - b.scale() - 1;
        String digits = b.unscaledValue().abs().toString();
        digits = (digits + "0".repeat(d + 1)).substring(0, d + 1);
        String ex = (exp < 0 ? "-" : "+") + (Math.abs(exp) < 10 ? "0" : "")
                    + Math.abs(exp);
        return (x < 0 ? "-" : "") + digits.charAt(0) + "." + digits.substring(1)
               + "e" + ex;
    }

    static String pad(String s, int w) {
        return " ".repeat(Math.max(0, w - s.length())) + s;
    }

    static String padRight(String s, int w) {
        return s + " ".repeat(Math.max(0, w - s.length()));
    }
}
```

Listing 8.3 computes the straight line from the centered sums, exactly as in **Table 8.1**, then builds the three-column design matrix and calls `lstsq`, then fits the battery line on logarithms.

Listing 8.3 `line_fit.py`: the least squares line with R^2, standard error and slope uncertainty; two predictors by QR; the exponential battery curve.

```
"""Mas Dimas's fuel notebook: a straight line by the least squares
formulas, then two predictors by QR, then a battery curve by logarithms."""
import math

from lsq import lstsq, predict, r_squared, read_csv, total


def main():
    _, rows = read_csv("../data/fuel_log.csv")
    km = [r[1] for r in rows]
    trips = [r[2] for r in rows]
    liters = [r[3] for r in rows]
    n = len(rows)
    mx, my = total(km) / n, total(liters) / n
    sxx = total((x - mx) * (x - mx) for x in km)
    sxy = total((x - mx) * (y - my) for x, y in zip(km, liters))
    b = sxy / sxx
    a = my - b * mx
    fitted = [a + b * x for x in km]
    res = [y - f for y, f in zip(liters, fitted)]
    sse = total(e * e for e in res)
    se = math.sqrt(sse / (n - 2))
    print(f"{n} days, mean {mx:.2f} km and {my:.4f} L per day")
    print(f"Sxx = {sxx:.2f}, Sxy = {sxy:.4f}")
    print(f"liters = {a:.4f} + {b:.6f} * km")
    print(f"R^2 = {r_squared(liters, fitted):.4f},"
          f" standard error {se:.4f} L")
    print(f"slope {b:.6f} +/- {2 * se / math.sqrt(sxx):.6f} L/km (about 95%)")
    worst = max(range(n), key=lambda i: abs(res[i]))
    print(f"largest residual: day {rows[worst][0]:.0f},"
          f" {res[worst]:+.3f} L")
    print()
    design = [[1.0, x, t] for x, t in zip(km, trips)]
    coef = lstsq(design, liters)
    fit2 = predict(design, coef)
    se2 = math.sqrt(total((y - f) * (y - f) for y, f in zip(liters, fit2))
                    / (n - 3))
    print(f"liters = {coef[0]:.4f} + {coef[1]:.6f} * km"
          f" + {coef[2]:.5f} * trips")
    print(f"R^2 = {r_squared(liters, fit2):.4f}, standard error {se2:.4f} L")
    plan = coef[0] + coef[1] * 120.0 + coef[2] * 17.0
    print(f"tomorrow, 120 km and 17 trips: {plan:.2f} L")
    print()
    _, rows = read_csv("../data/battery.csv")
    months = [r[0] for r in rows]
    logs = [math.log(r[1]) for r in rows]
```

Listing 8.3 (Continued) *line_fit.py: the least squares line with R^2, standard error and slope uncertainty; two predictors by QR; the exponential battery curve.*

```
    mt, ml = total(months) / len(rows), total(logs) / len(rows)
    k = (total((t - mt) * (g - ml) for t, g in zip(months, logs))
         / total((t - mt) * (t - mt) for t in months))
    c = math.exp(ml - k * mt)
    print(f"battery health = {c:.2f} * exp({k:.5f} * month)")
    print(f"80% is reached after about {math.log(80.0 / c) / k:.1f} months")


if __name__ == "__main__":
    main()
```

Listing 8.4 `LineFit.java`: the same fits in Java.

```
import java.io.IOException;

/** Mas Dimas's fuel notebook: a straight line by the least squares
 *  formulas, then two predictors by QR, then a battery curve by logarithms. */
public class LineFit {
    public static void main(String[] args) throws IOException {
        double[][] rows = Lsq.readCsv("../data/fuel_log.csv");
        int n = rows.length;
        double[] km = new double[n], trips = new double[n], liters = new double[n];
        for (int i = 0; i < n; i++) {
            km[i] = rows[i][1];
            trips[i] = rows[i][2];
            liters[i] = rows[i][3];
        }
        double mx = Lsq.total(km) / n, my = Lsq.total(liters) / n;
        double sxx = 0, sxy = 0;
        for (int i = 0; i < n; i++) sxx += (km[i] - mx) * (km[i] - mx);
        for (int i = 0; i < n; i++) sxy += (km[i] - mx) * (liters[i] - my);
        double b = sxy / sxx;
        double a = my - b * mx;
        double[] fitted = new double[n], res = new double[n];
        double sse = 0;
        for (int i = 0; i < n; i++) {
            fitted[i] = a + b * km[i];
            res[i] = liters[i] - fitted[i];
        }
        for (double e : res) sse += e * e;
        double se = Math.sqrt(sse / (n - 2));
        System.out.println(n + " days, mean " + Lsq.f(mx, 2) + " km and "
                           + Lsq.f(my, 4) + " L per day");
        System.out.println("Sxx = " + Lsq.f(sxx, 2) + ", Sxy = " + Lsq.f(sxy, 4));
        System.out.println("liters = " + Lsq.f(a, 4) + " + " + Lsq.f(b, 6)
                           + " * km");
        System.out.println("R^2 = " + Lsq.f(Lsq.rSquared(liters, fitted), 4)
                           + ", standard error " + Lsq.f(se, 4) + " L");
        System.out.println("slope " + Lsq.f(b, 6) + " +/- "
                           + Lsq.f(2 * se / Math.sqrt(sxx), 6)
                           + " L/km (about 95%)");
        int worst = 0;
        for (int i = 1; i < n; i++) {
            if (Math.abs(res[i]) > Math.abs(res[worst])) worst = i;
        }
        System.out.println("largest residual: day " + Lsq.f(rows[worst][0], 0)
                           + ", " + Lsq.signed(res[worst], 3) + " L");
        System.out.println();
        double[][] design = new double[n][];
```

Listing 8.4 (Continued) *LineFit.java: the same fits in Java.*

```
        for (int i = 0; i < n; i++) design[i] = new double[] {1.0, km[i], trips[i]};
        double[] coef = Lsq.lstsq(design, liters);
        double[] fit2 = Lsq.predict(design, coef);
        double sse2 = 0;
        for (int i = 0; i < n; i++) {
            sse2 += (liters[i] - fit2[i]) * (liters[i] - fit2[i]);
        }
        double se2 = Math.sqrt(sse2 / (n - 3));
        System.out.println("liters = " + Lsq.f(coef[0], 4) + " + "
                           + Lsq.f(coef[1], 6) + " * km + " + Lsq.f(coef[2], 5)
                           + " * trips");
        System.out.println("R^2 = " + Lsq.f(Lsq.rSquared(liters, fit2), 4)
                           + ", standard error " + Lsq.f(se2, 4) + " L");
        double plan = coef[0] + coef[1] * 120.0 + coef[2] * 17.0;
        System.out.println("tomorrow, 120 km and 17 trips: " + Lsq.f(plan, 2)
                           + " L");
        System.out.println();
        rows = Lsq.readCsv("../data/battery.csv");
        int m = rows.length;
        double[] months = new double[m], logs = new double[m];
        for (int i = 0; i < m; i++) {
            months[i] = rows[i][0];
            logs[i] = Math.log(rows[i][1]);
        }
        double mt = Lsq.total(months) / m, ml = Lsq.total(logs) / m;
        double num = 0, den = 0;
        for (int i = 0; i < m; i++) num += (months[i] - mt) * (logs[i] - ml);
        for (int i = 0; i < m; i++) den += (months[i] - mt) * (months[i] - mt);
        double k = num / den;
        double c = Math.exp(ml - k * mt);
        System.out.println("battery health = " + Lsq.f(c, 2) + " * exp("
                           + Lsq.f(k, 5) + " * month)");
        System.out.println("80% is reached after about "
                           + Lsq.f(Math.log(80.0 / c) / k, 1) + " months");
    }
}
```

Listing 8.4 is the Java version of the line fit. **Listing 8.5** builds the powers of the centered and scaled speed with `powers`, splits the days by their number, and loops over the degrees.

Listing 8.5 `poly_fit.py`: polynomial fits of degree 1 to 8 judged on held-back days, and the best cruising speed from the parabola.

```
"""Speed and fuel economy: polynomial least squares of degree 1 to 8,
judged on days the fit has not seen, and the best cruising speed."""
import math

from lsq import lstsq, predict, r_squared, read_csv, total

CENTER, SCALE = 45.0, 25.0      # x = (speed - 45) / 25 lies in about [-1, 1]


def powers(speed, degree):
    x = (speed - CENTER) / SCALE
    row = [1.0]
    for _ in range(degree):
        row.append(row[-1] * x)
    return row


def rmse(y, fitted):
    return math.sqrt(total((u - v) * (u - v) for u, v in zip(y, fitted))
                     / len(y))


def main():
    _, rows = read_csv("../data/speed_economy.csv")
    train = [r for r in rows if int(r[0]) % 3 != 0]
    test = [r for r in rows if int(r[0]) % 3 == 0]
    print(f"{len(rows)} days: {len(train)} to fit, {len(test)} held back")
    print("degree  R^2 (fit)  RMSE fit  RMSE held back")
    for d in range(1, 9):
        a = [powers(r[1], d) for r in train]
        y = [r[2] for r in train]
        c = lstsq(a, y)
        fit = predict(a, c)
        at = [powers(r[1], d) for r in test]
        yt = [r[2] for r in test]
        print(f"{d:6d}{r_squared(y, fit):11.4f}{rmse(y, fit):10.3f}"
              f"{rmse(yt, predict(at, c)):15.3f}")
    print()
    a = [powers(r[1], 2) for r in rows]
    y = [r[2] for r in rows]
    c = lstsq(a, y)
    print(f"all days, degree 2: km/L = {c[0]:.3f} {c[1]:+.3f} x"
          f" {c[2]:+.3f} x^2")
    print("  where x = (speed - 45) / 25")
    xbest = -c[1] / (2.0 * c[2])
    best = c[0] + c[1] * xbest + c[2] * xbest * xbest
```

Listing 8.5 (Continued) *poly_fit.py: polynomial fits of degree 1 to 8 judged on held-back days, and the best cruising speed from the parabola.*

```
    print(f"best average speed {CENTER + SCALE * xbest:.1f} km/h,"
          f" about {best:.1f} km/L")
    print(f"R^2 = {r_squared(y, predict(a, c)):.4f}")


if __name__ == "__main__":
    main()
```

Listing 8.6 `PolyFit.java`: the polynomial fits in Java.

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

/** Speed and fuel economy: polynomial least squares of degree 1 to 8,
 *  judged on days the fit has not seen, and the best cruising speed. */
public class PolyFit {
    static final double CENTER = 45.0, SCALE = 25.0;  // x in about [-1, 1]

    static double[] powers(double speed, int degree) {
        double x = (speed - CENTER) / SCALE;
        double[] row = new double[degree + 1];
        row[0] = 1.0;
        for (int j = 1; j <= degree; j++) row[j] = row[j - 1] * x;
        return row;
    }

    static double rmse(double[] y, double[] fitted) {
        double s = 0;
        for (int i = 0; i < y.length; i++) {
            s += (y[i] - fitted[i]) * (y[i] - fitted[i]);
        }
        return Math.sqrt(s / y.length);
    }

    static double[][] design(List<double[]> rows, int d) {
        double[][] a = new double[rows.size()][];
        for (int i = 0; i < a.length; i++) a[i] = powers(rows.get(i)[1], d);
        return a;
    }

    static double[] target(List<double[]> rows) {
        double[] y = new double[rows.size()];
        for (int i = 0; i < y.length; i++) y[i] = rows.get(i)[2];
        return y;
    }
```

Listing 8.6 (Continued) *PolyFit.java: the polynomial fits in Java.*

```
    public static void main(String[] args) throws IOException {
        List<double[]> rows = List.of(Lsq.readCsv("../data/speed_economy.csv"));
        List<double[]> train = new ArrayList<>(), test = new ArrayList<>();
        for (double[] r : rows) ((int) r[0] % 3 != 0 ? train : test).add(r);
        System.out.println(rows.size() + " days: " + train.size() + " to fit, "
                           + test.size() + " held back");
        System.out.println("degree  R^2 (fit)  RMSE fit  RMSE held back");
        for (int d = 1; d <= 8; d++) {
            double[][] a = design(train, d);
            double[] y = target(train);
            double[] c = Lsq.lstsq(a, y);
            double[] fit = Lsq.predict(a, c);
            double[] yt = target(test);
            System.out.println(Lsq.pad("" + d, 6)
                    + Lsq.f(Lsq.rSquared(y, fit), 4, 11)
                    + Lsq.f(rmse(y, fit), 3, 10)
                    + Lsq.f(rmse(yt, Lsq.predict(design(test, d), c)), 3, 15));
        }
        System.out.println();
        double[][] a = design(rows, 2);
        double[] y = target(rows);
        double[] c = Lsq.lstsq(a, y);
        System.out.println("all days, degree 2: km/L = " + Lsq.f(c[0], 3) + " "
                           + Lsq.signed(c[1], 3) + " x " + Lsq.signed(c[2], 3)
                           + " x^2");
        System.out.println("  where x = (speed - 45) / 25");
        double xbest = -c[1] / (2.0 * c[2]);
        double best = c[0] + c[1] * xbest + c[2] * xbest * xbest;
        System.out.println("best average speed " + Lsq.f(CENTER + SCALE * xbest, 1)
                           + " km/h, about " + Lsq.f(best, 1) + " km/L");
        System.out.println("R^2 = " + Lsq.f(Lsq.rSquared(y, Lsq.predict(a, c)), 4));
    }
}
```

In **Listing 8.6** the split into fitting and held-back days is a single line that chooses between two lists; everything else follows the Python version.

8.8 Running It

Compile once in the `java` folder with `javac -d out *.java`, then run `java -cp out LineFit` and `java -cp out PolyFit`; in the `python` folder, `python3 line_fit.py` and `python3 poly_fit.py`. Java and Python print identical text.

```
40 days, mean 111.75 km and 2.6825 L per day
Sxx = 36092.60, Sxy = 748.4575
liters = 0.3652 + 0.020737 * km
R^2 = 0.9757, standard error 0.1009 L
slope 0.020737 +/- 0.001062 L/km (about 95%)
largest residual: day 17, +0.316 L

liters = 0.2654 + 0.018568 * km + 0.02285 * trips
R^2 = 0.9894, standard error 0.0675 L
tomorrow, 120 km and 17 trips: 2.88 L

battery health = 100.03 * exp(-0.00954 * month)
80% is reached after about 23.4 months
```

The first block is the straight line of **Figure 8.1**: 0.3652 L per day plus 0.020737 L per kilometer, with $R^2 = 0.9757$, a standard error of 0.1009 L and a slope known to ± 0.0011 L/km. The largest residual is day 17, which used 0.316 L more than the line predicts, three standard errors: worth a look in the notebook (a long wait with the engine running, perhaps, or a fill-up to a higher level

than usual). One unusual day among forty is normal; many would suggest that something the model does not know is going on. The second block adds the trips and explains more of the variation, as Section 8.4.2 discussed, and plans 2.88 L for tomorrow. The third block is the battery: 80% after about 23.4 months, a little under two years.

What does the slope mean for Mas Dimas's first question? Fuel per kilometer is about 0.019 to 0.021 L depending on the model, so a 10 km ride costs around 0.2 L of fuel. He can multiply that by the pump price he pays to compare with the fare. That comparison, and whether to accept the ride, is his decision; the fit only makes the cost visible.

```
36 days: 24 to fit, 12 held back
degree  R^2 (fit)  RMSE fit  RMSE held back
     1     0.3998     2.330          3.834
     2     0.9120     0.892          1.106
     3     0.9141     0.881          1.198
     4     0.9141     0.881          1.199
     5     0.9265     0.815          3.103
     6     0.9373     0.753          7.460
     7     0.9377     0.751          5.286
     8     0.9377     0.751          5.948

all days, degree 2: km/L = 54.653 +0.535 x -8.459 x^2
  where x = (speed - 45) / 25
best average speed 45.8 km/h, about 54.7 km/L
R^2 = 0.9189
```

The table is **Figure 8.2** in numbers. The straight line (degree 1) fits badly on both groups, RMSE 2.3 and 3.8 km/L, because the data are curved. Degree 2 drops the held-back error to 1.106 km/L. Degrees 3 and 4 fit the training days a hair better (0.881 against 0.892) and the held-back days a hair worse (1.198 and 1.199); from degree 5 the held-back error jumps (3.1, then 7.5 at degree 6), while the fitting error keeps improving and R^2 keeps climbing to 0.938. A report that looked only at R^2 would choose degree 8 and be the worst of the lot on new days. The degree-2 fit on all 36 days puts the best average speed at 45.8 km/h with 54.7 km/L; **Figure 8.3** shows it next to the degree-8 fit, which bends to follow individual days, most visibly at the high-speed end.

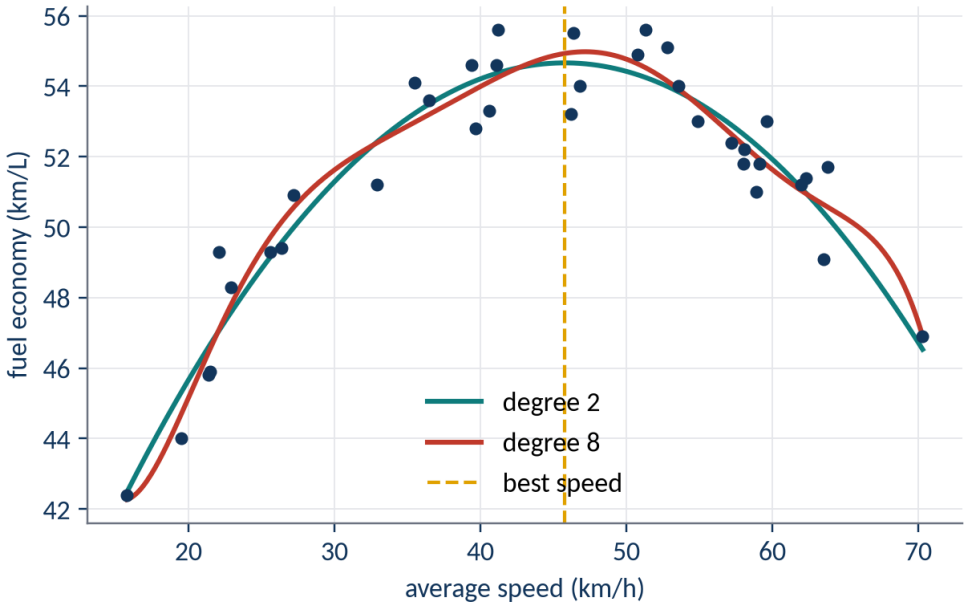


Figure 8.3 Fuel economy against average speed on 36 days, with the degree-2 fit (whose peak at 45.8 km/h is marked) and the degree-8 fit, which follows individual days instead of the trend.

The battery fit, drawn in **Figure 8.4**, is almost a straight line over two years: an exponential with a small rate looks linear over a short time. A straight line fitted directly to the health readings would say 80% after 23.1 months, the exponential 23.4. Over two years the difference is small; over five years the two models diverge, and only more readings could say which is right.

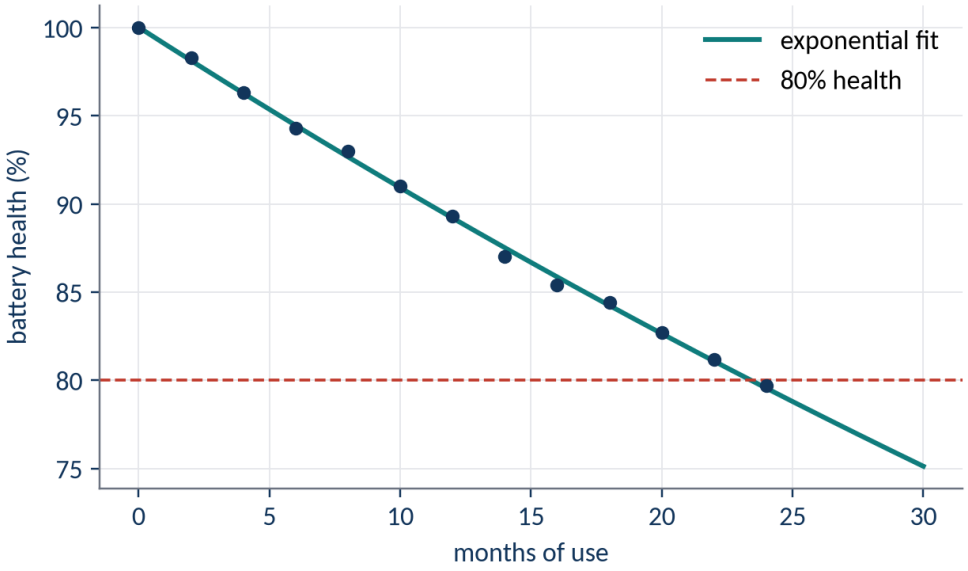


Figure 8.4 Battery health of Mas Dimas's phone every two months, the exponential fit through logarithms, and the 80% level it reaches after about 23.4 months.

8.9 Checking Against the Library

The Python ecosystem offers least squares at every level [7], [8]. `scipy.stats.linregress` fits a line and reports the slope's standard error; `numpy.linalg.lstsq` solves any linear least squares problem (with the singular value decomposition, the most robust member of the QR family); `numpy.polynomial.Polynomial.fit` fits polynomials and, like our programs, maps the data to $[-1,1]$ first; and `scipy.optimize.curve_fit` fits any model directly by nonlinear least squares [9]. **Listing 8.7** runs all four on the notebook.

Listing 8.7 `compare_numpy.py`: the fits checked against `linregress`, `lstsq`, `Polynomial.fit` and a direct nonlinear `curve_fit` of the battery curve.

```
"""Check our fits against NumPy and SciPy: linregress, lstsq, Polynomial.fit
and a direct nonlinear fit of the battery curve with curve_fit."""
import numpy as np
import scipy
from numpy.polynomial import Polynomial
from scipy.optimize import curve_fit
from scipy.stats import linregress


def main():
    print(f"NumPy {np.__version__}, SciPy {scipy.__version__}")
    fuel = np.loadtxt("../data/fuel_log.csv", delimiter=",", skiprows=1)
    km, trips, liters = fuel[:, 1], fuel[:, 2], fuel[:, 3]
    r = linregress(km, liters)
    print(f"linregress: intercept {r.intercept:.4f}, slope {r.slope:.6f},"
          f" R^2 {r.rvalue ** 2:.4f}")
    print(f"  standard error of the slope {r.stderr:.6f}")
    design = np.column_stack([np.ones_like(km), km, trips])
    coef, *_ = np.linalg.lstsq(design, liters)
    print("lstsq with trips:", np.round(coef, 6))

    speed = np.loadtxt("../data/speed_economy.csv", delimiter=",", skiprows=1)
    p = Polynomial.fit(speed[:, 1], speed[:, 2], 2)
    c = p.convert().coef                       # back to powers of the speed
    best = -c[1] / (2 * c[2])
    print(f"Polynomial.fit degree 2: best speed {best:.1f} km/h,"
          f" {p(best):.1f} km/L")

    bat = np.loadtxt("../data/battery.csv", delimiter=",", skiprows=1)
    (c0, k), _ = curve_fit(lambda t, c, k: c * np.exp(k * t),
                           bat[:, 0], bat[:, 1], p0=(100.0, -0.01))
    print(f"curve_fit: health = {c0:.2f} * exp({k:.5f} * month),"
          f" 80% after {np.log(80.0 / c0) / k:.1f} months")


if __name__ == "__main__":
    main()
```

```
NumPy 2.5.3, SciPy 1.18.1
linregress: intercept 0.3652, slope 0.020737, R^2 0.9757
  standard error of the slope 0.000531
lstsq with trips: [0.265357 0.018568 0.022854]
Polynomial.fit degree 2: best speed 45.8 km/h, 54.7 km/L
curve_fit: health = 100.04 * exp(-0.00956 * month), 80% after 23.4 months
```

The line and its R^2 agree to all printed digits, and `linregress` reports the slope's standard error as 0.000531, half of our ± 0.001062, which was two standard errors. The three coefficients with the trips agree with our QR to six decimals. `Polynomial.fit` gives the same best speed and economy. The direct nonlinear fit of the battery gives $100.04\,e^{-0.00956\,t}$ against our $100.03\,e^{-0.00954\,t}$ through logarithms, with the same 23.4 months: the difference in weighting that Section 8.5.3 described is real but small for these tidy readings.

8.10 Making It Better

- **Weighted least squares.** When some points are more reliable than others (a day with an exact fill-up versus a day with a rough estimate), give each squared residual a weight w_i, larger for reliable points. Multiplying row i of X and y_i by $\sqrt{w_i}$ turns it into an ordinary least squares problem, as Exercise 4 of Chapter 6 did for relative misses.
- **Robust fitting.** Squares punish big misses hard, so one wrong entry in the notebook (a typo of 30.6 L for 3.06 L) can drag the whole line. Robust methods reduce the weight of points with

large residuals, iteratively; the simplest check is to fit again without the worst point and see whether the answer changes much.

- **Nonlinear least squares.** Models that cannot be made linear, such as $y = C(1 - e^{-kt})$ for a battery charging curve, are fitted by iterating linear least squares on a linearization (the Gauss-Newton and Levenberg-Marquardt methods behind `curve_fit`). Chapter 13 treats this family as optimization.
- **Errors in both variables.** Least squares measures misses vertically, which is right when x is exact. When x is measured as roughly as y, the fitted slope is biased toward zero; *total least squares*, which measures misses perpendicular to the line and is computed from the singular value decomposition, treats both variables alike [5].
- **Regularization.** Instead of choosing a low degree, keep many columns but add a penalty $\lambda \| \beta \|^2$ to SSE (ridge regression). It is again a least squares problem, with $X^\top X + \lambda I$ in place of $X^\top X$, and it is one of the bridges to machine learning in Chapter 14 [6].

8.11 Using It for Real

To fit your own notebook, whether sales against daily temperature for an ice-tea stall, electricity use against the hours an air conditioner runs, or water use against the number of guests at a small inn, put the columns in a CSV file and change the file name and the columns in `line_fit.py`. Then follow a short routine.

1. **Plot first.** A scatter plot shows at once whether a line is sensible, whether there is a curve, and whether some points are strange. No statistic replaces looking. Keep a photo of each notebook page with the data file: when a point looks strange, the original entry is one glance away.
2. **Fit the simplest model that could work**, and look at the residuals. If they show a pattern (all positive in the middle and negative at the ends), the model is missing a curve.
3. **Hold data back.** Keep a quarter or a third of the days out of the fit and check the model on them before trusting it.
4. **Report the uncertainty.** A slope with its standard error, a prediction with the standard error of the estimate.
5. **Do not extrapolate far.** Mas Dimas's line was fitted for 55 to 165 km a day; for 300 km, on a highway, it says nothing reliable.

And one caution about meaning: a fit shows that two quantities move together in the data, not that one causes the other. Sales of iced tea rise with temperature, and so do sales of umbrellas in the rainy season; the umbrellas do not cause the tea. A fitted line is a summary of a notebook, not a law of nature, and it is certainly not financial advice: it can show where money goes, and the decisions stay with the person who earns it.

8.12 Numerical Methods in Practice: Profitina

Profitina's historical analytics dashboard keeps a weekly AI visibility score for each brand, re-scanned automatically on a schedule, with alerts and a CSV export. A brand owner looking at twelve weekly points that go up and down wants to know one thing: is the site getting better? The honest answer is the slope of a least squares line through the points, *points per week*, together with its standard error from Equation (8.6). A slope of $+0.8 \pm 0.3$ points per week says "improving, and not by accident"; a slope of $+0.8 \pm 1.5$ says "cannot tell yet", however encouraging the last two weeks look. The same care that Section 8.6 applied to polynomial

degrees applies to trend lines on a dashboard: a curve that follows every weekly wiggle is overfitted and predicts nothing.

Least squares also sits inside the loop that improves Profitina's own AI. Its transparent feedback pipeline turns users' feedback into a curated dataset on which the locally running Qwen2.5 model is trained and measured before a release, and the Content Engine's quality gate serves an AI draft only when it scores better than a built-in template (after the September 2026 revision, AI drafts win in 5 of 6 benchmark cases). Every such measurement is a small statistics problem of the kind this chapter taught: an average, a spread, a comparison that must be large relative to its noise before anyone believes it. As always, the numbers advise and the people decide.

The arithmetic of such a trend is exactly Equation (8.3) with x the week number. For twelve weeks numbered 1 to 12, $\bar{x} = 6.5$ and $S_{xx} = 143$, so by Equation (8.6) the slope's standard error is $s_e/\sqrt{143} \approx s_e/12$. If the weekly scores scatter around their trend by about 3 points, the slope is known to about ± 0.25 points per week, and a trend has to be steeper than half a point per week before twelve weeks can tell it from noise. Twenty-six weeks would give $S_{xx} = 1462.5$ and a standard error more than three times smaller: patience buys certainty, which is worth saying plainly to a brand owner who wants a verdict after three weeks.

8.13 Common Mistakes, and How to Avoid Them

- **Judging a model by R^2 alone.** R^2 never decreases when a column is added. Compare models on held-back data (Equation (8.13), **Figure 8.2**).
- **Fitting high-degree polynomials in raw units.** Powers of 70 up to the eighth are 15 digits long and nearly parallel. Center and scale the variable, and solve by QR, not by the normal equations.
- **Minimizing vertical misses when the error is in x.** Least squares as used here assumes the error is in y; the distance from the app is accurate, the fuel is not. If both are noisy, the fitted slope is biased toward zero, and methods such as total least squares are needed.
- **Forgetting what the log transform does.** Fitting $\ln y$ minimizes relative errors, not absolute ones. For precise answers in the original units, check with a direct nonlinear fit.
- **Extrapolating a fit.** Every model is trustworthy only over the range of the data that made it.
- **Throwing away the "outlier" without asking why.** Day 17 may be a mistake in the notebook, or it may be the only day with a long wait in the rain. Investigate before deleting, and never delete points just to improve R^2.
- **Comparing R^2 across different targets.** An R^2 computed on $\ln H$ and one computed on H measure different things and cannot be compared. Compare models on the same quantity, in the same units, preferably on held-back data.
- **Reading correlation as cause.** Two columns that move together may both follow a third.

8.14 Chapter Summary

- Least squares chooses the curve that minimizes the sum of squared residuals (Equation (8.1)). For a straight line the normal equations (Equation (8.2)) give the slope S_{xy}/S_{xx} and an intercept that makes the line pass through the means (Equation (8.3)); **Table 8.1** does it by hand for five days.
- The residuals of a fit with an intercept add up to zero and are uncorrelated with the predictors. R^2 (Equation (8.4)) is the fraction of variation explained, the standard error (Equation (8.5)) is

the typical miss, and the slope's standard error (Equation (8.6)) gives an honest uncertainty: 0.0207 ± 0.0011 L/km for Mas Dimas (**Figure 8.1**).

- A single new day is predicted with the wider interval Equation (8.7), which adds the day's own scatter to the line's uncertainty: 3.48 ± 0.21 L for 150 km.
- Several predictors enter through the design matrix (Equation (8.8)) and the normal equations $X^{\mathsf{T}}X\beta = X^{\mathsf{T}}y$ (Equation (8.9)), best solved by the Householder QR of Chapter 6. The coefficient of the trips, 0.023 L per trip, is the effect at a fixed distance.
- Polynomials (Equation (8.10)) and exponentials through logarithms (Equation (8.12)) are linear least squares problems. Center and scale before fitting powers. The parabola's vertex (Equation (8.11)) puts the best speed at 45.8 km/h (**Figure 8.3**); the battery reaches 80% after 23.4 months (**Figure 8.4**).
- More flexible models always fit the training data better; only the error on held-back data (Equation (8.13)) reveals overfitting (**Figure 8.2**). Here degree 2 wins.
- The Java and Python programs agree byte for byte, and `linregress`, `lstsq`, `Polynomial.fit` and `curve_fit` confirm every result.

8.15 Exercises

1. Using **Table 8.1**, check that the residuals of the five-day line add up to zero, and compute the five-day R^2. (The five $y_i - \bar{y}$ values give SST.)
2. Show from the first equation of Equation (8.2) that a least squares line with an intercept passes through $(\bar{x}, \bar{y})$. Then explain why the formula $S_{xx} = \sum x_i^2 - n\bar{x}^2$ is risky in floating point by trying it, in your head, on the distances 100000.1, 100000.2 and 100000.3 km stored with about 16 significant digits.
3. With the forty-day line, predict the fuel for a day of 150 km, and give a rough 95% range for that single day using $\pm 2s_e$. Then predict with the two-predictor model for 150 km and 20 trips.
4. The multiple fit says that one extra trip at the same distance costs 0.0229 L. If a delivery platform offers a bonus for completing 5 extra short trips in the same distance, how much extra fuel does the model expect? Why should Mas Dimas trust this number less than the cost per kilometer?
5. Fit a straight line (not an exponential) to the battery readings and find when it reaches 80%. Compare with 23.4 months and explain why the two models agree now but will disagree in five years.
6. Repeat the degree comparison of `poly_fit.py` with the held-back group chosen as the days whose number leaves remainder 1 when divided by 3, and then remainder 2. Which degree has the smallest held-back RMSE in each case? What does cross-validation, averaging the three splits, recommend?
7. Explain why, in Equation (8.6), a notebook of forty days with distances spread between 55 and 165 km gives a more precise cost per kilometer than eighty days all between 100 and 120 km. Estimate the ratio of the two S_{xx} values assuming evenly spread distances.
8. Use Equation (8.7) to give 95% prediction intervals for a day of 60 km and for a day of 250 km. Why is the second interval only moderately wider, and why should Mas Dimas still distrust it much more than the first?

8.16 Solutions to the Exercises

1. The residuals are $-0.096 + 0.034 - 0.017 + 0.089 - 0.010 = 0.000$ (exactly zero up to the rounding of the printed values). SST $= 0.812^2 + 0.222^2 + 1.008^2 + 0.342^2 + 0.368^2 = 0.6593 + 0.0493 + 1.0161 + 0.1170 + 0.1354 = 1.9771$. SSE $= 0.096^2 + 0.034^2 + 0.017^2 + 0.089^2 + 0.010^2 = 0.00922 + 0.00116 + 0.00029 + 0.00792 + 0.00010 = 0.01869$ (0.01862 with the unrounded residuals). So $R^2 = 1 - 0.01862/1.9771 = 0.9906$. Five points fit a line very well; with forty the fit is still excellent, $R^2 = 0.9757$, but more honest.

2. The first equation of Equation (8.2) is $na + b\sum x_i = \sum y_i$; dividing by n gives $a + b\bar{x} = \bar{y}$, which says that the point $(\bar{x}, \bar{y})$ lies on the line. For the distances: $\bar{x} = 100000.2$, the squares are about 10^{10}, and $\sum x_i^2 - n\bar{x}^2$ subtracts two numbers of about 3×10^{10} that agree in their first 12 digits, to get the true $S_{xx} = 0.02$. With 16 significant digits, only about 4 are left for the result, and with larger or more numbers none may be left at all. Centering first computes $(-0.1)^2 + 0^2 + 0.1^2 = 0.02$ from small numbers without cancellation.

3. $0.3652 + 0.020737 \times 150 = 3.476$ L. A single day scatters around the line with a typical size $s_e = 0.101$ L, so a rough 95% range is 3.48 ± 0.20 L, from about 3.28 to 3.68 L; the full prediction interval Equation (8.7), which also includes the small uncertainty of the line itself, gives ± 0.21 L. With trips: $0.2654 + 0.018568 \times 150 + 0.02285 \times 20 = 0.2654 + 2.7852 + 0.4570 = 3.508$ L.

4. $5 \times 0.0229 = 0.114$ L, about a tenth of a liter. It is less reliable than the cost per kilometer for two reasons: trips are strongly correlated with distance in the notebook, so their separate effect is estimated with more uncertainty; and short extra trips may differ from the average trip in the notebook (more waiting, or less). The coefficient describes the notebook's days, not a new way of working.

5. The least squares line through the thirteen readings is $H = 99.67 - 0.8533\,t$, which reaches 80% at $t = (99.67 - 80)/0.8533 = 23.1$ months, against 23.4 for the exponential. Over two years $e^{-0.00954t} \approx 1 - 0.00954t$ is almost linear, so the two models nearly coincide on the data. Extrapolated to 60 months, the line says $99.67 - 51.2 = 48.5\%$ and the exponential $100.03\,e^{-0.572} = 56.4\%$. The data cannot tell them apart; only readings in years three to five can, which is the rule against extrapolation in action.

6. Change the two filters to `int(r[0]) % 3 != 1` and `== 1` (then 2). Held-back RMSE for degrees 1 to 8: with remainder 1, 2.948, 0.795, 0.773, 0.849, 1.324, 2.576, 3.847, 5.340, smallest at degree 3 with degree 2 very close; with remainder 2, 2.106, 1.146, 1.177, 1.221, 1.216, 1.375, 1.335, 1.390, smallest at degree 2. Averaging the three splits (including the chapter's split, 3.834, 1.106, 1.198, …), degree 2 has the smallest mean held-back error (about 1.02 against 1.05 for degree 3). Cross-validation recommends degree 2, and the simpler model also has the physical story on its side (one peak).

7. S_{xx} grows with the number of days and with the square of their spread. For n values spread evenly over a range of width w, $S_{xx} \approx nw^2/12$. Forty days over 110 km: $40 \times 110^2/12 \approx 40{,}300$; eighty days over 20 km: $80 \times 20^2/12 \approx 2{,}670$. The ratio is about 15, so by Equation (8.6) the standard error of the slope is $\sqrt{15} \approx 3.9$ times smaller with the spread-out notebook, despite having half the days. To learn how something changes with x, vary x.

8. For 60 km: $\hat{y}_0 = 0.3652 + 0.020737 \times 60 = 1.609$ L, and the root is $\sqrt{1 + 0.025 + (60 - 111.75)^2/36092.6} = \sqrt{1.0992} = 1.048$, so the interval is $1.609 \pm 2.02 \times 0.1009 \times 1.048 = 1.609 \pm 0.214$ L. For 250 km: $\hat{y}_0 = 5.549$ L and the root is $\sqrt{1.5546} =$

1.247, so 5.549 ± 0.255 L. The second interval is only about 20% wider, because the formula measures only the random scatter *under the assumption that the straight line is still the right model* at 250 km. Nothing in the notebook supports that assumption: a 250 km day would include long highway stretches at speeds the notebook never saw, and Section 8.5 showed that fuel economy changes with speed. The interval is honest about noise and silent about the model, which is why extrapolation needs judgment, not only a formula.

References

[1] A.-M. Legendre, *Nouvelles méthodes pour la détermination des orbites des comètes*. Paris, France: Firmin Didot, 1805.

[2] C. F. Gauss, *Theoria motus corporum coelestium in sectionibus conicis solem ambientium*. Hamburg, Germany: Perthes und Besser, 1809.

[3] D. C. Montgomery, E. A. Peck, and G. G. Vining, *Introduction to Linear Regression Analysis*, 6th ed. Hoboken, NJ, USA: Wiley, 2021.

[4] S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers*, 8th ed. New York, NY, USA: McGraw-Hill Education, 2021.

[5] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical Analysis*, 10th ed. Boston, MA, USA: Cengage Learning, 2016.

[6] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, 2nd ed. New York, NY, USA: Springer, 2009.

[7] NumPy Developers, "NumPy 2.5.0 Release Notes," Jun. 21, 2026, and "numpy 2.5.3" on the Python Package Index, Sep. 6, 2026. [Online]. Available: https://numpy.org/doc/2.5/release/2.5.0-notes.html and https://pypi.org/project/numpy/

[8] SciPy Developers, "SciPy 1.18.0 Release Notes," Jun. 19, 2026, and "scipy 1.18.1" on the Python Package Index, Aug. 21, 2026. [Online]. Available: https://docs.scipy.org/doc/scipy/release/1.18.0-notes.html and https://pypi.org/project/scipy/

[9] SciPy Developers, "scipy.optimize.curve_fit" and "scipy.stats.linregress," SciPy v1.18 Manual, accessed Sep. 2026. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html

Chapter 9
Numerical Differentiation

A phone can tell you where you are, but not directly how fast you are going or how hard you just braked. Those are *rates of change*, derivatives, and a computer that only has readings taken once a second must estimate them from differences between the readings. This chapter shows how to do that well. By the end of the chapter you will be able to derive the forward, backward and central difference formulas and their second-derivative cousins from Taylor's theorem; state and check the order of their error; explain why making the step smaller first helps and then hurts, and choose a step that balances truncation and round-off; improve an estimate with Richardson extrapolation; build a trip analyzer in Java and in Python that turns an ojek driver's distance log into speed and acceleration and flags harsh braking; see why measurement noise is the real enemy of numerical derivatives and remove a false alarm with a least-squares smoothing formula; and check every answer against NumPy and SciPy.

9.1 Mas Dimas's Braking Report

Mas Dimas drives an online ojek in Yogyakarta. Most of his day is short trips between the campuses, the boarding houses and the markets around Jalan Kaliurang, with a student or a parcel on the back seat. One evening his app shows a small orange message on the trip summary: *harsh braking detected*. It is not a fine, but the message worries him. A few such warnings, a friend told him, and a driver's safety score goes down, and with it the chance of getting the better orders.

He remembers the moment well. He was riding at about forty kilometers an hour along a quiet lane when a cat ran out from under a parked car, and he braked hard. That was right, and he would do it again. But he also wonders about another trip a week earlier, when the app complained although he is sure he only rolled gently to a stop in front of a customer's gate. Was the app wrong? How does it know how hard he braked at all?

The phone knows only one kind of thing well: *where* it is, and therefore how far it has traveled since the trip began. Suppose it writes down the distance once every second, as the app on his phone does. A second later the number is bigger; the difference is the distance covered in that second, and distance per second is speed. The change in speed from one second to the next is acceleration, and a large negative acceleration is braking. So speed and acceleration are hidden in the list of distances, but they are not written in it. They must be *computed*, by subtracting neighboring readings and dividing by the time between them.

That sounds almost too simple to need a chapter. It is not. The same subtraction that reveals the speed also magnifies every small error in the readings, and the second subtraction, for acceleration, magnifies it again. An error of a few tens of centimeters in the phone's distance, which nobody would notice on a map, becomes an error of more than one meter per second squared in the acceleration, which is the difference between "normal stop" and "harsh braking". Understanding exactly how large these errors are, where they come from, and what can be done about them is the subject of numerical differentiation, and it matters far beyond ojek apps: the flow into a water tank from its level, the drain rate of a phone battery from its percentage, the

growth rate of a business from its monthly sales, and the trend of a website's score from its weekly scans are all derivatives estimated from samples.

In this chapter we take Mas Dimas's trip log, forty-one distance readings, one per second, and build the analyzer the app should have: it prints speed and acceleration for every second, marks harsh braking, and, in its improved form, tells the real hard brake from the false alarm.

9.2 The Idea in Plain Words

9.2.1 Speed from two readings

Table 9.1 shows five of the readings from the trip, around the moment of the cat.

Table 9.1 Five distance readings from Mas Dimas's trip, one per second, around the hard brake.

Time t (s)	16	17	18	19	20
distance s (m)	148.2	159.5	168.7	174.4	176.5

Between $t = 17$ and $t = 18$ the distance grew by $168.7 - 159.5 = 9.2$ m, so over that second the motorbike moved at an average of 9.2 m/s. Between $t = 18$ and $t = 19$ it moved $174.4 - 168.7 = 5.7$ m, an average of 5.7 m/s. The speed *at* $t = 18$ must be somewhere near both, and the natural guess is to look one second back and one second forward and take the average speed over those two seconds:

$$v(18) \approx \frac{174.4 - 159.5}{19 - 17} = \frac{14.9}{2} = 7.45 \text{ m/s},$$

about 27 km/h. On a graph of distance against time, this is the slope of the straight line through the points at $t = 17$ and $t = 19$. The true speed at $t = 18$ is the slope of the *tangent* to the curve at that point; the line through the two neighbors, a *secant*, has almost the same slope when the neighbors are close. That is the whole idea of numerical differentiation: replace the tangent, which needs the exact curve, by a secant, which needs only readings.

There are three obvious secants to choose from. Looking only forward, from $t = 18$ to $t = 19$, gives 5.7 m/s; looking only backward, from $t = 17$ to $t = 18$, gives 9.2 m/s; looking both ways gives 7.45 m/s. The three answers differ a lot, because the speed was changing quickly: the bike was braking. The forward look describes the speed half a second *after* $t = 18$, the backward look half a second *before*, and the two-sided look is centered on $t = 18$ itself. Section 9.3 proves that the centered choice is not only fairer but much more accurate.

9.2.2 Acceleration from three readings

Acceleration is the rate of change of speed. The speed over the second before $t = 18$ was 9.2 m/s, and over the second after it was 5.7 m/s; these two averages belong to the times 17.5 and 18.5, one second apart. The change is $5.7 - 9.2 = -3.5$ m/s in one second, so

$$a(18) \approx \frac{5.7 - 9.2}{1} = -3.5 \text{ m/s}^2.$$

Written with the readings themselves, this is $(174.4 - 2 \times 168.7 + 159.5)/1^2 = -3.5$: the *second difference* of three neighboring readings, divided by the square of the time step. A deceleration of 3.5 m/s^2 means losing 12.6 km/h of speed every second. Many telematics apps treat braking harder than about 3 m/s^2 as harsh, and our analyzer will use that threshold, -3.0 m/s^2. At $t = 18$ Mas Dimas was clearly above it, as he remembers.

Now look at the numbers again with a skeptical eye. The distances are given to a tenth of a meter, and a phone's position, even after the app smooths the GPS, is uncertain by a few tens of centimeters. Suppose the reading at $t = 18$ were 0.3 m too high. The speed at $t = 18$ would not change at all (the centered formula does not use the middle reading), but the acceleration would change by $-2 \times 0.3 = -0.6$ m/s^2. If the neighbors were each 0.3 m too low at the same time, the error would reach -1.2 m/s^2. An error that small in distance is invisible; the same error in acceleration can turn a gentle stop of -1.5 m/s^2 into a "harsh" -2.7 or worse. This is the mystery of the gate, and Section 9.6 will find it in the data.

> **KEY INSIGHT:** Differentiation from data divides small differences of large numbers by a small step. Every error in the readings is divided by the step, once for the first derivative and twice for the second. Integration, the subject of Chapter 10, adds readings and averages errors away; differentiation subtracts readings and amplifies them. That is why it is the more delicate of the two.

9.3 Deriving the Formulas from Taylor's Theorem

The secant idea gives formulas, but not their accuracy. To know how wrong a formula is, and how the error changes when the step changes, we use Taylor's theorem, the tool behind nearly every error estimate in this book [1], [2].

9.3.1 Taylor's theorem

Let f be a function with enough continuous derivatives near a point x, and let h be a small step, positive or negative. Taylor's theorem says that the value of f at $x + h$ can be built from the value and the derivatives of f at x:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \cdots + \frac{h^n}{n!}f^{(n)}(x) + R_n \tag{9.1}$$

where f', f'' and f''' are the first, second and third derivatives, $f^{(n)}$ is the n-th derivative, $n! = 1 \times 2 \times \cdots \times n$, and the remainder is $R_n = \frac{h^{n+1}}{(n+1)!}f^{(n+1)}(\xi)$ for some point ξ between x and $x + h$. In words: near x, the function looks like its value, plus the slope times the step, plus half the curvature times the step squared, and so on, each term smaller than the one before when h is small. We do not know ξ, but we usually know, or can bound, how large $f^{(n+1)}$ can be, and that bounds the error.

For the ojek, f is the distance $s(t)$, f' the speed and f'' the acceleration, and Equation (9.1) says something every driver knows: the distance after a short time h is the current distance, plus speed times time, plus half the acceleration times time squared, plus small corrections.

9.3.2 Forward and backward differences

Take Equation (9.1) with $n = 1$ and solve for $f'(x)$:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(\xi) \quad \Rightarrow \quad f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{h}{2}f''(\xi).$$

The first part is the **forward difference**, the slope of the secant to the point ahead:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}, \qquad \text{error} = -\frac{h}{2}f''(\xi). \tag{9.2}$$

The part we threw away, $-\frac{h}{2}f''(\xi)$, is the **truncation error**: the error that comes from cutting the Taylor series short, and that would be there even on a computer with infinitely many digits. It is proportional to h. Halve the step and the error halves; divide the step by ten and the error falls by ten. We say the forward difference is *first-order accurate* and write the error as $O(h)$, read "order h", meaning "at most a constant times h for small h".

The same argument with $-h$ in place of h gives the **backward difference**, the slope of the secant to the point behind:

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}, \qquad \text{error} = +\frac{h}{2}f''(\xi).$$

It is also first order. For Mas Dimas at $t = 18$, the forward difference gave 5.7 m/s and the backward 9.2 m/s. The error terms explain the gap: f'' is the acceleration, about -3.5 m/s^2, and $h = 1$ s, so each estimate is off by about $\frac{1}{2} \times 3.5 = 1.75$ m/s, in opposite directions. The truth should lie near the middle, 7.45.

9.3.3 The central difference

The observation that the forward and backward errors have opposite signs suggests adding them. Write Equation (9.1) for $+h$ and for $-h$, one term further:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(\xi_1),$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(\xi_2).$$

Subtract the second line from the first. The $f(x)$ terms cancel, and so do the $f''(x)$ terms, because h^2 has the same sign for $+h$ and $-h$:

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{h^3}{6}(f'''(\xi_1) + f'''(\xi_2)).$$

Divide by $2h$, and use the intermediate value theorem to replace the average of $f'''(\xi_1)$ and $f'''(\xi_2)$ by a single $f'''(\xi)$:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}, \qquad \text{error} = -\frac{h^2}{6} f'''(\xi). \tag{9.3}$$

This is the **central difference**. Its error is proportional to h^2: halve the step and the error falls to a quarter; divide it by ten and the error falls a hundredfold. It is *second-order accurate*, $O(h^2)$, and it costs exactly the same two function values as the forward difference. There is rarely a reason to use the forward difference when the point ahead and the point behind are both available.

A useful special case: when f is a parabola, $f''' = 0$ and the central difference is *exact*, whatever h. A motorbike braking with constant deceleration has a distance that is a parabola in time, so on any stretch of constant braking, the central difference gives the exact speed, apart from measurement error. On Mas Dimas's trip the acceleration changes abruptly (he starts and stops braking), so f''' is large at a few moments and zero in between.

9.3.4 The second derivative

For the acceleration we need f''. This time *add* the two expansions, taken to the fourth power of h:

$$f(x+h) + f(x-h) = 2f(x) + h^2 f''(x) + \frac{h^4}{12} f^{(4)}(\xi).$$

The odd powers cancel, because h and h^3 change sign with h. Solving for $f''(x)$:

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}, \qquad \text{error} = -\frac{h^2}{12} f^{(4)}(\xi). \tag{9.4}$$

This is the **central second difference**, the formula of Section 9.2.2. It is also second-order accurate, and exact for cubics. Notice the h^2 in the denominator: a reading error δ in each of the three values produces an error of up to $4\delta/h^2$ in the result. With $h = 1$ s and $\delta = 0.3$ m that is 1.2 m/s^2, exactly the danger noticed in Section 9.2.2.

9.3.5 One-sided formulas for the ends

At the first reading of a trip there is no point behind, and at the last there is no point ahead. The two-point forward and backward differences work there but are only first order. A better choice uses *three* points on one side. We look for weights c_0, c_1, c_2 such that

$$f'(x) \approx \frac{c_0 f(x) + c_1 f(x+h) + c_2 f(x+2h)}{h}$$

is as accurate as possible. Expand $f(x+h)$ and $f(x+2h)$ by Equation (9.1) and collect the terms in f, f' and f''. For the formula to reproduce $f'(x)$, the coefficient of $f(x)$ must vanish, the coefficient of $hf'(x)$ must be 1, and the coefficient of $h^2 f''(x)$ must vanish:

$$c_0 + c_1 + c_2 = 0, \qquad c_1 + 2c_2 = 1, \qquad \frac{1}{2} c_1 + 2c_2 = 0.$$

From the third equation $c_1 = -4c_2$; the second gives $-4c_2 + 2c_2 = 1$, so $c_2 = -\frac{1}{2}$ and $c_1 = 2$; the first gives $c_0 = -\frac{3}{2}$. Hence

$$f'(x) \approx \frac{-3f(x) + 4f(x+h) - f(x+2h)}{2h}, \qquad \text{error} = \frac{h^2}{3} f'''(\xi), \tag{9.5}$$

a second-order formula that looks only forward. Its mirror image, $(3f(x) - 4f(x-h) + f(x-2h))/(2h)$, looks only backward. This way of finding weights, by demanding that the formula be exact for $1, h, h^2, \ldots$, is the **method of undetermined coefficients**. It produces every formula in this chapter, and Fornberg's algorithm does it automatically for any set of points, even unevenly spaced ones [3]. The same method with four points gives a second-order one-sided second derivative, $(2f(x) - 5f(x+h) + 4f(x+2h) - f(x+3h))/h^2$, which the analyzer uses at the two ends of the trip.

The error constant of Equation (9.5), $\frac{1}{3}$, is twice that of the central difference, $\frac{1}{6}$. One-sided formulas are always less accurate than centered ones of the same order, and they are also more sensitive to noise, because their weights are larger ($-3, 4, -1$ against $-1, 1$). Section 9.6 shows both effects at the start and at the end of the trip.

9.3.6 The formulas side by side

Table 9.2 collects the formulas. The pattern is worth remembering: to gain one order of accuracy you need one more point, and centered formulas get a free order because odd error terms cancel by symmetry.

Table 9.2 Finite-difference formulas for the first and second derivative, with the number of function values they need and their truncation error (ξ omitted).

Derivative	Formula	Points	Truncation error
$f'(x)$ forward	$[f(x+h) - f(x)]/h$	2	$-\frac{h}{2} f''$
$f'(x)$ backward	$[f(x) - f(x-h)]/h$	2	$+\frac{h}{2} f''$
$f'(x)$ central	$[f(x+h) - f(x-h)]/(2h)$	2	$-\frac{h^2}{6} f'''$
$f'(x)$ three-point forward	$[-3f(x) + 4f(x+h) - f(x+2h)]/(2h)$	3	$+\frac{h^2}{3} f'''$
$f'(x)$ five-point central	$[f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)]/(12h)$	4	$+\frac{h^4}{30} f^{(5)}$
$f''(x)$ central	$[f(x+h) - 2f(x) + f(x-h)]/h^2$	3	$-\frac{h^2}{12} f^{(4)}$
$f''(x)$ four-point forward	$[2f(x) - 5f(x+h) + 4f(x+2h) - f(x+3h)]/h^2$	4	$+\frac{11h^2}{12} f^{(4)}$

9.4 Two Kinds of Error: Truncation and Round-off

Taylor's theorem says the error shrinks as h shrinks. So why not take $h = 10^{-12}$ and get a nearly perfect derivative? For readings logged once a second the answer is obvious: we cannot choose h, it is one second. But even when we can evaluate a function anywhere we like, making h tiny is a mistake, and seeing why teaches the most important lesson of the chapter.

To study the error we need a function whose derivative we know exactly. Take $f(x) = \sqrt{x}$ at $x = 4$. Its derivative is $f'(x) = \frac{1}{2\sqrt{x}}$, so $f'(4) = 0.25$ exactly. **Listing 9.1** computes the forward and

central differences for twelve steps from 10^{-1} down to 10^{-12} and prints each estimate with its error.

Listing 9.1 `diff_steps.py`: forward and central differences of $\sqrt{x}$ at $x = 4$ for ever smaller steps.

```
"""Forward and central differences for f(x) = sqrt(x) at x = 4 with ever
smaller steps h. The exact derivative is 1/(2*sqrt(4)) = 0.25."""
import math


def forward(f, x, h):
    return (f(x + h) - f(x)) / h           # error O(h)


def central(f, x, h):
    return (f(x + h) - f(x - h)) / (2 * h)  # error O(h^2)


def main():
    x, exact = 4.0, 0.25
    steps = [1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6,
             1e-7, 1e-8, 1e-9, 1e-10, 1e-11, 1e-12]
    print("     h     forward        error    central        error")
    for h in steps:
        d1 = forward(math.sqrt, x, h)
        d2 = central(math.sqrt, x, h)
        print(f"{h:6.0e}  {d1:.10f}  {abs(d1 - exact):.2e}"
              f"  {d2:.10f}  {abs(d2 - exact):.2e}")


if __name__ == "__main__":
    main()
```

Listing 9.2 is the same program in Java. The function is passed as a `DoubleUnaryOperator`, so `Math::sqrt` can be handed to the formulas just as `math.sqrt` is in Python. Every output line is formatted with `Locale.ROOT`, so that a computer set to Indonesian does not print a decimal comma; with that, the two programs print exactly the same characters.

Listing 9.2 `DiffSteps.java`: the same experiment in Java.

```java
import java.util.Locale;
import java.util.function.DoubleUnaryOperator;

/** Forward and central differences for f(x) = sqrt(x) at x = 4 with ever
 *  smaller steps h. The exact derivative is 1/(2*sqrt(4)) = 0.25. */
public class DiffSteps {
    static double forward(DoubleUnaryOperator f, double x, double h) {
        return (f.applyAsDouble(x + h) - f.applyAsDouble(x)) / h;  // O(h)
    }

    static double central(DoubleUnaryOperator f, double x, double h) {
        // error O(h^2)
        return (f.applyAsDouble(x + h) - f.applyAsDouble(x - h)) / (2 * h);
    }

    public static void main(String[] args) {
        double x = 4.0, exact = 0.25;
        double[] steps = {1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6,
                          1e-7, 1e-8, 1e-9, 1e-10, 1e-11, 1e-12};
        System.out.println(
            "     h     forward        error    central        error");
        for (double h : steps) {
            double d1 = forward(Math::sqrt, x, h);
            double d2 = central(Math::sqrt, x, h);
            System.out.println(String.format(Locale.ROOT,
                "%6.0e  %.10f  %.2e  %.10f  %.2e",
                h, d1, Math.abs(d1 - exact), d2, Math.abs(d2 - exact)));
        }
    }
}
```

Both print:

```
     h     forward        error    central        error
 1e-01  0.2484567313  1.54e-03  0.2500195366  1.95e-05
 1e-02  0.2498439450  1.56e-04  0.2500001953  1.95e-07
 1e-03  0.2499843770  1.56e-05  0.2500000020  1.95e-09
 1e-04  0.2499984375  1.56e-06  0.2500000000  2.05e-11
 1e-05  0.2499998438  1.56e-07  0.2500000000  1.64e-12
 1e-06  0.2499999843  1.57e-08  0.2499999999  7.61e-11
 1e-07  0.2499999985  1.52e-09  0.2499999996  4.09e-10
 1e-08  0.2499999763  2.37e-08  0.2499999874  1.26e-08
 1e-09  0.2500000207  2.07e-08  0.2500000207  2.07e-08
 1e-10  0.2500000207  2.07e-08  0.2500000207  2.07e-08
 1e-11  0.2499778162  2.22e-05  0.2499889185  1.11e-05
 1e-12  0.2500222251  2.22e-05  0.2500222251  2.22e-05
```

Read the forward column from the top. From $h = 10^{-1}$ to $h = 10^{-7}$ the error falls by a factor of ten for each factor of ten in h, exactly as $O(h)$ promises; the numbers even match the formula: $\frac{h}{2}|f''(4)| = \frac{h}{2} \times \frac{1}{32} = 0.0156\,h$, which gives 1.56×10^{-3} for $h = 0.1$. The central column falls a hundredfold per step, as $O(h^2)$ promises, with $\frac{h^2}{6}|f'''(4)| = \frac{h^2}{6} \times \frac{3}{256} = 0.00195\,h^2$, which gives 1.95×10^{-5} for $h = 0.1$. Then, for both formulas, something breaks. The forward error is smallest at $h = 10^{-7}$ and the central error at $h = 10^{-5}$; below those steps the errors *grow*, and at $h = 10^{-12}$ both estimates are wrong in the fifth digit, worse than with $h = 0.01$.

The culprit is **round-off error**, the subject of Chapter 1. A double-precision number carries about 16 significant digits; the relative spacing of doubles is the machine epsilon $\varepsilon = 2^{-52} \approx 2.2 \times 10^{-16}$ [4]. The value $\sqrt{4 + h}$ is therefore stored with an error of up to about $\varepsilon \times 2$, and so is $\sqrt{4}$. When h is tiny, the two values agree in most of their digits, and subtracting them cancels those digits, leaving only the uncertain ones: this is the catastrophic cancellation of Chapter 1. The

difference, with its absolute error of a few times 10^{-16}, is then divided by h. The round-off part of the error is roughly

$$E_{\text{round}} \approx \frac{2\varepsilon|f(x)|}{h}$$

for the forward difference: it grows as h shrinks. The total error is the sum of a truncation part that shrinks with h and a round-off part that grows as h shrinks. **Figure 9.1** draws both columns of the output on logarithmic axes. Each curve is a "V": a straight descent with slope 1 (forward) or 2 (central) while truncation dominates, then a noisy climb with slope -1 once round-off takes over.

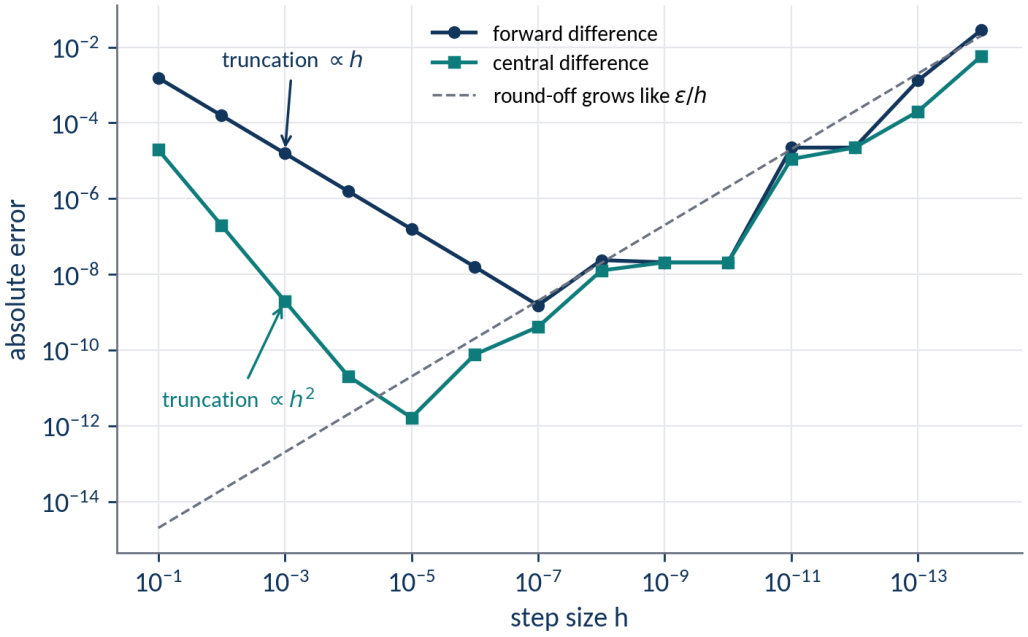


Figure 9.1 Error of the forward and central differences of $\sqrt{x}$ at $x = 4$ against the step h (both axes logarithmic, h shrinking to the right). Truncation error falls like h or h^2; round-off error rises like ε/h.

Where is the bottom of the V? For the forward difference the total error is about

$$E(h) = \frac{h}{2}M_2 + \frac{2\varepsilon|f|}{h},$$

where M_2 is a bound on \$f''\$. Its derivative with respect to h is $\frac{M_2}{2} - \frac{2\varepsilon|f|}{h^2}$, which is zero at

$$h^* = 2\sqrt{\frac{\varepsilon|f|}{M_2}} \quad \text{(forward)}, \qquad h^* = \left(\frac{3\varepsilon|f|}{M_3}\right)^{1/3} \quad \text{(central)}, \tag{9.6}$$

where the central version is found the same way from $E(h) = \frac{h^2}{6}M_3 + \frac{\varepsilon|f|}{h}$ with M_3 a bound on \$f'''\$. For a function whose value and derivatives are all of size about 1, this gives the rules of thumb $h^* \approx \sqrt{\varepsilon} \approx 10^{-8}$ for the forward difference and $h^* \approx \varepsilon^{1/3} \approx 10^{-5}$ for the central difference, with best errors of about 10^{-8} and 10^{-11}. For $\sqrt{x}$ at 4, with $|f| = 2$, $M_2 = 1/32$ and $M_3 = 3/256$, Equation (9.6) gives $h^* \approx 2.4 \times 10^{-7}$ and $h^* \approx 4.8 \times 10^{-5}$, in good agreement with the bottoms of

the curves in **Figure 9.1**. In practice one also scales h with $x, taking $h = \sqrt{\varepsilon}\max(1, |x|)$, because the spacing of doubles grows with the size of x [2], [4].

The flat stretch of both curves between $h = 10^{-9}$ and 10^{-10}, where the forward and central estimates are identical, is round-off in its purest form: $4 + h$ is rounded to the nearest double, so the step actually used is not the h we asked for, and both formulas divide the same few surviving bits by the same wrong step. A careful program computes the step it really used as `(x + h) - x` and divides by that; Exercise 9.3 explores the rules of thumb further.

> **WARNING:** There is no step that makes a finite-difference derivative as accurate as the function values. With double precision the forward difference loses about half the digits and the central difference about a third. If you need a derivative to full precision, differentiate the formula by hand, or use automatic differentiation, which Chapter 14 meets in machine learning.

For measured data the lesson is even sharper. There, the "round-off" is the measurement error of the readings, which is not 10^{-16} but, for Mas Dimas's phone, around 10^{-1} m. The optimal-step argument still holds, with f replaced by the reading error δ: a larger step would reduce the effect of noise, a smaller one the effect of curvature. With readings once a second we cannot shrink the step, but we can widen it, and Section 9.7 does something even better.

9.5 Richardson Extrapolation

The error formula of the central difference says more than "order h^2". Taylor's theorem, carried further, shows that the central difference $D(h)$ has an error that is a series in *even* powers of h:

$$D(h) = f'(x) + a_1 h^2 + a_2 h^4 + a_3 h^6 + \cdots$$

where the a_i depend on the higher derivatives of f at x but not on h. Compute D twice, with h and $h/2$:

$$D(h) = f'(x) + a_1 h^2 + a_2 h^4 + \cdots, \qquad D(h/2) = f'(x) + \frac{a_1 h^2}{4} + \frac{a_2 h^4}{16} + \cdots$$

Four times the second minus the first eliminates the h^2 term:

$$\frac{4D(h/2) - D(h)}{3} = f'(x) - \frac{a_2 h^4}{4} + \cdots$$

a fourth-order estimate from two second-order ones. This is **Richardson extrapolation**, introduced by Lewis Fry Richardson in 1911 for his weather computations [5]. Nothing about it is specific to derivatives: whenever an approximation has an error that is a known power series in h, combining two step sizes cancels the leading term. Repeating the trick gives a triangular table. If $D_{i,0} = D(h/2^i)$, then

$$D_{i,j} = \frac{4^j D_{i,j-1} - D_{i-1,j-1}}{4^j - 1}, \qquad j = 1, 2, \ldots, i, \tag{9.7}$$

and column j has error $O(h^{2j+2})$. Chapter 10 builds exactly the same table for integrals, where it is called Romberg integration.

A small check by hand shows the first step at work. The central difference of $\sqrt{x}$ at 4 with $h = 0.8$ is $\left(\sqrt{4.8} - \sqrt{3.2}\right)/1.6 = (2.190890 - 1.788854)/1.6 = 0.251272$, and with $h = 0.4$ it is $\left(\sqrt{4.4} - \sqrt{3.6}\right)/0.8 = (2.097618 - 1.897367)/0.8 = 0.250314$. Their errors are 1.27×10^{-3} and 3.14×10^{-4}, a ratio of almost exactly 4, as $O(h^2)$ predicts. The extrapolation gives $(4 \times 0.250314 - 0.251272)/3 = 0.249994$, with an error of about 6×10^{-6}: more than fifty times better than the better of the two estimates, from the same four square roots.

Listing 9.3 builds the table of Equation (9.7) with four levels, starting from the deliberately coarse step $h = 0.8$.

Listing 9.3 `richardson.py`: the Richardson table for the central difference.

```
"""Richardson extrapolation of the central difference for f(x) = sqrt(x)
at x = 4 (exact derivative 0.25)."""
import math


def central(f, x, h):
    return (f(x + h) - f(x - h)) / (2 * h)


def richardson(f, x, h, levels):
    """Row i starts with D(h/2^i); column j removes the h^(2j) error term."""
    table = []
    for i in range(levels):
        row = [central(f, x, h / 2 ** i)]
        for j in range(1, i + 1):
            factor = 4 ** j
            row.append((factor * row[j - 1] - table[i - 1][j - 1])
                       / (factor - 1))
        table.append(row)
    return table


def main():
    exact = 0.25
    table = richardson(math.sqrt, 4.0, 0.8, 4)
    for i, row in enumerate(table):
        cells = "  ".join(f"{d:.12f}" for d in row)
        print(f"h = {0.8 / 2 ** i:.3f}  {cells}")
    print("errors of the diagonal:")
    for i, row in enumerate(table):
        print(f"  level {i}: {abs(row[i] - exact):.2e}")


if __name__ == "__main__":
    main()
```

Listing 9.4 `Richardson.java`: the same table in Java.

```java
import java.util.Locale;
import java.util.function.DoubleUnaryOperator;

/** Richardson extrapolation of the central difference for f(x) = sqrt(x)
 *  at x = 4 (exact derivative 0.25). */
public class Richardson {
    static double central(DoubleUnaryOperator f, double x, double h) {
        return (f.applyAsDouble(x + h) - f.applyAsDouble(x - h)) / (2 * h);
    }

    /** Row i starts with D(h/2^i); column j removes the h^(2j) error term. */
    static double[][] richardson(DoubleUnaryOperator f, double x, double h,
                                 int levels) {
        double[][] table = new double[levels][];
        for (int i = 0; i < levels; i++) {
            table[i] = new double[i + 1];
            table[i][0] = central(f, x, h / Math.pow(2, i));
            for (int j = 1; j <= i; j++) {
                double factor = Math.pow(4, j);
                table[i][j] = (factor * table[i][j - 1] - table[i - 1][j - 1])
                              / (factor - 1);
            }
        }
        return table;
    }

    public static void main(String[] args) {
        double exact = 0.25;
        double[][] table = richardson(Math::sqrt, 4.0, 0.8, 4);
        for (int i = 0; i < table.length; i++) {
            StringBuilder cells = new StringBuilder();
            for (int j = 0; j < table[i].length; j++) {
                cells.append(j > 0 ? "  " : "")
                     .append(String.format(Locale.ROOT, "%.12f", table[i][j]));
            }
            System.out.println(String.format(Locale.ROOT, "h = %.3f  %s",
                0.8 / Math.pow(2, i), cells));
        }
        System.out.println("errors of the diagonal:");
        for (int i = 0; i < table.length; i++) {
            System.out.println(String.format(Locale.ROOT, "  level %d: %.2e",
                i, Math.abs(table[i][i] - exact)));
        }
    }
}
```

Listing 9.4 prints the same table:

```
h = 0.800  0.251272405013
h = 0.400  0.250313875299  0.249994365394
h = 0.200  0.250078210575  0.249999655667  0.250000008352
h = 0.100  0.250019536593  0.249999978598  0.250000000127  0.249999999996
errors of the diagonal:
  level 0: 1.27e-03
  level 1: 5.63e-06
  level 2: 8.35e-09
  level 3: 3.55e-12
```

The first column, the plain central differences, gains a factor of four per row. The diagonal gains far more: from 1.27×10^{-3} to 3.55×10^{-12} with eight square roots in all, none with a step smaller than 0.1. Compare that with **Figure 9.1**, where the best a single central difference could do was about 10^{-12} with $h = 10^{-5}$, balanced on the edge of round-off. Richardson extrapolation reaches high accuracy with *large* steps, which keeps round-off harmless. That is its real value.

It has a price: the error series must really exist, so f must be smooth, and the extrapolation must be applied to values that are accurate enough for their differences to mean something. On noisy data, such as the trip log, extrapolation amplifies the noise; its weight $4/3$ on one estimate and $-1/3$ on the other enlarges random errors. Richardson is a tool for functions you can evaluate, not for readings you are given.

9.6 Building the Trip Analyzer

Back to Mas Dimas. His trip log, `data/trip.csv`, has a header line and then 41 lines of the form `t,distance`, from `0,0.0` to `40,305.4`. The log in the book's code folder was prepared for this chapter to match a trip of exactly the kind he describes, so we know what really happened, which is what makes it a good test: he rode at about 40 km/h (11 m/s), braked hard for the cat for three seconds from $t = 17$, rode on at about 30 km/h, and from $t = 34$ rolled gently to a stop at the customer's gate, at about -1.4 m/s^2. The phone's distance is uncertain by about 0.3 m and is stored to a tenth of a meter.

The analyzer needs three pieces: a reader for the file, a function for the speeds and a function for the accelerations. The speed uses the central difference Equation (9.3) at every reading that has a neighbor on both sides, and the three-point one-sided formula Equation (9.5) at the first and last readings, so that every value is second-order accurate. The acceleration uses the central second difference Equation (9.4) inside and the four-point one-sided formula of **Table 9.2** at the ends. Readings more negative than the threshold `HARSH` are flagged. **Listing 9.5** is the whole program.

Listing 9.5 `speed.py`: speed and acceleration from the distance log, with harsh braking flagged.

```
"""Speed and acceleration of an ojek trip from the distance the phone logged
once a second. Usage: python speed.py [csv file]  (default ../data/trip.csv)"""
import sys

HARSH = -3.0   # m/s^2: braking harder than this is reported


def load(path):
    times, dist = [], []
    with open(path) as f:
        next(f)                                   # skip the header line
        for line in f:
            t, s = line.strip().split(",")
            times.append(float(t))
            dist.append(float(s))
    return times, dist


def speeds(s, h):
    """First derivative: central inside, three-point one-sided at the ends."""
    n = len(s)
    v = [0.0] * n
    v[0] = (-3 * s[0] + 4 * s[1] - s[2]) / (2 * h)
    for i in range(1, n - 1):
        v[i] = (s[i + 1] - s[i - 1]) / (2 * h)
    v[n - 1] = (3 * s[n - 1] - 4 * s[n - 2] + s[n - 3]) / (2 * h)
    return v


def accelerations(s, h):
    """Second derivative: central inside, four-point one-sided at the ends."""
    n = len(s)
    a = [0.0] * n
    a[0] = (2 * s[0] - 5 * s[1] + 4 * s[2] - s[3]) / (h * h)
    for i in range(1, n - 1):
        a[i] = (s[i + 1] - 2 * s[i] + s[i - 1]) / (h * h)
    a[n - 1] = (2 * s[n - 1] - 5 * s[n - 2] + 4 * s[n - 3] - s[n - 4]) / (h * h)
    return a
```

Listing 9.5 (Continued) *speed.py: speed and acceleration from the distance log, with harsh braking flagged.*

```
def main(args):
    path = args[0] if args else "../data/trip.csv"
    t, s = load(path)
    h = t[1] - t[0]
    v = speeds(s, h)
    a = accelerations(s, h)
    print("  t     s(m)    v(m/s)  v(km/h)  a(m/s^2)")
    for i in range(len(t)):
        flag = "  harsh braking" if a[i] < HARSH else ""
        print(f"{t[i]:3.0f}  {s[i]:7.1f}  {v[i]:7.2f}  {v[i] * 3.6:7.1f}"
              f"  {a[i]:8.2f}{flag}")
    top = max(range(len(v)), key=lambda i: v[i])
    print(f"top speed {v[top] * 3.6:.1f} km/h at t = {t[top]:.0f} s")


if __name__ == "__main__":
    main(sys.argv[1:])
```

Three details deserve a word. The step `h` is taken from the file (`t[1] - t[0]`) instead of being written into the program, so a log taken every two seconds works unchanged. The functions `speeds` and `accelerations` take a plain list and a step and know nothing about ojeks; they will be reused by the smoothing program and by the tests. And the top speed is found with `max` over the

indices, so that the time at which it happened can be printed too. **Listing 9.6** is the Java version; `load` returns the two columns as a two-row array.

Listing 9.6 `Speed.java`: the same analyzer in Java.

```java
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;
import java.util.Locale;

/** Speed and acceleration of an ojek trip from the distance the phone logged
 *  once a second. Usage: java Speed [csv file]  (default ../data/trip.csv) */
public class Speed {
    static final double HARSH = -3.0;   // m/s^2: harder braking is reported

    /** Returns {times, distances}. */
    static double[][] load(String path) throws IOException {
        List<String> lines = Files.readAllLines(Path.of(path));
        int n = lines.size() - 1;                    // the first line is a header
        double[] t = new double[n], s = new double[n];
        for (int i = 0; i < n; i++) {
            String[] parts = lines.get(i + 1).trim().split(",");
            t[i] = Double.parseDouble(parts[0]);
            s[i] = Double.parseDouble(parts[1]);
        }
        return new double[][] {t, s};
    }

    /** First derivative: central inside, three-point one-sided at the ends. */
    static double[] speeds(double[] s, double h) {
        int n = s.length;
        double[] v = new double[n];
        v[0] = (-3 * s[0] + 4 * s[1] - s[2]) / (2 * h);
        for (int i = 1; i < n - 1; i++) {
            v[i] = (s[i + 1] - s[i - 1]) / (2 * h);
        }
        v[n - 1] = (3 * s[n - 1] - 4 * s[n - 2] + s[n - 3]) / (2 * h);
        return v;
    }
```

Listing 9.6 (Continued) *Speed.java: the same analyzer in Java.*

```
    /** Second derivative: central inside, four-point one-sided at the ends. */
    static double[] accelerations(double[] s, double h) {
        int n = s.length;
        double[] a = new double[n];
        a[0] = (2 * s[0] - 5 * s[1] + 4 * s[2] - s[3]) / (h * h);
        for (int i = 1; i < n - 1; i++) {
            a[i] = (s[i + 1] - 2 * s[i] + s[i - 1]) / (h * h);
        }
        a[n - 1] = (2 * s[n - 1] - 5 * s[n - 2] + 4 * s[n - 3] - s[n - 4])
                   / (h * h);
        return a;
    }

    public static void main(String[] args) throws IOException {
        double[][] data = load(args.length > 0 ? args[0] : "../data/trip.csv");
        double[] t = data[0], s = data[1];
        double h = t[1] - t[0];
        double[] v = speeds(s, h), a = accelerations(s, h);
        System.out.println("  t     s(m)   v(m/s)  v(km/h)  a(m/s^2)");
        int top = 0;
        for (int i = 0; i < t.length; i++) {
            String flag = a[i] < HARSH ? "  harsh braking" : "";
            System.out.println(String.format(Locale.ROOT,
                "%3.0f  %7.1f  %7.2f  %7.1f  %8.2f%s",
                t[i], s[i], v[i], v[i] * 3.6, a[i], flag));
            if (v[i] > v[top]) {
                top = i;
            }
        }
        System.out.println(String.format(Locale.ROOT,
            "top speed %.1f km/h at t = %.0f s", v[top] * 3.6, t[top]));
    }
}
```

Run from the `python` or `java` folder, both print the same 43 lines:

```
  t     s(m)   v(m/s)  v(km/h)  a(m/s^2)
  0      0.0    -0.90     -3.2      3.90
  1      0.6     2.10      7.6      3.00
  2      4.2     4.65     16.7      2.10
  3      9.9     6.40     23.0      1.40
  4     17.0     8.70     31.3      3.20
  5     27.3    11.15     40.1      1.70
  6     39.3    10.85     39.1     -2.30
  7     49.0    10.50     37.8      1.60
  8     60.3    11.25     40.5     -0.10
  9     71.5    10.75     38.7     -0.90
 10     81.8    11.00     39.6      1.40
 11     93.5    11.50     41.4     -0.40
 12    104.8    10.95     39.4     -0.70
 13    115.4    10.65     38.3      0.10
 14    126.1    11.05     39.8      0.70
 15    137.5    11.05     39.8     -0.70
 16    148.2    11.00     39.6      0.60
 17    159.5    10.25     36.9     -2.10
 18    168.7     7.45     26.8     -3.50  harsh braking
 19    174.4     3.90     14.0     -3.60  harsh braking
 20    176.5     2.15      7.7      0.10
 21    178.7     2.65      9.5      0.90
 22    181.8     4.40     15.8      2.60
 23    187.5     6.50     23.4      1.60
 24    194.8     7.85     28.3      1.10
 25    203.2     8.55     30.8      0.30
 26    211.9     8.55     30.8     -0.30
 27    220.3     8.50     30.6      0.20
 28    228.9     8.50     30.6     -0.20
 29    237.3     8.35     30.1     -0.10
 30    245.6     8.50     30.6      0.40
 31    254.3     8.65     31.1     -0.10
 32    262.9     8.65     31.1      0.10
 33    271.6     8.30     29.9     -0.80
 34    279.5     8.15     29.3      0.50
 35    287.9     6.85     24.7     -3.10  harsh braking
 36    293.2     5.35     19.3      0.10
 37    298.6     4.60     16.6     -1.60
 38    302.4     2.90     10.4     -1.80
 39    304.4     1.50      5.4     -1.00
 40    305.4     0.50      1.8     -0.20
top speed 41.4 km/h at t = 11 s
```

Read the table the way Mas Dimas would. The speed climbs to about 40 km/h in five seconds and stays there, wobbling between 38 and 41; falls to under 8 km/h at $t = 20$; climbs again to about 31 km/h; and falls to almost zero at the end. The acceleration at $t = 18$ and $t = 19$ is -3.50 and -3.60 m/s^2, flagged: that is the cat, and the app was right. But there is a third flag, at $t = 35$, during the gentle stop at the gate, where the true deceleration was only -1.4 m/s^2. The app on his phone probably made the same mistake. Mas Dimas was right too.

The table shows the other symptoms of noise that Section 9.3 predicted. The speed at $t = 0$ is -0.90 m/s, which is impossible: the one-sided formula at the start has weights $-3, 4, -1$, and a noise of 0.3 m in the reading at $t = 1$ alone shifts it by $4 \times 0.3/2 = 0.6$ m/s. The top speed, 41.4 km/h at $t = 11$, is a lucky wobble of the noise, not a moment of real speeding; the true cruising speed was 39.6 km/h. And the acceleration column jumps about by one or two meters per second squared during the steady cruise between $t = 6$ and $t = 16$, where the true acceleration was zero; the value -2.30 at $t = 6$ came close to a second false alarm. None of this is a programming error. It is the arithmetic of Equation (9.4): an error δ in the readings becomes an error up to $4\delta/h^2$ in the acceleration.

We can be more precise. If the reading errors are independent with standard deviation σ, the error of $s_{i+1} - 2s_i + s_{i-1}$ has variance $(1 + 4 + 1)\sigma^2 = 6\sigma^2$, so the computed acceleration has a standard deviation of $\sqrt{6}\,\sigma/h^2 \approx 2.45 \times 0.3 = 0.73$ m/s^2. During the gentle stop the true value -1.42 is only 1.58 m/s^2, a bit more than two standard deviations, above the threshold, so each of those six readings has a chance of one or two percent of being flagged by noise alone. Over the many stops of a working day, false alarms are not bad luck; they are certain.

9.7 Making It Better: Smooth Before You Differentiate

9.7.1 Fit a parabola, then differentiate the parabola

The central difference differentiates the parabola through three neighboring readings. Through noisy readings that parabola follows the noise faithfully. A better idea, due to Savitzky and Golay in 1964 [6], is to fit a parabola through *five* neighboring readings by least squares, the method of Chapter 8, and differentiate the fitted parabola instead. The fit cannot pass through all five points, so it ignores part of the noise, and the derivative of a parabola is easy.

Measure time from the middle reading in steps: $\tau = -2, -1, 0, 1, 2$, with readings $s_{-2}, \dots, s_2$, and fit $p(\tau) = c_0 + c_1\tau + c_2\tau^2$. The normal equations of Chapter 8 involve the sums $\sum\tau^k$ over the five points: $\sum 1 = 5$, $\sum\tau = 0$, $\sum\tau^2 = 10$, $\sum\tau^3 = 0$ and $\sum\tau^4 = 34$. Because the odd sums vanish, the equation for c_1 separates from the others:

$$c_1 = \frac{\sum \tau s_\tau}{\sum \tau^2} = \frac{-2s_{-2} - s_{-1} + s_1 + 2s_2}{10},$$

and c_0 and c_2 solve $5c_0 + 10c_2 = \sum s_\tau$ and $10c_0 + 34c_2 = \sum \tau^2 s_\tau$, which gives $c_2 = (\sum \tau^2 s_\tau - 2\sum s_\tau)/14$. The derivatives of p at $\tau = 0$, converted from steps to seconds, are c_1/h and $2c_2/h^2$:

$$v \approx \frac{-2s_{-2} - s_{-1} + s_1 + 2s_2}{10h}, \qquad a \approx \frac{2s_{-2} - s_{-1} - 2s_0 - s_1 + 2s_2}{7h^2}. \tag{9.8}$$

These are fixed weighted sums of five readings, as cheap as the plain formulas. Their effect on noise is easy to compute. For the speed, the variance factor is $(4 + 1 + 1 + 4)/100 = 0.1$ instead of 0.5 for the central difference; for the acceleration, $(4 + 1 + 4 + 1 + 4)/49 = 0.286$ instead of 6. The standard deviation of the noise in the acceleration falls from 0.73 to 0.16 m/s^2, more than four times smaller.

Nothing is free. A parabola through five points spanning four seconds cannot follow a change that happens within one second, so a short hard brake is spread out and looks a little milder. That is the same trade as the optimal step of Section 9.4: less noise, more truncation. For a brake of three seconds, as here, the trade is good.

9.7.2 The smoothed analyzer

Listing 9.7 computes Equation (9.8) at every reading that has two neighbors on each side, prints it next to the plain values, and lists the harsh-braking moments found each way. It imports `load`, `speeds`, `accelerations` and `HARSH` from `speed.py`, so the two programs cannot drift apart.

Listing 9.7 `smooth.py`: speed and acceleration from a five-point least-squares parabola, next to the plain formulas.

```python
"""Speed and acceleration from a local least-squares parabola through five
points (Savitzky-Golay), compared with the plain three-point formulas.
Usage: python smooth.py [csv file]  (default ../data/trip.csv)"""
import sys
from speed import HARSH, load, speeds, accelerations


def smooth_speed(s, i, h):
    return (-2 * s[i - 2] - s[i - 1] + s[i + 1] + 2 * s[i + 2]) / (10 * h)


def smooth_accel(s, i, h):
    return (2 * s[i - 2] - s[i - 1] - 2 * s[i] - s[i + 1]
            + 2 * s[i + 2]) / (7 * h * h)


def main(args):
    path = args[0] if args else "../data/trip.csv"
    t, s = load(path)
    h = t[1] - t[0]
    v, a = speeds(s, h), accelerations(s, h)
    plain, smooth = [], []
    print("  t   v plain  v smooth   a plain  a smooth")
    for i in range(2, len(t) - 2):
        vs, as_ = smooth_speed(s, i, h), smooth_accel(s, i, h)
        print(f"{t[i]:3.0f}  {v[i]:7.2f}  {vs:8.2f}  {a[i]:8.2f}  {as_:8.2f}")
        if a[i] < HARSH:
            plain.append(f"{t[i]:.0f}")
        if as_ < HARSH:
            smooth.append(f"{t[i]:.0f}")
    print("harsh braking, plain formulas:  t =", ", ".join(plain))
    print("harsh braking, smoothed:        t =", ", ".join(smooth))


if __name__ == "__main__":
    main(sys.argv[1:])
```

Listing 9.8 `Smooth.java`: the same comparison in Java.

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;

/** Speed and acceleration from a local least-squares parabola through five
 *  points (Savitzky-Golay), compared with the plain three-point formulas.
 *  Usage: java Smooth [csv file]  (default ../data/trip.csv) */
public class Smooth {
    static double smoothSpeed(double[] s, int i, double h) {
        return (-2 * s[i - 2] - s[i - 1] + s[i + 1] + 2 * s[i + 2]) / (10 * h);
    }

    static double smoothAccel(double[] s, int i, double h) {
        return (2 * s[i - 2] - s[i - 1] - 2 * s[i] - s[i + 1]
                + 2 * s[i + 2]) / (7 * h * h);
    }

    public static void main(String[] args) throws IOException {
        String path = args.length > 0 ? args[0] : "../data/trip.csv";
        double[][] data = Speed.load(path);
        double[] t = data[0], s = data[1];
        double h = t[1] - t[0];
        double[] v = Speed.speeds(s, h), a = Speed.accelerations(s, h);
        List<String> plain = new ArrayList<>(), smooth = new ArrayList<>();
        System.out.println("  t   v plain  v smooth   a plain  a smooth");
        for (int i = 2; i < t.length - 2; i++) {
            double vs = smoothSpeed(s, i, h), as = smoothAccel(s, i, h);
            System.out.println(String.format(Locale.ROOT,
                "%3.0f  %7.2f  %8.2f  %8.2f  %8.2f", t[i], v[i], vs, a[i], as));
            if (a[i] < Speed.HARSH) {
                plain.add(String.format(Locale.ROOT, "%.0f", t[i]));
            }
            if (as < Speed.HARSH) {
                smooth.add(String.format(Locale.ROOT, "%.0f", t[i]));
            }
        }
        System.out.println("harsh braking, plain formulas:  t = "
                           + String.join(", ", plain));
        System.out.println("harsh braking, smoothed:        t = "
                           + String.join(", ", smooth));
    }
}
```

Listing 9.8 reuses `Speed.load`, `Speed.speeds` and `Speed.accelerations` in the same way. Both print:

```
  t   v plain  v smooth   a plain  a smooth
  2     4.65      4.33      2.10      2.16
  3     6.40      6.62      1.40      2.11
  4     8.70      8.76      3.20      2.26
  5    11.15     10.05      1.70      0.99
  6    10.85     10.83     -2.30     -0.04
  7    10.50     10.94      1.60      0.00
  8    11.25     10.75     -0.10      0.16
  9    10.75     11.05     -0.90     -0.01
 10    11.00     11.10      1.40      0.23
 11    11.50     11.08     -0.40      0.03
 12    10.95     11.05     -0.70     -0.39
 13    10.65     10.93      0.10      0.04
 14    11.05     10.89      0.70      0.13
 15    11.05     11.03     -0.70      0.07
 16    11.00     10.72      0.60     -0.54
 17    10.25      9.43     -2.10     -1.73
 18     7.45      7.15     -3.50     -3.13
 19     3.90      4.62     -3.60     -2.51
 20     2.15      3.05      0.10     -0.73
 21     2.65      3.15      0.90      1.16
 22     4.40      4.54      2.60      1.83
 23     6.50      6.20      1.60      1.74
 24     7.85      7.59      1.10      1.01
 25     8.55      8.27      0.30      0.36
 26     8.55      8.53     -0.30      0.01
 27     8.50      8.52      0.20     -0.06
 28     8.50      8.44     -0.20     -0.06
 29     8.35      8.47     -0.10      0.01
 30     8.50      8.50      0.40      0.11
 31     8.65      8.59     -0.10      0.10
 32     8.65      8.51      0.10     -0.21
 33     8.30      8.38     -0.80     -0.17
 34     8.15      7.69      0.50     -0.90
 35     6.85      6.77     -3.10     -1.16
 36     5.35      5.65      0.10     -1.30
 37     4.60      4.22     -1.60     -1.17
 38     2.90      3.02     -1.80     -1.51
harsh braking, plain formulas:  t = 18, 19, 35
harsh braking, smoothed:        t = 18
```

The smoothed acceleration during the cruise stays within about ± 0.4 m/s^2 of zero, where the plain one swung between -2.3 and $+1.6$. The hard brake is still found, at $t = 18$ with -3.13 m/s^2, a little milder than the true -3.5 because the fit spreads it out, but clearly above the threshold. And the false alarm at $t = 35$ has become -1.16 m/s^2, close to the true -1.4: a gentle stop, as Mas Dimas said. **Figure 9.2** shows the speed and both acceleration curves; the plain curve wanders across the red threshold, the smoothed one crosses it only where the brake was.

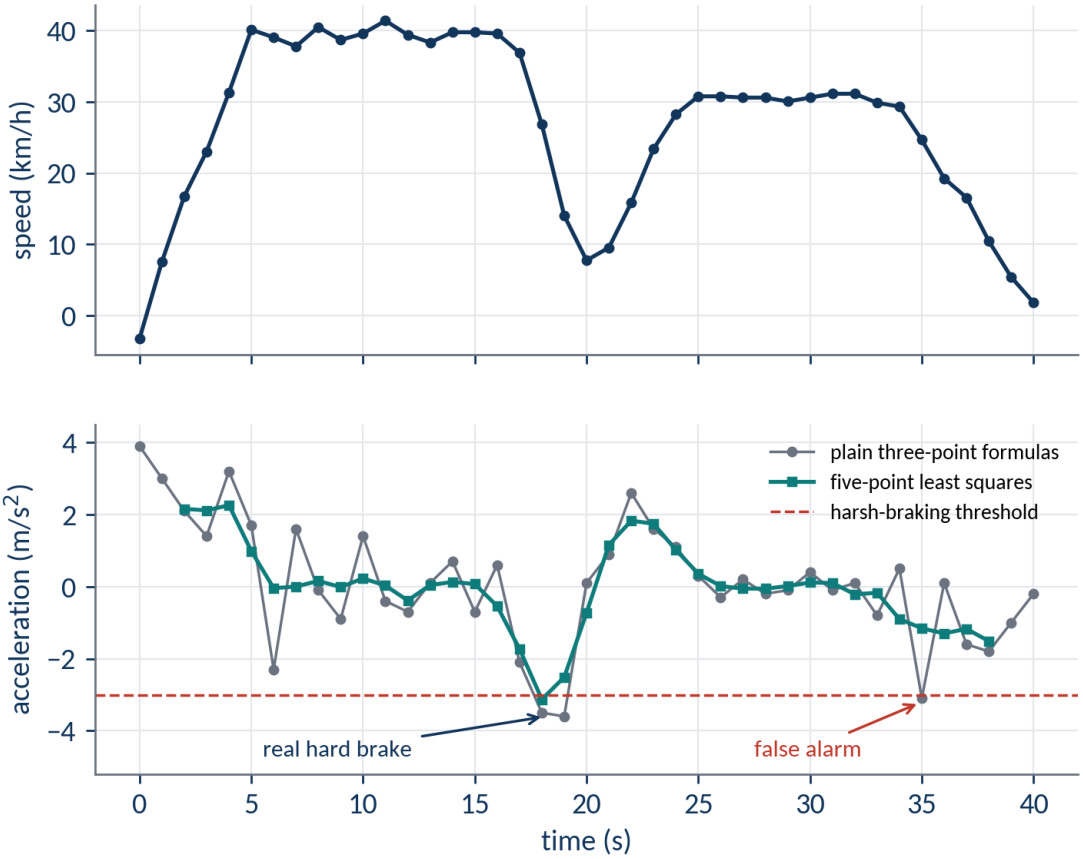


Figure 9.2 Speed of Mas Dimas's trip (top) and its acceleration (bottom) from the plain three-point formulas and from the five-point least-squares formulas; only the smoothed curve separates the real hard brake from the false alarm.

Let us check one value by hand, as the program's user should. At $t = 35$ the five readings are 271.6,279.5,287.9,293.2,298.6. By Equation (9.8), $a = (2 \times 271.6 - 279.5 - 2 \times 287.9 - 293.2 + 2 \times 298.6)/7 = (543.2 - 279.5 - 575.8 - 293.2 + 597.2)/7 = -8.1/7 = -1.157$ m/s^2, the -1.16 of the table.

> **WARNING:** A smoothing formula uses readings from the future: at $t = 18$ it needs the readings at 19 and 20. An app that must warn *while* the driver brakes cannot wait two seconds; it can only use one-sided formulas, which are noisier, or accept a short delay. Every real-time derivative faces this choice between noise and delay.

9.8 Checking Against the Library, and Testing

In real work you rarely write difference formulas by hand; the libraries have them. The current releases at the time of writing are NumPy 2.5.3, released on September 6, 2026 [7], and SciPy 1.18.1, released on August 21, 2026 [8]; both require Python 3.12 or newer. NumPy's `np.gradient` applies the central difference inside and one-sided formulas at the ends; with `edge_order=2` the ends use exactly our three-point formula Equation (9.5), and the spacing may be a number or an array of unevenly spaced coordinates [9]. SciPy, since version 1.15, has `scipy.differentiate.derivative`, which differentiates a *function* (not data) with a sequence of shrinking central differences and Richardson-style extrapolation, stopping when its own error estimate is small enough [10]. **Listing 9.9** uses both.

Listing 9.9 `compare_numpy.py`: the trip speeds from `np.gradient` and the derivative of $\sqrt{x}$ from `scipy.differentiate.derivative`.

```
"""Check this chapter's formulas against NumPy and SciPy."""
import numpy as np
import scipy
from scipy.differentiate import derivative
from speed import load, speeds, accelerations

t, s = load("../data/trip.csv")
h = t[1] - t[0]
v_ours = np.array(speeds(s, h))
v_np = np.gradient(np.array(s), h, edge_order=2)   # same formulas as ours
print(f"NumPy {np.__version__}, SciPy {scipy.__version__}")
print(f"largest speed difference, ours vs np.gradient: "
      f"{np.max(np.abs(v_ours - v_np)):.1e} m/s")
a_np = np.gradient(v_np, h, edge_order=2)          # gradient of the gradient
a_ours = np.array(accelerations(s, h))
print(f"a at t = 19 s: ours {a_ours[19]:.2f}, gradient twice {a_np[19]:.2f}")
res = derivative(np.sqrt, 4.0)                     # adaptive, with error estimate
print(f"scipy derivative of sqrt at 4: {res.df:.15f}"
      f" (estimated error {res.error:.1e}, {res.nfev} evaluations)")
```

```
NumPy 2.4.4, SciPy 1.17.1
largest speed difference, ours vs np.gradient: 0.0e+00 m/s
a at t = 19 s: ours -3.60, gradient twice -2.65
scipy derivative of sqrt at 4: 0.250000000000000 (estimated error 8.9e-14, 11 evaluations)
```

The output, from the versions installed on the machine that built this book, shows three things. The speeds from `np.gradient(..., edge_order=2)` are identical to ours, to the last bit. Applying `np.gradient` twice does *not* give our acceleration: at $t = 19$ it gives -2.65 instead of -3.60, because the gradient of the gradient is the second difference with step $2h$, $(s_{i+2} - 2s_i + s_{i-2})/(4h^2)$, a wider and smoother formula, so it spreads the brake over four seconds. Neither answer is wrong; they are different trades between noise and resolution, and you should know which one you are using. Finally, SciPy's `derivative` finds $\sqrt{x}$'s derivative at 4 correct to about 15 digits with 11 evaluations, by never letting the step become small enough for round-off to matter: the Richardson idea of Section 9.5 in industrial form.

The Java edition has no standard numerical library, so the Java programs are checked against the Python ones instead: each pair prints identical output, which the build of this book verifies character by character. The code folder also holds tests, `test_diff.py` and `DiffTest.java`, which check four facts that must hold whatever the data: the central difference is exact for a parabola; the forward error halves when h halves; Richardson's diagonal beats the plain central difference; and for a distance $s = t^2$ (constant acceleration 2 m/s^2) the analyzer returns exactly $v = 2t$ and $a = 2$ at *every* point, including the ends. The last test checks the one-sided formulas, which are the easiest to get wrong, and it works because all our formulas are exact for parabolas.

9.9 Numerical Methods in Practice: Profitina

Profitina, an Indonesian AI-visibility platform, measures how ready a brand's website is to be cited by AI answer engines and helps the owner improve it. One of its shipped features is a historical analytics dashboard: the platform re-scans each brand's site automatically at a cadence that depends on the subscription tier, stores a score per week, draws the weekly trend, exports it as CSV, and raises alerts. The questions a brand owner asks of that trend are derivative questions: *Is my score improving? How fast? Did it just drop?*

The week-to-week change of the score is a finite difference with $h = 1$ week, and everything in this chapter applies to it. The newest week has no "next week", so the latest trend is necessarily a backward, one-sided estimate, the least accurate kind, just as at the last second of Mas Dimas's trip. A single scan is also noisy for reasons unrelated to the owner's work, for example a page that happened to respond slowly, so an alert on "the score fell sharply" computed from two single weeks would produce false alarms of exactly the kind that annoyed Mas Dimas. The class of solution is the one of Section 9.7: estimate the trend from a least-squares fit over several recent weeks, accept that a real change is recognized a little later, and explain the trend to the owner in plain words instead of shouting about every wobble. How Profitina computes its scores and alerts is its own business; the shape of the problem is exactly this chapter's.

The same shape appears in Profitina's PROFIT utility coin, whose emission is designed to decay over time. The dashboard that shows *how many coins are emitted per day* from a running total of coins emitted is taking a backward difference of a cumulative curve, and a decaying curve is one whose rate of change is itself changing, which is where the choice between a one-sided and a centered formula shows. As always with Profitina, the numbers advise; the user decides.

9.10 Using It for Real

- **Know your noise first.** Before differentiating any log, estimate the error of a single reading: repeat a measurement that should not change (a parked motorbike, a full tank, a phone on the charger) and look at how the readings wobble. Then use $\sqrt{2}\,\sigma/(2h)$ and $\sqrt{6}\,\sigma/h^2$ from Section 9.6 to predict the noise in your first and second derivatives *before* trusting them.
- **Choose the spacing to suit the question.** A phone battery shows whole percents. Its drain rate from readings one minute apart is mostly quantization noise (a change of 0 or 1 per minute); readings ten minutes apart give a useful rate. A water-tank level sensor that reads to a centimeter gives a useful flow rate over minutes, not seconds. When the logger records faster than you need, use a wider step or the smoothing formulas; do not throw away the readings in between, average them.
- **Use centered formulas inside and second-order formulas at the ends.** Never mix a first-order end with second-order interior values without knowing it; `np.gradient` does exactly that unless you pass `edge_order=2`.
- **Handle missing readings honestly.** Phones skip readings. Do not pretend the spacing is still one second; use the unevenly spaced formula of Exercise 9.6, or `np.gradient` with the actual time stamps.
- **Adapt the analyzer.** For a water tank, the "distance" column becomes the level in centimeters and the first derivative is the net inflow; multiply by the tank's cross-section to get liters per minute. For a small shop, the column can be cumulative sales, and the derivative is sales per day, which tells you whether a promotion changed anything. The programs do not change; only the file and the threshold do.
- **Keep a person in the loop.** An alert computed from a derivative should say what it saw, for example "a drop of 12 km/h in one second at 07:41", so that the person who was there can agree or disagree, as Mas Dimas could.

9.11 Common Mistakes, and How to Avoid Them

- **Making h as small as possible.** Below the optimum of Equation (9.6) round-off takes over, and the derivative gets worse, not better. With doubles, use $h \approx 10^{-8}$ for forward and $h \approx 10^{-5}$ for central differences (scaled by x), or use Richardson extrapolation with larger steps.
- **Using forward differences when both neighbors exist.** The central difference costs the same and is an order more accurate. Keep one-sided formulas for the ends, and make them three-point.
- **Forgetting the square in the second derivative.** The second difference is divided by h^2, not h or $2h$. With $h = 1$ s the mistake is invisible; with a log taken every 0.5 s it makes the acceleration four times too small.
- **Differentiating noisy data twice without thinking.** Every differentiation divides the noise by the step. Estimate the noise first (Section 9.10), then smooth (Equation (9.8)) or widen the step until the noise is well below what you want to detect.
- **Believing impossible values.** A negative speed at the start of a trip or a top speed from a single reading is noise. Check derivatives against what the physics allows.
- **Assuming even spacing.** If the time stamps are 14, 15, 16, 18, 19 because one reading was lost, the ordinary formulas give wrong answers at 16 and 18. Check the spacing in the program, not in your head.
- **Calling `np.gradient` with the default edges.** The default `edge_order=1` is first order at the ends. And remember that the gradient of the gradient is not the second difference of Equation (9.4) but a wider one.

9.12 Chapter Summary

- A derivative from samples replaces the tangent by a secant. Taylor's theorem (Equation (9.1)) turns the idea into formulas with known error: forward and backward differences are first order (Equation (9.2)), the central difference is second order (Equation (9.3)), and so is the central second difference (Equation (9.4)).
- The method of undetermined coefficients gives formulas for any points, such as the second-order one-sided formula for the ends (Equation (9.5)); **Table 9.2** collects the common ones.
- Truncation error falls as h falls, round-off error grows as $1/h$; the total is smallest at an optimal step (Equation (9.6)), about 10^{-8} for the forward and 10^{-5} for the central difference in double precision.
- Richardson extrapolation (Equation (9.7)) combines estimates with different steps to cancel error terms, reaching high accuracy with steps large enough to keep round-off harmless.
- For measured data the "round-off" is the measurement noise, amplified by $1/h$ for speed and $1/h^2$ for acceleration. On Mas Dimas's trip it caused a negative starting speed, a false top speed and a false harsh-braking alarm.
- Differentiating a least-squares parabola through five points (Equation (9.8)) reduced the noise in the acceleration more than fourfold, kept the real hard brake and removed the false alarm.
- `np.gradient` with `edge_order=2` reproduces our speeds exactly; `scipy.differentiate.derivative` differentiates functions adaptively.

9.13 Exercises

1. By hand, from the readings at $t = 34,35,36$ (279.5, 287.9, 293.2 m), compute the plain speed and acceleration at $t = 35$. Explain in terms of reading errors how a gentle stop of about -1.42 m/s^2 could produce this acceleration. The true distances at those times were 279.75, 287.54 and 293.92 m; what were the reading errors, and how do they add up?
2. Show that one Richardson step applied to the central difference, $(4D(h/2) - D(h))/3$, is exactly the five-point central formula of **Table 9.2** with step $h/2$. What is its order of accuracy?
3. For $f(x) = e^x$ at $x = 0$, use Equation (9.6) with $\varepsilon = 2^{-52}$ to predict the best step and the smallest error for the forward and for the central difference. Check your prediction with a small program.
4. The forward difference has an error series $D(h) = f'(x) + b_1 h + b_2 h^2 + \cdots$. Derive the Richardson combination that removes the $b_1 h$ term, apply it to $\sqrt{x}$ at 4 with $h = 0.1$ and $h = 0.05$, and compare its error with those of the two forward differences.
5. Mas Dimas's phone battery shows 100, 97, 93, 90, 86, 81 and 77 percent at 0, 10, 20, ..., 60 minutes of riding with the map on. Estimate the drain rate in percent per minute at every reading, using second-order formulas throughout. How long will the remaining 77% last? Why is the one-sided estimate at 60 minutes a poor basis for that answer, and what is a better one?
6. On another trip the phone missed the reading at $t = 17$, so the log has times 14, 15, 16, 18, 19, 20 with distances 126.1, 137.5, 148.2, 168.7, 174.4, 176.5 m. Derive the derivative of the parabola through three unevenly spaced points at the middle point, and write programs in Python and Java that print the speed at the four interior times.

9.14 Solutions to the Exercises

1. Speed: $v = (293.2 - 279.5)/2 = 6.85$ m/s. Acceleration: $a = 293.2 - 2 \times 287.9 + 279.5 = -3.1$ m/s^2, just beyond the threshold, as in the table of Section 9.6. The true value was about -1.42, so the reading errors contributed about -1.7 m/s^2. If the three readings have errors e_{34}, e_{35}, e_{36}, the second difference changes by $e_{34} - 2e_{35} + e_{36}$, because the formula has weights $1, -2, 1$. Here the errors were $e_{34} = 279.5 - 279.75 = -0.25$, $e_{35} = 287.9 - 287.54 = +0.36$ and $e_{36} = 293.2 - 293.92 = -0.72$ m, so the change was $-0.25 - 0.72 - 0.72 = -1.69$ m/s^2, and $-1.42 - 1.69 \approx -3.1$. The worst possible pattern, a middle reading too high between two readings too low, is exactly what happened; with a noise of about 0.3 m such a pattern, and even an error of 0.7 m in one reading, is unusual but far from rare over a long log. The five-point formula gives -1.16 (Section 9.7.2).

2. With $D(h) = \frac{f(x+h) - f(x-h)}{2h}$ and $D(h/2) = \frac{f(x+h/2) - f(x-h/2)}{h}$,

$$\frac{4D(h/2) - D(h)}{3} = \frac{8\left(f\left(x + \frac{h}{2}\right) - f\left(x - \frac{h}{2}\right)\right) - (f(x+h) - f(x-h))}{6h}.$$

Writing $k = h/2$, so that $6h = 12k$, this is $\frac{f(x-2k) - 8f(x-k) + 8f(x+k) - f(x+2k)}{12k}$, the five-point formula with step k. Richardson removed the h^2 term, and the error series of the central difference has only even powers, so the result is fourth order, $O(k^4)$, in agreement with the error $\frac{k^4}{30} f^{(5)}$ in **Table 9.2**.

3. For e^x at 0 all derivatives and \$f\$ equal 1. Forward: $h^* = 2\sqrt{\varepsilon} = 2.98 \times 10^{-8}$, and the error bound there is $\frac{h^*}{2} + \frac{2\varepsilon}{h^*} = 1.49 \times 10^{-8} + 1.49 \times 10^{-8} \approx 3.0 \times 10^{-8}$. Central: $h^* = (3\varepsilon)^{1/3} = 8.7 \times 10^{-6}$, and the bound is $\frac{(h^*)^2}{6} + \frac{\varepsilon}{h^*} \approx 1.3 \times 10^{-11} + 2.5 \times 10^{-11} \approx 3.8 \times 10^{-11}$. Changing `diff_steps.py` to use `math.exp` at 0 with exact value 1 and these two steps gives actual errors of 1.6×10^{-8} (forward) and 1.4×10^{-11} (central), below the bounds and of the predicted size. Our estimates are bounds, so actual errors are usually somewhat smaller.

4. $D(h/2) = f'(x) + b_1 h/2 + b_2 h^2/4 + \cdots$. Then $2D(h/2) - D(h) = f'(x) - b_2 h^2/2 + \cdots$: the combination $2D(h/2) - D(h)$ is second order. With the forward values of Section 9.4, $D(0.1) = 0.2484567313$ and $D(0.05) = 0.2492235950$ (errors 1.54×10^{-3} and 7.76×10^{-4}), the combination gives $2 \times 0.2492235950 - 0.2484567313 = 0.2499904587$, with error 9.5×10^{-6}, eighty times smaller than the better forward difference. It is in fact the three-point forward formula Equation (9.5) with step 0.05.

5. The step is $h = 10$ min. Inside, the central difference gives: at 10 min $(93 - 100)/20 = -0.35$; at 20 min $(90 - 97)/20 = -0.35$; at 30 min $(86 - 93)/20 = -0.35$; at 40 min $(81 - 90)/20 = -0.45$; at 50 min $(77 - 86)/20 = -0.45$ percent per minute. At 0 min the three-point forward formula gives $(-3 \times 100 + 4 \times 97 - 93)/20 = -0.25$, and at 60 min the three-point backward formula gives $(3 \times 77 - 4 \times 81 + 86)/20 = -0.35$. The drain has become faster in the second half hour, but the one-sided value at 60 minutes, -0.35, suggests the opposite; it is distorted by the whole-percent display, whose error of up to half a percent is multiplied by the weights $3, -4, 1$. A better basis is the least-squares slope over the last four readings (30 to 60 minutes: 90, 86, 81, 77). With $\tau = -1.5, -0.5, 0.5, 1.5$ steps, the slope is $\sum \tau s_\tau / \sum \tau^2 = (-135 - 43 + 40.5 + 115.5)/5 = -4.4$ percent per 10 minutes, or -0.44 per minute. The remaining 77% then last about $77/0.44 \approx 175$ minutes, just under three hours, if he keeps riding the same way.

6. Let the three points be $t_0 < t_1 < t_2$ with $h_1 = t_1 - t_0$ and $h_2 = t_2 - t_1$. By undetermined coefficients (or by differentiating the Lagrange form of Chapter 7), the derivative at t_1 of the parabola through the three points is

$$f'(t_1) \approx -\frac{h_2}{h_1(h_1 + h_2)} f_0 + \frac{h_2 - h_1}{h_1 h_2} f_1 + \frac{h_1}{h_2(h_1 + h_2)} f_2,$$

which reduces to the central difference when $h_1 = h_2$ and is second order in general. **Listing 9.10** and **Listing 9.11** implement it.

Listing 9.10 `uneven.py`: speeds from unevenly spaced readings.

```python
"""Exercise 9.6: speed from readings that are not evenly spaced (the phone
missed the reading at t = 17 s)."""


def speed_uneven(t, s, i):
    """Derivative of the parabola through points i-1, i, i+1 at t[i]."""
    h1, h2 = t[i] - t[i - 1], t[i + 1] - t[i]
    return (-h2 / (h1 * (h1 + h2)) * s[i - 1]
            + (h2 - h1) / (h1 * h2) * s[i]
            + h1 / (h2 * (h1 + h2)) * s[i + 1])


def main():
    t = [14.0, 15.0, 16.0, 18.0, 19.0, 20.0]
    s = [126.1, 137.5, 148.2, 168.7, 174.4, 176.5]
    for i in range(1, len(t) - 1):
        print(f"t = {t[i]:4.1f} s   v = {speed_uneven(t, s, i):6.2f} m/s")


if __name__ == "__main__":
    main()
```

Listing 9.11 `Uneven.java`: the same computation in Java.

```java
import java.util.Locale;

/** Exercise 9.6: speed from readings that are not evenly spaced (the phone
 *  missed the reading at t = 17 s). */
public class Uneven {
    /** Derivative of the parabola through points i-1, i, i+1 at t[i]. */
    static double speedUneven(double[] t, double[] s, int i) {
        double h1 = t[i] - t[i - 1], h2 = t[i + 1] - t[i];
        return -h2 / (h1 * (h1 + h2)) * s[i - 1]
               + (h2 - h1) / (h1 * h2) * s[i]
               + h1 / (h2 * (h1 + h2)) * s[i + 1];
    }

    public static void main(String[] args) {
        double[] t = {14.0, 15.0, 16.0, 18.0, 19.0, 20.0};
        double[] s = {126.1, 137.5, 148.2, 168.7, 174.4, 176.5};
        for (int i = 1; i < t.length - 1; i++) {
            System.out.println(String.format(Locale.ROOT,
                "t = %4.1f s   v = %6.2f m/s", t[i], speedUneven(t, s, i)));
        }
    }
}
```

Both print:

```
t = 15.0 s   v =  11.05 m/s
t = 16.0 s   v =  10.55 m/s
t = 18.0 s   v =   7.22 m/s
t = 19.0 s   v =   3.90 m/s
```

At $t = 16$ the formula weighs the reading one second back more than the one two seconds ahead, and gives 10.55 m/s; the ordinary central difference, applied as if the readings were evenly spaced, would have given $(168.7 - 137.5)/2 = 15.6$ m/s, a speed of 56 km/h that never happened.

References

[1] S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers*, 8th ed. New York, NY, USA: McGraw-Hill Education, 2021.

[2] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical Analysis*, 10th ed. Boston, MA, USA: Cengage Learning, 2016.

[3] B. Fornberg, “Generation of finite difference formulas on arbitrarily spaced grids,” *Mathematics of Computation*, vol. 51, no. 184, pp. 699–706, 1988.

[4] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: SIAM, 2002.

[5] L. F. Richardson, “The approximate arithmetical solution by finite differences of physical problems involving differential equations, with an application to the stresses in a masonry dam,” *Philosophical Transactions of the Royal Society A*, vol. 210, pp. 307–357, 1911.

[6] A. Savitzky and M. J. E. Golay, “Smoothing and differentiation of data by simplified least squares procedures,” *Analytical Chemistry*, vol. 36, no. 8, pp. 1627–1639, 1964.

[7] NumPy Developers, “numpy 2.5.3,” Python Package Index, Sep. 6, 2026. [Online]. Available: https://pypi.org/project/numpy/

[8] SciPy Developers, “scipy 1.18.1,” Python Package Index, Aug. 21, 2026. [Online]. Available: https://pypi.org/project/scipy/

[9] NumPy Developers, “numpy.gradient,” NumPy v2.5 Manual, accessed Sep. 2026. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.gradient.html

[10] SciPy Developers, “scipy.differentiate.derivative,” SciPy v1.18 Manual, accessed Sep. 2026. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.differentiate.derivative.html

Chapter 10

Numerical Integration: Trapezoid, Simpson, Romberg and Gauss

Adding up many small pieces is one of the oldest things people do with numbers: the area of a field from strips, the rain that fell in a storm from readings every ten minutes, the electricity a machine used from its power every minute. Each of these is an integral, and when all we have are measurements, or a formula that has no antiderivative, the integral must be computed numerically. By the end of this chapter you will be able to derive the trapezoidal rule and Simpson's rules from interpolating polynomials and prove their error terms; estimate the error of an answer from the answer itself by halving the step; compute the area of an irregular piece of land from tape-measure offsets in Java and in Python; verify the order of each rule experimentally; build Romberg's table from the Euler–Maclaurin formula; derive Gauss–Legendre quadrature and explain why it needs only half as many points; let an adaptive Simpson routine put its effort where the function is difficult; and check every result against SciPy.

10.1 How Big Is the Orchard?

Pak Harjono's family owns a fruit and durian orchard on the edge of a village in the hills. It is a long strip of land between a straight village road and a small river, 160 meters along the road. The road side is easy: it is straight, and the family has planted a row of young rambutan trees along it. The river side is not: the bank wanders, bulging out toward the hills around a bend and coming back closer to the road toward the far end.

The family needs the area for two reasons. The first is practical. Durian trees are planted about ten meters apart in each direction, so each tree needs about a hundred square meters, and Anto, the son who is taking over the orchard, wants to know how many seedlings to order for the part of the land that is still empty and how much compost to bring for the whole orchard. The second is paperwork: the old land letter gives an area that nobody in the family quite believes, and before paying a surveyor to measure it officially, Anto wants to know roughly what the answer should be.

He has a 50-meter tape and a Saturday afternoon. With his cousin he walks along the road and, every ten meters, stretches the tape at right angles to the road to the river bank and writes down the distance, the *offset*. At 0 m it is 38.6 m, at 10 m it is 40.0 m, and so on to 31.9 m at the far corner, 160 m along: seventeen numbers in a notebook. How big is the orchard?

This is the classic surveyor's problem, and it is exactly the problem of numerical integration: the area under a curve $y(x)$ from $x = 0$ to $x = 160$ is the integral $\int_0^{160} y\,(x)\,dx$, and we know the curve only at seventeen points. The same mathematics turns rainfall intensity readings into millimeters of rain, a phone's power readings into energy, and a car's speed readings into distance. In this chapter we compute the orchard's area several ways, learn how accurate each way is and why, and then leave the orchard for the question every integration method must answer: how to get the most accuracy from the fewest function values.

10.2 The Idea in Plain Words

Imagine the orchard cut into sixteen strips, each ten meters wide, by lines from the road to the river at every measured point. Each strip has two straight sides (the offsets), a straight bottom (the road) and a top that is a short piece of river bank. If we pretend that the top is a straight line, each strip is a *trapezoid*, a four-sided figure with two parallel sides, and its area is the width times the average of the two parallel sides. For the first strip:

$$A_1 = 10 \times \frac{38.6 + 40.0}{2} = 10 \times 39.3 = 393.0\ \text{m}^2.$$

The second strip gives $10 \times (40.0 + 41.2)/2 = 406.0\ \text{m}^2$, and so on. Adding the sixteen strips gives the area. When we add them, every inner offset appears in two strips, once as the right side of one and once as the left side of the next, with half its value each time, while the two end offsets appear only once, with half their value. So the whole sum is

$$A \approx 10 \times \left(\frac{38.6}{2} + 40.0 + 41.2 + \cdots + 35.0 + \frac{31.9}{2} \right).$$

That is the **trapezoidal rule**, and **Figure 10.1** shows the strips.

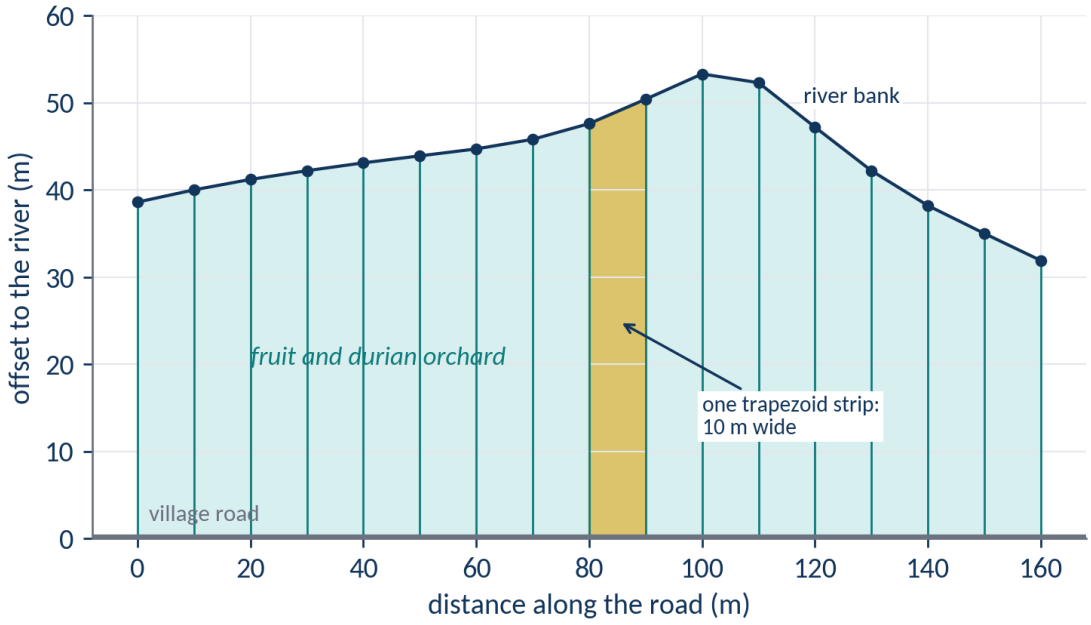


Figure 10.1 The orchard between the straight village road and the river bank, with the seventeen offsets measured every 10 m; the highlighted strip is one of the sixteen trapezoids of the trapezoidal rule.

Where the bank is straight, the trapezoids fit it well. Where it bends, as around 100 m, a straight top cuts off part of the curve or adds a sliver that is not there, and that is the error of the rule. A better idea is to draw a *curved* top: take the strips in pairs, draw the parabola through the three offsets of each pair, and use the area under the parabola. This is **Simpson's rule**. For the first pair of strips, from 0 to 20 m, it gives

$$A_{1,2} \approx \frac{10}{3}(38.6 + 4 \times 40.0 + 41.2) = \frac{10}{3} \times 239.8 = 799.3\ \text{m}^2,$$

where the trapezoids gave $393.0 + 406.0 = 799.0$ m^2. The difference, 0.3 m^2, is small because the bank near the road's start is almost straight. Around the bend the two rules will disagree much more, and that disagreement, as we shall see, is itself the best tool for estimating the error.

Why the weights 1,4,1 and the factor $\frac{1}{3}$? Section 10.4 derives them. For now notice that they give the middle offset four times the weight of the ends, and that the weights add up to 6, so that on flat land with all offsets equal to y, the formula gives $\frac{10}{3} \times 6y = 20y$, the area of a rectangle 20 m long, as it must.

> **KEY INSIGHT:** Every rule in this chapter does the same two things: it replaces the unknown curve by a simple one that passes through the known points (a line, a parabola, a polynomial), and it integrates the simple curve exactly. The rules differ only in which simple curve they use and where the known points are. Their errors are the areas between the true curve and the simple one.

10.3 The Trapezoidal Rule

10.3.1 Derivation

Let f be a function on an interval $[a, b]$, and let $h = b - a$. The straight line through $(a, f(a))$ and $(b, f(b))$, the linear interpolant of Chapter 7, is $p(x) = f(a) + \frac{f(b)-f(a)}{h}(x - a)$. Its integral is

$$\int_a^b p\,(x)\,dx = hf(a) + \frac{f(b) - f(a)}{h} \cdot \frac{h^2}{2} = \frac{h}{2}(f(a) + f(b)),$$

the area of the trapezoid. To find the error, expand f around the midpoint $m = (a + b)/2$ with Taylor's theorem (Chapter 9): $f(x) = f(m) + f'(m)(x - m) + \frac{1}{2}f''(m)(x - m)^2 + \cdots$. Integrating over $[a, b]$, the odd powers of $(x - m)$ vanish by symmetry, and $\int_a^b (x - m)^2\,dx = h^3/12$, so

$$\int_a^b f\,(x)\,dx = hf(m) + \frac{h^3}{24}f''(m) + O(h^5).$$

At the ends, $f(a) + f(b) = 2f(m) + \frac{h^2}{4}f''(m) + O(h^4)$, so the trapezoid is $\frac{h}{2}(f(a) + f(b)) = hf(m) + \frac{h^3}{8}f''(m) + O(h^5)$. Subtracting,

$$\int_a^b f\,(x)\,dx - \frac{h}{2}(f(a) + f(b)) = \left(\frac{1}{24} - \frac{1}{8}\right)h^3 f''(m) + \cdots = -\frac{h^3}{12}f''(\xi) \qquad (10.1)$$

for some ξ in $[a, b]$ [1], [2]. The error is zero when $f'' = 0$: the rule is exact for straight lines, as it should be. When f curves upward ($f'' > 0$), the chord lies above the curve and the trapezoid is too large; the minus sign says the same.

10.3.2 The composite rule

One trapezoid over a long interval is crude. Divide $[a, b]$ into n equal strips of width $h = (b - a)/n$, with points $x_i = a + ih$ and values $f_i = f(x_i)$, and add the trapezoids:

$$T(h) = h\left(\frac{f_0}{2} + f_1 + f_2 + \cdots + f_{n-1} + \frac{f_n}{2}\right), \qquad I - T(h) = -\frac{(b-a)h^2}{12} f''(\xi). \tag{10.2}$$

The error follows by adding the n errors of Equation (10.1): n terms of size $\frac{h^3}{12}f''$, and $nh = b - a$, so the total is $\frac{(b-a)h^2}{12}$ times an average value of f'', which is $f''(\xi)$ for some ξ by the intermediate value theorem. The composite trapezoidal rule is therefore **second order**: halving h divides the error by four.

The error formula is also a practical tool, because it gives a *guaranteed* bound when we can bound f''. For $\int_0^1 \frac{dx}{1+x}$, whose exact value is ln2, the second derivative $f''(x) = 2/(1+x)^3$ is largest at $x = 0$, where it equals 2. With $n = 4$ strips, $h = 0.25$, Equation (10.2) promises an error of at most $\frac{1 \times 0.25^2}{12} \times 2 = 0.0104$. Section 10.6 will show the actual error, 0.0039: inside the bound, as it must be, and smaller, because f'' is 2 only at one end and much smaller elsewhere. A bound tells you when you can stop worrying; the estimate of Section 10.5 tells you how large the error probably is.

The same formula also answers the planning question *how many strips do I need?* before any computing is done. For an error of at most 10^{-6} in ln2 we need $\frac{h^2}{12} \times 2 \le 10^{-6}$, that is $h \le 0.00245$, or $n \ge 409$ strips. Simpson's error formula Equation (10.4) of the next section, with $f^{(4)}(x) = 24/(1+x)^5$ at most 24, needs only $\frac{h^4}{180} \times 24 \le 10^{-6}$, that is $h \le 0.0523$, or $n \ge 20$ strips. Twenty function values instead of four hundred: that comparison, made before a single value is computed, is the whole case for higher-order rules.

To see the rule at work by hand, take $n = 4$ strips for ln2, so $h = 0.25$. The values of $f(x) = 1/(1+x)$ at $x = 0, 0.25, 0.5, 0.75, 1$ are 1, 0.8, 0.666667, 0.571429 and 0.5. By Equation (10.2), $T = 0.25 \times (0.5 + 0.8 + 0.666667 + 0.571429 + 0.25) = 0.25 \times 2.788095 = 0.697024$, against the exact 0.693147: an error of 0.0039, inside the bound of 0.0104 computed above. Simpson's weights on the same five values give $\frac{0.25}{3}(1 + 4 \times 0.8 + 2 \times 0.666667 + 4 \times 0.571429 + 0.5) = \frac{0.25}{3} \times 8.319048 = 0.693254$, an error of 0.0001, forty times smaller from the same arithmetic.

There is more to say about the error, and it is the key to Romberg's method. The **Euler–Maclaurin formula** gives the whole error as a series in even powers of h:

$$T(h) - I = \frac{h^2}{12}(f'(b) - f'(a)) - \frac{h^4}{720}(f'''(b) - f'''(a)) + \cdots \tag{10.3}$$

The coefficients depend only on derivatives at the two ends, not on h [2], [3]. Two consequences are worth remembering. First, since the error is $c_1 h^2 + c_2 h^4 + \cdots$ with fixed c_i, Richardson extrapolation (Section 9.5) applies to it directly; that is Section 10.7. Second, for a *periodic* function integrated over a whole period, $f'(b) = f'(a)$, $f'''(b) = f'''(a)$, and so on, every term vanishes, and the humble trapezoidal rule becomes spectacularly accurate: this is why it is the rule of choice for periodic signals.

10.4 Simpson's Rules

10.4.1 Simpson's 1/3 rule

Take two strips, from $x_0 = m - h$ to $x_2 = m + h$ with midpoint $x_1 = m$, and replace f by the parabola through the three points. Instead of writing the parabola out, use the method of undetermined coefficients from Chapter 9: look for weights with

$$\int_{m-h}^{m+h} f\,(x)\,dx \approx h(w_0 f(m-h) + w_1 f(m) + w_2 f(m+h))$$

that are exact for $f = 1$, $f = x - m$ and $f = (x-m)^2$. The three integrals are $2h$, 0 and $\frac{2}{3}h^3$, so

$$w_0 + w_1 + w_2 = 2, \qquad -w_0 + w_2 = 0, \qquad w_0 + w_2 = \frac{2}{3}.$$

Hence $w_0 = w_2 = \frac{1}{3}$ and $w_1 = \frac{4}{3}$: the weights $\frac{h}{3}(1,4,1)$. Now a small surprise: the rule is also exact for $(x-m)^3$, because both the integral and the rule are zero for an odd function around m. A rule built to integrate parabolas exactly integrates cubics exactly too, for free, just as the central difference of Chapter 9 gained an order from symmetry.

For the error, repeat the midpoint expansion of Section 10.3.1 to the fourth power. The integral is $2hf(m) + \frac{h^3}{3}f''(m) + \frac{h^5}{60}f^{(4)}(m) + \cdots$ and the rule is $\frac{h}{3}\left(6f(m) + h^2 f''(m) + \frac{h^4}{12}f^{(4)}(m) + \cdots\right)$. The terms in f and f'' agree; the difference is $\left(\frac{1}{60} - \frac{1}{36}\right)h^5 f^{(4)} = -\frac{h^5}{90}f^{(4)}$. Adding $n/2$ such panels over $[a,b]$ with n even gives the **composite Simpson's rule**:

$$S(h) = \frac{h}{3}(f_0 + 4f_1 + 2f_2 + 4f_3 + \cdots + 2f_{n-2} + 4f_{n-1} + f_n), \qquad I - S(h) = -\frac{(b-a)h^4}{180}f^{(4)}(\xi). \tag{10.4}$$

The pattern of weights inside is 4,2,4,2, … ,4: points in the middle of a panel get 4, points shared by two panels get $1 + 1 = 2$. Simpson's rule is **fourth order**: halving h divides the error by sixteen, for the same kind of work as the trapezoidal rule. It needs an even number of strips. Anto's seventeen offsets make sixteen strips, so it applies.

There is a neat connection between the two rules. Take the trapezoidal rule with step h and with step $2h$ (every other offset). A little algebra shows

$$S(h) = \frac{4T(h) - T(2h)}{3}, \tag{10.5}$$

which is one step of Richardson extrapolation applied to Equation (10.3): the h^2 term cancels, and what is left is Simpson's rule. Exercise 10.3 asks you to check it.

10.4.2 Simpson's 3/8 rule

If the number of strips is odd, say the neighbor's fence stops the tape at 150 m, leaving fifteen strips, Simpson's 1/3 rule cannot cover them all. The cubic through four points spanning three strips gives **Simpson's 3/8 rule**,

$$\int_{x_0}^{x_3} f(x)\,dx \approx \frac{3h}{8}\left(f_0 + 3f_1 + 3f_2 + f_3\right), \qquad \text{error} = -\frac{3h^5}{80} f^{(4)}(\xi), \tag{10.6}$$

also exact for cubics, with a slightly larger error constant than two panels of the 1/3 rule [1]. The usual recipe for an odd number of strips is the 1/3 rule on all but the last three and the 3/8 rule on those; Exercise 10.2 does exactly that for fifteen strips.

10.5 Building the Orchard Calculator

Anto's notebook becomes the file `data/orchard.csv`: a header line `x_m,offset_m` and seventeen lines such as `0,38.6` and `160,31.9`. Like the trip log of Chapter 9, the file in the code folder was prepared for this chapter from a known river curve, so we know the true area, 7026.8 m^2, and can judge every estimate against it; Anto, of course, cannot, and the program must tell him how far to trust its answer without knowing the truth.

It does that by computing each rule twice: once with all seventeen offsets (step 10 m) and once with every other offset (nine offsets, step 20 m). The difference between the two answers estimates the error. For the trapezoidal rule, if $T(h) = I + ch^2$ and $T(2h) = I + 4ch^2$, then $T(h) - T(2h) = -3ch^2$, so the error of the finer answer is

$$I - T(h) \approx \frac{T(h) - T(2h)}{3}, \qquad I - S(h) \approx \frac{S(h) - S(2h)}{15}, \tag{10.7}$$

the second by the same argument with h^4 and $16 - 1 = 15$. The program also adds the estimated error to Simpson's answer, one more Richardson step, and reports the result in square meters, in hectares, and as a number of durian trees at ten meters by ten. **Listing 10.1** is the whole program.

Listing 10.1 `orchard.py`: the orchard's area by the trapezoidal and Simpson rules, with error estimates from halving the data.

```
"""Area of the orchard between the road and the river from offsets measured
every 10 m. Usage: python orchard.py [csv file]  (default ../data/orchard.csv)"""
import sys


def load(path):
    xs, ys = [], []
    with open(path) as f:
        next(f)                                  # skip the header line
        for line in f:
            x, y = line.strip().split(",")
            xs.append(float(x))
            ys.append(float(y))
    return xs, ys


def trapezoid(y, h):
    """Composite trapezoidal rule for equally spaced values y."""
    return h * (y[0] / 2 + sum(y[1:-1]) + y[-1] / 2)


def simpson(y, h):
    """Composite Simpson 1/3 rule; needs an even number of intervals."""
    n = len(y) - 1
    if n % 2:
        raise ValueError("Simpson's 1/3 rule needs an even number of intervals")
    odd = sum(y[1:n:2])                          # y1, y3, ..., y(n-1)
    even = sum(y[2:n:2])                         # y2, y4, ..., y(n-2)
    return h / 3 * (y[0] + 4 * odd + 2 * even + y[n])


def main(args):
    path = args[0] if args else "../data/orchard.csv"
    x, y = load(path)
    h = x[1] - x[0]
    print(f"{len(y)} offsets, every {h:.0f} m along {x[-1] - x[0]:.0f} m of road")
    t1, t2 = trapezoid(y, h), trapezoid(y[::2], 2 * h)
    s1, s2 = simpson(y, h), simpson(y[::2], 2 * h)
    print("rule        every 10 m  every 20 m  error estimate")
    print(f"trapezoid   {t1:10.1f}  {t2:10.1f}  {(t1 - t2) / 3:14.1f}")
    print(f"Simpson     {s1:10.1f}  {s2:10.1f}  {(s1 - s2) / 15:14.1f}")
    best = s1 + (s1 - s2) / 15                   # Richardson on Simpson
    print(f"area {best:.0f} m2 = {best / 10000:.2f} ha,"
          f" room for about {best / 100:.0f} durian trees at 10 m x 10 m")
```

Listing 10.1 (Continued) *orchard.py: the orchard's area by the trapezoidal and Simpson rules, with error estimates from halving the data.*

```
if __name__ == "__main__":
    main(sys.argv[1:])
```

The slice `y[::2]` takes every other offset, and `y[1:n:2]` and `y[2:n:2]` pick the odd and even inner points for Simpson's weights 4 and 2. The function raises an error instead of quietly returning a wrong answer when the number of strips is odd. **Listing 10.2** is the Java version, with a small helper `everyOther` for the slice.

Listing 10.2 `Orchard.java`: the same calculator in Java.

```java
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;
import java.util.Locale;

/** Area of the orchard between the road and the river from offsets measured
 *  every 10 m. Usage: java Orchard [csv file]  (default ../data/orchard.csv) */
public class Orchard {
    /** Returns {positions, offsets}. */
    static double[][] load(String path) throws IOException {
        List<String> lines = Files.readAllLines(Path.of(path));
        int n = lines.size() - 1;                       // the first line is a header
        double[] x = new double[n], y = new double[n];
        for (int i = 0; i < n; i++) {
            String[] parts = lines.get(i + 1).trim().split(",");
            x[i] = Double.parseDouble(parts[0]);
            y[i] = Double.parseDouble(parts[1]);
        }
        return new double[][] {x, y};
    }

    /** Composite trapezoidal rule for equally spaced values y. */
    static double trapezoid(double[] y, double h) {
        double inner = 0;
        for (int i = 1; i < y.length - 1; i++) {
            inner += y[i];
        }
        return h * (y[0] / 2 + inner + y[y.length - 1] / 2);
    }

    /** Composite Simpson 1/3 rule; needs an even number of intervals. */
    static double simpson(double[] y, double h) {
        int n = y.length - 1;
        if (n % 2 != 0) {
            throw new IllegalArgumentException(
                "Simpson's 1/3 rule needs an even number of intervals");
        }
        double odd = 0, even = 0;
        for (int i = 1; i < n; i += 2) {
            odd += y[i];                                // y1, y3, ..., y(n-1)
        }
        for (int i = 2; i < n; i += 2) {
            even += y[i];                               // y2, y4, ..., y(n-2)
        }
        return h / 3 * (y[0] + 4 * odd + 2 * even + y[n]);
```

Listing 10.2 (Continued) *Orchard.java: the same calculator in Java.*

```
    }

    /** Every second value: y0, y2, y4, ... */
    static double[] everyOther(double[] y) {
        double[] r = new double[(y.length + 1) / 2];
        for (int i = 0; i < r.length; i++) {
            r[i] = y[2 * i];
        }
        return r;
    }

    public static void main(String[] args) throws IOException {
        double[][] data = load(args.length > 0 ? args[0] : "../data/orchard.csv");
        double[] x = data[0], y = data[1];
        double h = x[1] - x[0];
        System.out.println(String.format(Locale.ROOT,
            "%d offsets, every %.0f m along %.0f m of road",
            y.length, h, x[x.length - 1] - x[0]));
        double t1 = trapezoid(y, h), t2 = trapezoid(everyOther(y), 2 * h);
        double s1 = simpson(y, h), s2 = simpson(everyOther(y), 2 * h);
        System.out.println("rule        every 10 m  every 20 m  error estimate");
        System.out.println(String.format(Locale.ROOT,
            "trapezoid   %10.1f  %10.1f  %14.1f", t1, t2, (t1 - t2) / 3));
        System.out.println(String.format(Locale.ROOT,
            "Simpson     %10.1f  %10.1f  %14.1f", s1, s2, (s1 - s2) / 15));
        double best = s1 + (s1 - s2) / 15;            // Richardson on Simpson
        System.out.println(String.format(Locale.ROOT,
            "area %.0f m2 = %.2f ha, room for about %.0f durian trees"
            + " at 10 m x 10 m", best, best / 10000, best / 100));
    }
}
```

Both print:

```
17 offsets, every 10 m along 160 m of road
rule        every 10 m  every 20 m  error estimate
trapezoid       7023.5      7011.0             4.2
Simpson         7027.7      7039.3            -0.8
area 7027 m2 = 0.70 ha, room for about 70 durian trees at 10 m x 10 m
```

Read it with the true area, 7026.8 m^2, in mind. The trapezoidal rule with all offsets is 3.3 m^2 short, and its error estimate says “add about 4.2”: a good warning, of the right sign and size. Simpson’s rule is 0.9 m^2 over, and its estimate says “subtract about 0.8”. The corrected answer, 7026.9 m^2, is within a tenth of a square meter of the truth, far closer than the tape measure’s own accuracy deserves. Notice also that $T(h) + \frac{T(h)-T(2h)}{3} = 7023.5 + 4.2 = 7027.7$ is exactly Simpson’s answer, as Equation (10.5) promised.

Where does the remaining error come from? Look at the pairs of strips around the bend. From 100 to 120 m the offsets are 53.3, 52.3 and 47.2 m. The trapezoids give $10 \times (26.65 + 52.3 + 23.6) = 1025.5$ m^2, Simpson gives $\frac{10}{3}(53.3 + 4 \times 52.3 + 47.2) = 1032.3$ m^2, and the true area of that piece is 1031.3 m^2. The bank bulges outwards there, so the straight tops of the trapezoids cut off almost six square meters, and the parabola follows the bulge to within one. Along the straight first stretch, from 0 to 20 m, the two rules differ by only 0.3 m^2, as we saw in Section 10.2.

But there is a second source of error that no rule can remove: the offsets themselves. Each was written down to the nearest 0.1 m, so each carries an error of up to 0.05 m. An error of 0.05 m in one offset changes the area by up to $0.05 \times 10 = 0.5$ m^2 (a little more with Simpson’s weight 4); spread over seventeen offsets with errors of random sign, the total is typically around one square

meter. On the first piece, for example, the true area is 798.8 m², while both rules, fed the rounded offsets, give about 799. Simpson's truncation error on the whole orchard, about one square meter, is thus of the same size as the measurement error. Going to Romberg or Gauss on these data would polish the arithmetic below the level of the tape; the next improvement must come from better measurements, not better formulas. This balance between method error and data error is the integration twin of the optimal step of Chapter 9.

For Anto the conclusion is simple: the orchard is 0.70 hectare, give or take a few square meters, with room for about seventy durian trees at the usual spacing. If the old land letter says 0.6 hectare, it is worth asking the surveyor to look carefully; if it says 0.7, the family can relax. Either way the official measurement is the surveyor's to make; the calculation is for planning.

10.6 Checking the Order

The orders claimed in Equation (10.2) and Equation (10.4) are worth seeing with our own eyes, on an integral whose exact value we know. Take

$$\int_0^1 \frac{dx}{1+x} = \ln 2 = 0.693147\ldots$$

and compute both rules for $n = 2, 4, 8, \ldots, 256$ strips. If a rule has order p, doubling n divides the error by 2^p: by 4 for the trapezoidal rule and by 16 for Simpson's. **Listing 10.3** prints each error and the ratio to the previous one. The exact value is written as the double-precision constant `0.6931471805599453`, so that the Java and Python programs compare against the same number.

Listing 10.3 `quadrature.py`: composite trapezoid and Simpson rules for a known integral, with the ratios of successive errors.

```
"""Composite trapezoid and Simpson rules on the integral of 1/(1+x) from
0 to 1, which equals ln 2. Doubling n shows the order of each rule."""

LN2 = 0.6931471805599453                          # ln 2 to double precision


def f(x):
    return 1.0 / (1.0 + x)


def trapezoid(f, a, b, n):
    h = (b - a) / n
    inner = sum(f(a + i * h) for i in range(1, n))
    return h * (f(a) / 2 + inner + f(b) / 2)


def simpson(f, a, b, n):
    h = (b - a) / n                               # n must be even
    odd = sum(f(a + i * h) for i in range(1, n, 2))
    even = sum(f(a + i * h) for i in range(2, n, 2))
    return h / 3 * (f(a) + 4 * odd + 2 * even + f(b))


def main():
    print("   n   trapezoid error  ratio    Simpson error  ratio")
    old_t = old_s = None
    n = 2
    while n <= 256:
        et = abs(trapezoid(f, 0.0, 1.0, n) - LN2)
        es = abs(simpson(f, 0.0, 1.0, n) - LN2)
        rt = f"{old_t / et:6.1f}" if old_t else "      "
        rs = f"{old_s / es:6.1f}" if old_s else "      "
        print(f"{n:4d}  {et:15.3e}  {rt}  {es:15.3e}  {rs}")
        old_t, old_s = et, es
        n *= 2


if __name__ == "__main__":
    main()
```

Listing 10.4 `Quadrature.java`: the same check in Java.

```java
import java.util.Locale;
import java.util.function.DoubleUnaryOperator;

/** Composite trapezoid and Simpson rules on the integral of 1/(1+x) from
 *  0 to 1, which equals ln 2. Doubling n shows the order of each rule. */
public class Quadrature {
    static final double LN2 = 0.6931471805599453;    // ln 2 to double precision

    static double f(double x) {
        return 1.0 / (1.0 + x);
    }

    static double trapezoid(DoubleUnaryOperator f, double a, double b, int n) {
        double h = (b - a) / n, inner = 0;
        for (int i = 1; i < n; i++) {
            inner += f.applyAsDouble(a + i * h);
        }
        return h * (f.applyAsDouble(a) / 2 + inner + f.applyAsDouble(b) / 2);
    }

    static double simpson(DoubleUnaryOperator f, double a, double b, int n) {
        double h = (b - a) / n, odd = 0, even = 0;   // n must be even
        for (int i = 1; i < n; i += 2) {
            odd += f.applyAsDouble(a + i * h);
        }
        for (int i = 2; i < n; i += 2) {
            even += f.applyAsDouble(a + i * h);
        }
        return h / 3 * (f.applyAsDouble(a) + 4 * odd + 2 * even
                        + f.applyAsDouble(b));
    }

    public static void main(String[] args) {
        System.out.println("   n   trapezoid error  ratio    Simpson error  ratio");
        double oldT = 0, oldS = 0;
        for (int n = 2; n <= 256; n *= 2) {
            double et = Math.abs(trapezoid(Quadrature::f, 0.0, 1.0, n) - LN2);
            double es = Math.abs(simpson(Quadrature::f, 0.0, 1.0, n) - LN2);
            String rt = oldT > 0 ? String.format(Locale.ROOT, "%6.1f", oldT / et)
                                 : "      ";
            String rs = oldS > 0 ? String.format(Locale.ROOT, "%6.1f", oldS / es)
                                 : "      ";
            System.out.println(String.format(Locale.ROOT,
                "%4d  %15.3e  %s  %15.3e  %s", n, et, rt, es, rs));
            oldT = et;
            oldS = es;
```

Listing 10.4 (Continued) *Quadrature.java: the same check in Java.*

```java
        }
    }
}
```

Listing 10.4 gives the same table:

```
   n   trapezoid error  ratio    Simpson error  ratio
   2       1.519e-02                1.297e-03
   4       3.877e-03     3.9        1.068e-04    12.1
   8       9.747e-04     4.0        7.350e-06    14.5
  16       2.440e-04     4.0        4.723e-07    15.6
  32       6.103e-05     4.0        2.973e-08    15.9
  64       1.526e-05     4.0        1.862e-09    16.0
 128       3.815e-06     4.0        1.164e-10    16.0
 256       9.537e-07     4.0        7.275e-12    16.0
```

The trapezoid ratios are 4.0 from the start; Simpson's climb to 16.0 once h is small enough for the leading error term to dominate. The numbers also match Equation (10.3): with $f'(x) = -1/(1+x)^2$, $f'(1) - f'(0) = 0.75$, and for $n = 2$ ($h = 0.5$) the first term is $\frac{0.25}{12} \times 0.75 = 0.0156$, against the actual 0.0152. For the work involved, Simpson is overwhelmingly better: 257 function values give the trapezoidal rule an error of 10^{-6} and Simpson's rule 7×10^{-12}.

The ratio test is also the best debugging tool for a quadrature program you write yourself. A trapezoidal rule whose ratios settle at 2 instead of 4 has almost certainly lost the halves at the ends; a Simpson rule whose ratios settle at 2 has its weights 4 and 2 swapped. Wrong weights often still give answers that look plausible to three digits, but they never give the right ratios.

10.7 Romberg Integration

The error series Equation (10.3) contains only even powers of h, with coefficients that do not depend on h. That is precisely the situation of Section 9.5, and Richardson's table applies without change. Let $R_{i,0} = T(h/2^i)$ be the trapezoidal rule with 2^i strips. Then

$$R_{i,j} = \frac{4^j R_{i,j-1} - R_{i-1,j-1}}{4^j - 1}, \qquad j = 1, \ldots, i, \tag{10.8}$$

has error $O(h^{2j+2})$. The first extrapolated column is Simpson's rule, by Equation (10.5); the second turns out to be Boole's rule, of order six; and further columns have no names but keep gaining two orders each. This scheme was proposed by Werner Romberg in 1955 [4] and is called **Romberg integration**.

One more trick makes it cheap. When the number of strips doubles, all the old points are reused, and only the new midpoints must be evaluated:

$$T\left(\frac{h}{2}\right) = \frac{1}{2}T(h) + \frac{h}{2}\sum_{k \text{ odd}} f\left(a + k\frac{h}{2}\right).$$

So the whole table with five rows, 1,2,4,8,16 strips, needs only the 17 function values of the finest row. **Listing 10.5** builds it for ln2.

Listing 10.5 `romberg.py`: the Romberg table, adding only the new midpoints on each row.

```
"""Romberg integration: trapezoid values with h, h/2, h/4, ... improved by
Richardson extrapolation. Test: the integral of 1/(1+x) from 0 to 1 is ln 2."""
from quadrature import f, LN2


def romberg(f, a, b, levels):
    r = [[(b - a) / 2 * (f(a) + f(b))]]           # R[0][0]: one trapezoid
    evals = 2
    for i in range(1, levels):
        n = 2 ** i                                # intervals on this row
        h = (b - a) / n
        mids = sum(f(a + k * h) for k in range(1, n, 2))   # only new points
        evals += n // 2
        row = [r[i - 1][0] / 2 + h * mids]        # trapezoid with n intervals
        for j in range(1, i + 1):
            factor = 4 ** j
            row.append((factor * row[j - 1] - r[i - 1][j - 1]) / (factor - 1))
        r.append(row)
    return r, evals


def main():
    table, evals = romberg(f, 0.0, 1.0, 5)
    for i, row in enumerate(table):
        print(f"n = {2 ** i:2d}  " + "  ".join(f"{v:.12f}" for v in row))
    best = table[-1][-1]
    print(f"R[4][4] = {best:.15f}, error {abs(best - LN2):.1e},"
          f" {evals} function values")


if __name__ == "__main__":
    main()
```

Listing 10.6 `Romberg.java`: the same table in Java.

```
import java.util.Locale;
import java.util.function.DoubleUnaryOperator;

/** Romberg integration: trapezoid values with h, h/2, h/4, ... improved by
 *  Richardson extrapolation. Test: the integral of 1/(1+x) from 0 to 1 is ln 2. */
public class Romberg {
    static int evals;

    static double[][] romberg(DoubleUnaryOperator f, double a, double b,
                              int levels) {
        double[][] r = new double[levels][];
        r[0] = new double[] {(b - a) / 2
                             * (f.applyAsDouble(a) + f.applyAsDouble(b))};
        evals = 2;
        for (int i = 1; i < levels; i++) {
            int n = 1 << i;                          // intervals on this row
            double h = (b - a) / n, mids = 0;
            for (int k = 1; k < n; k += 2) {         // only the new points
                mids += f.applyAsDouble(a + k * h);
            }
            evals += n / 2;
            r[i] = new double[i + 1];
            r[i][0] = r[i - 1][0] / 2 + h * mids;    // trapezoid, n intervals
            for (int j = 1; j <= i; j++) {
                double factor = Math.pow(4, j);
                r[i][j] = (factor * r[i][j - 1] - r[i - 1][j - 1]) / (factor - 1);
            }
        }
        return r;
    }

    public static void main(String[] args) {
        double[][] table = romberg(Quadrature::f, 0.0, 1.0, 5);
        for (int i = 0; i < table.length; i++) {
            StringBuilder line = new StringBuilder(
                String.format(Locale.ROOT, "n = %2d", 1 << i));
            for (double v : table[i]) {
                line.append(String.format(Locale.ROOT, "  %.12f", v));
            }
            System.out.println(line);
        }
        double best = table[4][4];
        System.out.println(String.format(Locale.ROOT,
            "R[4][4] = %.15f, error %.1e, %d function values",
            best, Math.abs(best - Quadrature.LN2), evals));
    }
```

Listing 10.6 (Continued) *Romberg.java: the same table in Java.*

```
}
```

Listing 10.6 prints the same numbers:

```
n =  1  0.750000000000
n =  2  0.708333333333  0.694444444444
n =  4  0.697023809524  0.693253968254  0.693174603175
n =  8  0.694121850372  0.693154530655  0.693147901481  0.693147477645
n = 16  0.693391202208  0.693147652819  0.693147194297  0.693147183072  0.693147181917
R[4][4] = 0.693147181916745, error 1.4e-09, 17 function values
```

The first column creeps toward 0.693147: the trapezoidal rule with 16 strips is still wrong in the fourth decimal. The diagonal races there: $R_{4,4}$ is correct to eight decimals with the same 17 function values that give Simpson's rule an error of 4.7×10^{-7} in the table of Section 10.6. Romberg's method is the natural choice when you can evaluate f wherever you like, f is smooth, and you want many digits cheaply.

Can Anto use it? Almost. His seventeen offsets are exactly the $2^4 + 1$ points Romberg needs for four halvings, and SciPy's `romb` function (Section 10.10) accepts such data. But extrapolation assumes the error series Equation (10.3), which needs a smooth curve, and a tape-measured offset is uncertain by some centimeters; the higher columns amplify those small errors as Richardson always does. For measured data one or two extrapolation steps, as in Section 10.5, are the sensible limit.

10.8 Gauss Quadrature: Choosing Where to Measure

10.8.1 The idea

All the rules so far use equally spaced points and choose only the weights. Suppose Anto were free to choose *where* to measure the offsets, not every ten meters but at cleverly chosen places. With n offsets he would then have $2n$ numbers to choose: n positions and n weights. A polynomial of degree $2n - 1$ has $2n$ coefficients, so one can hope for a rule that is exact for every polynomial up to degree $2n - 1$, nearly twice what equally spaced points allow. Carl Friedrich Gauss showed in 1814 that this hope is right, and the resulting rules are called **Gauss quadrature** [2], [3].

On the standard interval $[-1,1]$, look for two points $\pm t$ and equal weights w (symmetry makes them equal) with $\int_{-1}^{1} f(x)\,dx \approx wf(-t) + wf(t)$. Odd powers are integrated exactly by symmetry. For $f = 1$ we need $2w = 2$, so $w = 1$; for $f = x^2$ we need $2t^2 = \frac{2}{3}$, so $t = 1/\sqrt{3} \approx 0.57735$. The **two-point Gauss–Legendre rule**

$$\int_{-1}^{1} f(x)\,dx \approx f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right) \tag{10.9}$$

is exact for $1, x, x^2$ and x^3: two function values integrate every cubic exactly, which Simpson's rule needs three for.

10.8.2 Legendre polynomials and the general rule

For n points the conditions become a nonlinear system that is unpleasant to solve directly, but the answer is beautiful. The best points are the n roots of the **Legendre polynomial** $P_n(x)$, the polynomials defined by $P_0 = 1$, $P_1 = x$ and

$$(k+1)P_{k+1}(x) = (2k+1)xP_k(x) - kP_{k-1}(x),$$

so that, for example, $P_2(x) = \frac{1}{2}(3x^2 - 1)$ with roots $\pm 1/\sqrt{3}$, exactly the points found above, and $P_3(x) = \frac{1}{2}(5x^3 - 3x)$ with roots 0 and $\pm\sqrt{3/5}$. The weights are $w_i = \frac{2}{(1-t_i^2)[P_n'(t_i)]^2}$. The reason, in brief, is that the Legendre polynomials are *orthogonal*: $\int_{-1}^{1} P_n(x)q(x)\,dx = 0$ for every polynomial q of degree less than n. Divide any polynomial f of degree $2n - 1$ by P_n: $f = qP_n + r$ with q and r of degree less than n. The integral of qP_n is zero by orthogonality, and the rule gives zero for it too, because P_n vanishes at the nodes. The remainder r has degree less than n and is integrated exactly by any interpolatory rule on n points. Hence f is integrated exactly [2]. The error of the n-point rule on $[a, b]$ is

$$I - G_n = \frac{(b-a)^{2n+1}(n!)^4}{(2n+1)[(2n)!]^3} f^{(2n)}(\xi),$$

which falls extremely fast with n for smooth functions. In practice the nodes and weights are computed as the eigenvalues and eigenvector components of a small tridiagonal matrix, by the Golub–Welsch algorithm [5]; eigenvalues are the subject of Chapter 12. For up to five points they have closed forms, which our program uses.

An integral over $[a, b]$ is mapped onto $[-1,1]$ with $x = \frac{a+b}{2} + \frac{b-a}{2}t$, so that $dx = \frac{b-a}{2}dt$:

$$\begin{aligned} \int_a^b f(x)\,dx &= \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{a+b}{2} + \frac{b-a}{2}t\right) dt \\ &\approx \frac{b-a}{2}\sum_{i=1}^{n} w_i\, f\left(\frac{a+b}{2} + \frac{b-a}{2}t_i\right). \end{aligned} \tag{10.10}$$

A hand computation for ln2 with two points: the nodes ± 0.57735 map to $x = 0.5 \pm 0.5 \times 0.57735$, that is 0.211325 and 0.788675; there $f = 1/(1+x)$ is 0.825542 and 0.559073; the sum times $\frac{1}{2}$ is 0.692308, already correct to three digits from two function values. (The exact result of the two-point rule here is $9/13$.)

10.8.3 The program

Listing 10.7 holds the nodes and weights for $n = 1, \ldots, 5$ in a dictionary, built from their closed forms with square roots so that nothing has to be copied from a table, applies Equation (10.10), and prints the error next to those of the trapezoidal and Simpson rules with the same number of function values.

Listing 10.7 `gauss.py`: Gauss–Legendre quadrature with one to five points.

```
"""Gauss-Legendre quadrature with 1 to 5 points on the integral of 1/(1+x)
from 0 to 1 (= ln 2), next to the trapezoid and Simpson rules that use the
same number of function values."""
import math
from quadrature import f, LN2, trapezoid, simpson

S = math.sqrt
NODES = {   # points and weights on [-1, 1]
    1: [(0.0, 2.0)],
    2: [(-1 / S(3), 1.0), (1 / S(3), 1.0)],
    3: [(-S(3 / 5), 5 / 9), (0.0, 8 / 9), (S(3 / 5), 5 / 9)],
    4: [(-S(3 / 7 + 2 / 7 * S(6 / 5)), (18 - S(30)) / 36),
        (-S(3 / 7 - 2 / 7 * S(6 / 5)), (18 + S(30)) / 36),
        (S(3 / 7 - 2 / 7 * S(6 / 5)), (18 + S(30)) / 36),
        (S(3 / 7 + 2 / 7 * S(6 / 5)), (18 - S(30)) / 36)],
    5: [(-S(5 + 2 * S(10 / 7)) / 3, (322 - 13 * S(70)) / 900),
        (-S(5 - 2 * S(10 / 7)) / 3, (322 + 13 * S(70)) / 900),
        (0.0, 128 / 225),
        (S(5 - 2 * S(10 / 7)) / 3, (322 + 13 * S(70)) / 900),
        (S(5 + 2 * S(10 / 7)) / 3, (322 - 13 * S(70)) / 900)],
}


def gauss(f, a, b, n):
    """Map [-1, 1] onto [a, b]: x = mid + half * t, dx = half * dt."""
    mid, half = (a + b) / 2, (b - a) / 2
    return half * sum(w * f(mid + half * t) for t, w in NODES[n])


def main():
    print("points  Gauss-Legendre   error    trapezoid error  Simpson error")
    for n in range(1, 6):
        g = gauss(f, 0.0, 1.0, n)
        line = f"{n:6d}  {g:.12f}  {abs(g - LN2):.1e}"
        if n > 1:                                  # n points = n - 1 intervals
            t = trapezoid(f, 0.0, 1.0, n - 1)
            line += f"  {abs(t - LN2):15.1e}"
        if n % 2 and n > 1:                        # Simpson needs n odd
            s = simpson(f, 0.0, 1.0, n - 1)
            line += f"  {abs(s - LN2):13.1e}"
        print(line)


if __name__ == "__main__":
    main()
```

Listing 10.8 `Gauss.java`: the same rules in Java, with the nodes in a three-dimensional array.

```
import java.util.Locale;
import java.util.function.DoubleUnaryOperator;

/** Gauss-Legendre quadrature with 1 to 5 points on the integral of 1/(1+x)
 *  from 0 to 1 (= ln 2), next to the trapezoid and Simpson rules that use the
 *  same number of function values. */
public class Gauss {
    static double s(double x) {
        return Math.sqrt(x);
    }

    /** NODES[n] holds {point, weight} pairs on [-1, 1]. */
    static final double[][][] NODES = {
        {},
        {{0.0, 2.0}},
        {{-1 / s(3), 1.0}, {1 / s(3), 1.0}},
        {{-s(3.0 / 5), 5.0 / 9}, {0.0, 8.0 / 9}, {s(3.0 / 5), 5.0 / 9}},
        {{-s(3.0 / 7 + 2.0 / 7 * s(6.0 / 5)), (18 - s(30)) / 36},
         {-s(3.0 / 7 - 2.0 / 7 * s(6.0 / 5)), (18 + s(30)) / 36},
         {s(3.0 / 7 - 2.0 / 7 * s(6.0 / 5)), (18 + s(30)) / 36},
         {s(3.0 / 7 + 2.0 / 7 * s(6.0 / 5)), (18 - s(30)) / 36}},
        {{-s(5 + 2 * s(10.0 / 7)) / 3, (322 - 13 * s(70)) / 900},
         {-s(5 - 2 * s(10.0 / 7)) / 3, (322 + 13 * s(70)) / 900},
         {0.0, 128.0 / 225},
         {s(5 - 2 * s(10.0 / 7)) / 3, (322 + 13 * s(70)) / 900},
         {s(5 + 2 * s(10.0 / 7)) / 3, (322 - 13 * s(70)) / 900}},
    };

    /** Map [-1, 1] onto [a, b]: x = mid + half * t, dx = half * dt. */
    static double gauss(DoubleUnaryOperator f, double a, double b, int n) {
        double mid = (a + b) / 2, half = (b - a) / 2, sum = 0;
        for (double[] tw : NODES[n]) {
            sum += tw[1] * f.applyAsDouble(mid + half * tw[0]);
        }
        return half * sum;
    }
```

Listing 10.8 (Continued) *Gauss.java: the same rules in Java, with the nodes in a three-dimensional array.*

```
    public static void main(String[] args) {
        double ln2 = Quadrature.LN2;
        System.out.println(
            "points  Gauss-Legendre   error    trapezoid error  Simpson error");
        for (int n = 1; n <= 5; n++) {
            double g = gauss(Quadrature::f, 0.0, 1.0, n);
            StringBuilder line = new StringBuilder(String.format(Locale.ROOT,
                "%6d  %.12f  %.1e", n, g, Math.abs(g - ln2)));
            if (n > 1) {                                  // n points = n - 1 intervals
                double t = Quadrature.trapezoid(Quadrature::f, 0.0, 1.0, n - 1);
                line.append(String.format(Locale.ROOT, "  %15.1e",
                                          Math.abs(t - ln2)));
            }
            if (n % 2 == 1 && n > 1) {                    // Simpson needs n odd
                double sv = Quadrature.simpson(Quadrature::f, 0.0, 1.0, n - 1);
                line.append(String.format(Locale.ROOT, "  %13.1e",
                                          Math.abs(sv - ln2)));
            }
            System.out.println(line);
        }
    }
}
```

Listing 10.8 prints the same table:

```
points  Gauss-Legendre   error    trapezoid error  Simpson error
     1  0.666666666667  2.6e-02
     2  0.692307692308  8.4e-04          5.7e-02
     3  0.693121693122  2.5e-05          1.5e-02        1.3e-03
     4  0.693146417445  7.6e-07          6.9e-03
     5  0.693147157853  2.3e-08          3.9e-03        1.1e-04
```

With five function values Gauss is correct to 2.3×10^{-8}; Simpson with five values is at 1.1×10^{-4} and the trapezoidal rule at 3.9×10^{-3}. **Figure 10.2** puts all four methods of this chapter on one chart, error against the number of function values; the steeper a line, the more accuracy each extra evaluation buys.

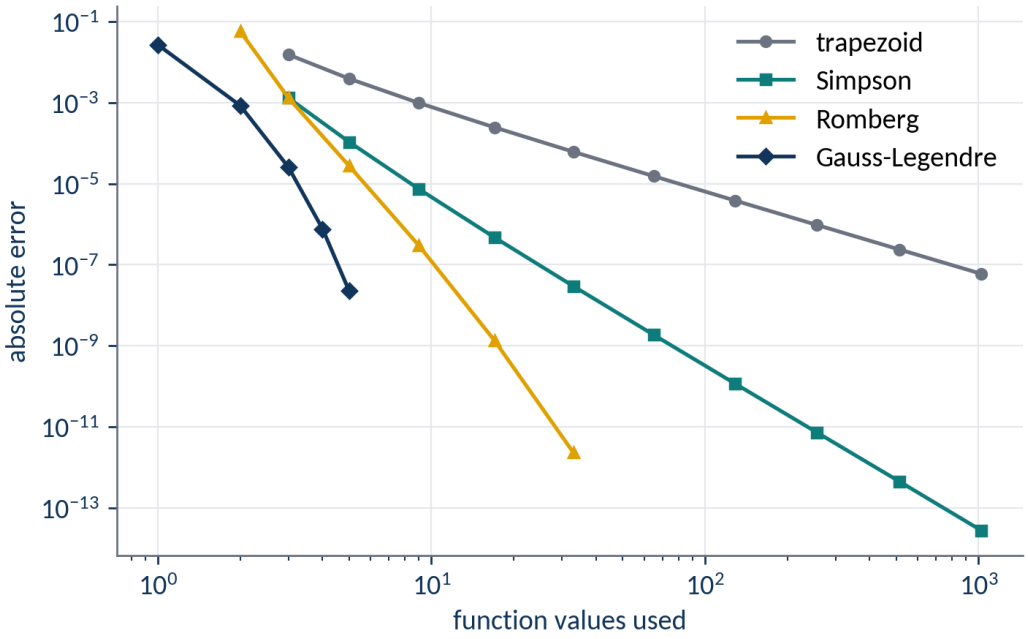


Figure 10.2 Error against the number of function values for $\int_0^1 dx/(1+x) = \ln 2$: trapezoid (order 2), Simpson (order 4), Romberg and Gauss–Legendre.

Gauss's rule has two limitations, both relevant to the orchard. It needs values at irrational positions such as $0.5 \pm 0.5/\sqrt{3}$ of the interval, so it cannot be used with data that were measured at fixed places; Anto's offsets every ten meters are useless to it. And a single high-order rule over a long interval is risky for functions that are not smooth. Both are answered by using Gauss rules on *panels*: split $[a, b]$ into pieces and use a three- or five-point rule on each. If Anto measures again one day, he could stretch his tape at the three Gauss points of each twenty-meter panel, at 2.25 m, 10 m and 17.75 m into the panel, or, with the two-point rule, at 4.23 m and 15.77 m, and get better accuracy with fewer offsets.

How much would that buy? Because the file was made from a known river curve, we can try it. With the two-point rule on each of the eight twenty-meter panels, Anto would take sixteen offsets, one fewer than now, at 4.23 m and 15.77 m into each panel. If he could read the tape perfectly, the area would come out as 7026.81 m^2, wrong by less than a hundredth of a square meter, a hundred times better than Simpson's rule with seventeen offsets. Written down to the nearest 0.1 m, as he actually writes them, the same sixteen offsets give 7028.0 m^2, wrong by 1.2 m^2, no better than Simpson. The lesson of Section 10.5 returns in a sharper form: a cleverer rule only pays when the data are accurate enough to deserve it. For a surveyor with a laser distance meter reading to the millimeter, Gauss points would be a real saving; for a tape read to the decimeter, the choice of rule hardly matters once it is at least Simpson.

10.9 Making It Better: Adaptive Simpson

Uniform steps waste effort. Where a function is smooth, large steps suffice; where it bends sharply, small ones are needed. An *adaptive* rule chooses the steps itself. The classic version is adaptive Simpson [6], [7]: compute Simpson's rule S_1 on an interval and S_2 on its two halves. By Equation (10.7) the error of S_2 is about $(S_2 - S_1)/15$. If that is below the tolerance, accept S_2 plus the correction; otherwise treat each half the same way with half the tolerance. The recursion stops by itself where the function is easy and digs deep only where it is hard.

The test function is $\sqrt{x}$ on $[0,1]$, whose integral is $2/3$. It looks harmless, but its derivative is infinite at $x = 0$, so the error terms of Equation (10.4), which need a bounded $f^{(4)}$, do not apply there, and uniform Simpson converges slowly. **Listing 10.9** counts every evaluation of f.

Listing 10.9 `adaptive.py`: adaptive Simpson integration, compared with uniform Simpson.

```
"""Adaptive Simpson integration of sqrt(x) from 0 to 1 (exact 2/3): small
panels only where the function is hard, near x = 0."""
import math

calls = 0


def f(x):
    global calls
    calls += 1
    return math.sqrt(x)


def simpson_panel(a, fa, m, fm, b, fb):
    return (b - a) / 6 * (fa + 4 * fm + fb)


def adapt(a, fa, m, fm, b, fb, whole, tol, depth):
    """Split [a, b] in two; accept when the two halves agree with the whole."""
    lm, rm = (a + m) / 2, (m + b) / 2
    flm, frm = f(lm), f(rm)
    left = simpson_panel(a, fa, lm, flm, m, fm)
    right = simpson_panel(m, fm, rm, frm, b, fb)
    diff = left + right - whole
    if depth == 0 or abs(diff) <= 15 * tol:
        return left + right + diff / 15          # Richardson correction
    return (adapt(a, fa, lm, flm, m, fm, left, tol / 2, depth - 1)
            + adapt(m, fm, rm, frm, b, fb, right, tol / 2, depth - 1))


def adaptive_simpson(a, b, tol):
    fa, fm, fb = f(a), f((a + b) / 2), f(b)
    whole = simpson_panel(a, fa, (a + b) / 2, fm, b, fb)
    return adapt(a, fa, (a + b) / 2, fm, b, fb, whole, tol, 50)


def uniform_simpson(n):
    h = 1.0 / n
    return h / 3 * (f(0.0) + 4 * sum(f(i * h) for i in range(1, n, 2))
                    + 2 * sum(f(i * h) for i in range(2, n, 2)) + f(1.0))
```

Listing 10.9 (Continued) *adaptive.py: adaptive Simpson integration, compared with uniform Simpson.*

```
def main():
    global calls
    exact = 2 / 3
    print("tolerance   adaptive result  error    function values")
    for tol in (1e-4, 1e-6, 1e-8):
        calls = 0
        r = adaptive_simpson(0.0, 1.0, tol)
        print(f"{tol:9.0e}   {r:.12f}   {abs(r - exact):.1e}  {calls:8d}")
    print("uniform Simpson needs, for an error below 1e-8:")
    n = 2
    while True:
        calls = 0
        err = abs(uniform_simpson(n) - exact)
        if err <= 1e-8:
            print(f"  n = {n} intervals, error {err:.1e}, {calls} function values")
            break
        n *= 2


if __name__ == "__main__":
    main()
```

Listing 10.10 `Adaptive.java`: the same comparison in Java.

```
import java.util.Locale;

/** Adaptive Simpson integration of sqrt(x) from 0 to 1 (exact 2/3): small
 *  panels only where the function is hard, near x = 0. */
public class Adaptive {
    static int calls = 0;

    static double f(double x) {
        calls++;
        return Math.sqrt(x);
    }

    static double simpsonPanel(double a, double fa, double m, double fm,
                               double b, double fb) {
        return (b - a) / 6 * (fa + 4 * fm + fb);
    }

    /** Split [a, b] in two; accept when the two halves agree with the whole. */
    static double adapt(double a, double fa, double m, double fm, double b,
                        double fb, double whole, double tol, int depth) {
        double lm = (a + m) / 2, rm = (m + b) / 2;
        double flm = f(lm), frm = f(rm);
        double left = simpsonPanel(a, fa, lm, flm, m, fm);
        double right = simpsonPanel(m, fm, rm, frm, b, fb);
        double diff = left + right - whole;
        if (depth == 0 || Math.abs(diff) <= 15 * tol) {
            return left + right + diff / 15;          // Richardson correction
        }
        return adapt(a, fa, lm, flm, m, fm, left, tol / 2, depth - 1)
               + adapt(m, fm, rm, frm, b, fb, right, tol / 2, depth - 1);
    }

    static double adaptiveSimpson(double a, double b, double tol) {
        double fa = f(a), fm = f((a + b) / 2), fb = f(b);
        double whole = simpsonPanel(a, fa, (a + b) / 2, fm, b, fb);
        return adapt(a, fa, (a + b) / 2, fm, b, fb, whole, tol, 50);
    }
```

Listing 10.10 (Continued) *Adaptive.java: the same comparison in Java.*

```
    static double uniformSimpson(int n) {
        double h = 1.0 / n, odd = 0, even = 0;
        double first = f(0.0);
        for (int i = 1; i < n; i += 2) {
            odd += f(i * h);
        }
        for (int i = 2; i < n; i += 2) {
            even += f(i * h);
        }
        return h / 3 * (first + 4 * odd + 2 * even + f(1.0));
    }

    public static void main(String[] args) {
        double exact = 2.0 / 3;
        System.out.println("tolerance   adaptive result  error    function values");
        for (double tol : new double[] {1e-4, 1e-6, 1e-8}) {
            calls = 0;
            double r = adaptiveSimpson(0.0, 1.0, tol);
            System.out.println(String.format(Locale.ROOT,
                "%9.0e   %.12f   %.1e  %8d", tol, r, Math.abs(r - exact), calls));
        }
        System.out.println("uniform Simpson needs, for an error below 1e-8:");
        for (int n = 2; ; n *= 2) {
            calls = 0;
            double err = Math.abs(uniformSimpson(n) - exact);
            if (err <= 1e-8) {
                System.out.println(String.format(Locale.ROOT,
                    "  n = %d intervals, error %.1e, %d function values",
                    n, err, calls));
                break;
            }
        }
    }
}
```

Listing 10.10 passes the function values down the recursion exactly as the Python version does, so that no point is evaluated twice. Both print:

```
tolerance   adaptive result  error    function values
    1e-04   0.666663972068   2.7e-06        37
    1e-06   0.666666647130   2.0e-08       105
    1e-08   0.666666666641   2.6e-11       317
uniform Simpson needs, for an error below 1e-8:
  n = 65536 intervals, error 4.8e-09, 65537 function values
```

With 105 function values the adaptive rule reaches an error of 2×10^{-8}; uniform Simpson needs 65,537 values for a similar error, more than six hundred times as many. The adaptive rule spends almost all its evaluations in a tiny region near $x = 0$, where $\sqrt{x}$ rises steeply, and takes a handful of large steps elsewhere. The depth limit of 50 is a safety net against a function that never satisfies the test.

> **WARNING:** An adaptive rule is only as good as its error estimate, and the estimate comes from comparing two answers on the same points. A function with a narrow spike that falls between the first few sample points can fool it completely: both answers miss the spike, agree, and are accepted. When you know where a function misbehaves, split the interval there yourself.

10.10 Checking Against the Library

SciPy's `integrate` module has all of this chapter's rules and more. The current release, SciPy 1.18.1, came out on August 21, 2026, and NumPy 2.5.3 on September 6, 2026 [8], [9]. For *data*,

`trapezoid`, `simpson` and `romb` take arrays of values; `simpson` handles an odd number of strips with a correction at the end, and `romb` needs $2^k + 1$ equally spaced values. For *functions*, `quad` calls the adaptive Gauss–Kronrod routines of the QUADPACK library [10], and `fixed_quad` is plain Gauss–Legendre with n points [11]. Two names from older tutorials are gone: `numpy.trapz` was removed in NumPy 2.4 in favor of `numpy.trapezoid` [12], and `scipy.integrate.romberg` and `quadrature` were removed in SciPy 1.15 in favor of `quad` [13]. **Listing 10.11** checks our numbers.

Listing 10.11 `compare_scipy.py`: the orchard and the test integrals with SciPy.

```
"""Check this chapter's rules against SciPy."""
import math
import numpy as np
import scipy
from scipy import integrate
from orchard import load

x, y = load("../data/orchard.csv")
print(f"NumPy {np.__version__}, SciPy {scipy.__version__}")
print(f"orchard, trapezoid: {integrate.trapezoid(y, x):.1f} m2")
print(f"orchard, simpson:   {integrate.simpson(y, x=x):.1f} m2")
print(f"orchard, romb:      {integrate.romb(y, dx=x[1] - x[0]):.1f} m2")
val, err = integrate.quad(lambda t: 1 / (1 + t), 0, 1)
print(f"quad 1/(1+x):   {val:.15f} (estimated error {err:.1e})")
val, _ = integrate.fixed_quad(lambda t: 1 / (1 + t), 0, 1, n=5)
print(f"fixed_quad n=5: {val:.12f}")
val, err, info = integrate.quad(math.sqrt, 0, 1, full_output=True)[:3]
print(f"quad sqrt(x):   {val:.15f} ({info['neval']} function values)")
```

```
NumPy 2.4.4, SciPy 1.17.1
orchard, trapezoid: 7023.5 m2
orchard, simpson:   7027.7 m2
orchard, romb:      7026.6 m2
quad 1/(1+x):   0.693147180559945 (estimated error 7.7e-15)
fixed_quad n=5: 0.693147157853
quad sqrt(x):   0.666666666666667 (231 function values)
```

The trapezoid and Simpson areas agree with ours to the printed digit. `romb`, which applies the full Romberg table to Anto's seventeen offsets, gives 7026.6 m^2; as Section 10.7 warned, the high extrapolation columns are less trustworthy on measured data, although here it happens to land within a quarter of a square meter of the truth. `fixed_quad` with five points reproduces our Gauss value, `0.693147157853`, digit for digit. And `quad` finds ln2 to full precision and even $\int_0^1 \sqrt{x}\,dx$ to full precision with 231 function values, more than our adaptive Simpson used for 10^{-8} but for sixteen correct digits: QUADPACK's Gauss–Kronrod rules combine the two big ideas of this chapter, Gauss points and adaptivity.

The code folder also has tests (`test_integ.py`, `IntegTest.java`) that check facts no correct implementation can violate: the trapezoidal rule is exact for a straight line, Simpson's for a cubic, three-point Gauss for x^5, Romberg's diagonal beats the trapezoid it starts from, and a rectangular "orchard" has the area width times length.

10.11 Numerical Methods in Practice: Profitina

Profitina's historical analytics dashboard keeps a weekly AI-visibility score for each brand and draws its trend. A brand owner often wants a single number for a period, such as "what was my average score this quarter?", and an average over time is an integral divided by the length of the period. When the scans are equally spaced, the trapezoidal rule over the weekly scores is the natural average, with half weight on the first and last week, which is fairer than a plain mean

when a period starts or ends in the middle of a change. When scans are not equally spaced, as happens when an owner requests extra scans before a campaign, the trapezoidal rule on the actual scan dates still works, while a plain mean would give the busy weeks too much weight. The same reasoning applies to "total improvement achieved during the campaign": the area between the score curve and the starting level.

Profitina's PROFIT utility coin is designed with a decaying emission: new coins are released at a rate that falls over time, and some are burned. The total emitted over any period is the integral of the emission rate. For a rate given by a simple formula this can be done by hand, but a schedule that changes in steps, or that is combined with actual daily burns, is a table of values, and the tools of this chapter, trapezoid for data and Gauss or adaptive rules for smooth formulas, are how such totals are computed and checked. The exact emission parameters and oracle internals are Profitina's own; the mathematics is this chapter's, and, as always, the numbers advise while the user decides.

10.12 Using It for Real

- **Measure offsets at right angles, at equal spacing, with an even number of strips.** Then Simpson's rule applies directly and the error estimate of Equation (10.7) is free. If the land ends at an odd strip, use the 3/8 rule on the last three.
- **Measure more where the boundary bends.** Where the river turns, add offsets at 5 m spacing and treat that stretch as a separate piece with its own step; the pieces add up. This is adaptivity done by hand.
- **Cut irregular land into pieces with straight baselines.** A plot with no straight side can be divided into a polygon (whose area comes from corner coordinates) plus strips along each side, each integrated from its own baseline.
- **Rain into liters.** Rainfall intensity from a rain gauge or weather station, in millimeters per hour, integrated over time gives millimeters of rain; one millimeter on one square meter is one liter. Multiply by the roof area and by the share that reaches the tank (Exercise 10.6).
- **Areas into volumes.** The same rules integrate twice. To know how much water a fish pond or an irrigation pond on the orchard holds, measure the depth along several parallel lines across it; integrate each line's depths with Simpson's rule to get a cross-sectional area in square meters, then integrate those areas along the pond with Simpson's rule again to get cubic meters. One cubic meter is a thousand liters, which tells you how long a pump of known capacity needs to empty or fill it, and how much lime or fish feed a given dose per cubic meter requires.
- **Power into energy.** A smart plug that reports a washing machine's power in watts every minute can be integrated to watt-hours with the trapezoidal rule; a laundry such as Profitina Laundry, the case study used throughout this book, could compare its machines this way.
- **Estimate the error with the data you have.** Halve the data (every other point), recompute, and use Equation (10.7). If the two answers disagree by more than you can accept, measure more densely.
- **Leave official numbers to officials.** A computed area is for planning. Land certificates in Indonesia are issued from official measurements by the land agency's surveyors, and the tape-and-Simpson answer is best used to know what to expect.

10.13 Common Mistakes, and How to Avoid Them

- **Simpson with an odd number of strips.** The weights 1,4,2,4, … ,4,1 only make sense for an even number of strips. Our function raises an error; a careless one silently returns a wrong area. Use the 3/8 rule for the odd remainder.
- **Forgetting the halves at the ends.** The trapezoidal rule gives the end values weight $\frac{1}{2}$. A plain sum times h is the rectangle rule, which is only first order and too large or too small by about half a strip at each end.
- **Unequal spacing with an equal-spacing formula.** If one offset was taken at 25 m instead of 20 m, the composite formulas are wrong. Integrate each strip with its own width, or pass the actual positions (`x=`) to SciPy.
- **Trusting high-order extrapolation on noisy data.** Romberg's upper columns assume a smooth function. On measured values, stop after one or two columns.
- **Expecting Gauss to use your data.** Gauss rules need values at their own points. For tabulated data use trapezoid, Simpson or `romb`.
- **Using `np.trapz` or `scipy.integrate.romberg`.** Both have been removed. Use `numpy.trapezoid` or `scipy.integrate.trapezoid`, and `scipy.integrate.quad`.
- **Ignoring singular behavior.** Error formulas such as Equation (10.4) need bounded derivatives. At a square root, a jump or a kink the order drops; split the interval at the trouble spot or use an adaptive rule.

10.14 Chapter Summary

- Every quadrature rule integrates a simple interpolating curve exactly. The trapezoidal rule uses lines and has error $O(h^2)$ (Equation (10.1), Equation (10.2)); Simpson's 1/3 rule uses parabolas, is exact for cubics and has error $O(h^4)$ (Equation (10.4)); the 3/8 rule (Equation (10.6)) handles an odd number of strips.
- The Euler–Maclaurin formula (Equation (10.3)) shows that the trapezoidal error is a series in h^2. Richardson extrapolation of it gives Simpson's rule (Equation (10.5)) and, repeated, Romberg's table (Equation (10.8)).
- Comparing an answer with step h and with step $2h$ estimates its error without knowing the truth (Equation (10.7)); for the orchard it predicted the trapezoid and Simpson errors well and gave an area of 7026.9 m² against a true 7026.8 m².
- Gauss–Legendre quadrature chooses the points as well as the weights and integrates polynomials of degree $2n - 1$ exactly with n points (Equation (10.9), Equation (10.10)); it needs a function, not fixed data.
- Adaptive Simpson puts small steps only where the function is hard; for $\sqrt{x}$ it needed 105 values where uniform Simpson needed 65,537.
- SciPy provides `trapezoid`, `simpson` and `romb` for data and `quad` and `fixed_quad` for functions; `np.trapz` and `integrate.romberg` no longer exist.

10.15 Exercises

1. By hand, compute the area of the first 40 m of the orchard (offsets 38.6, 40.0, 41.2, 42.2 and 43.1 m) with the trapezoidal rule and with Simpson's rule, and estimate the trapezoid's error from the difference.

2. The neighbor's fence stops the tape at 150 m, so only the first sixteen offsets (0 to 150 m, fifteen strips) are usable. Compute the area with Simpson's 1/3 rule on the first twelve strips and the 3/8 rule on the last three, and compare with the trapezoidal rule. The true area of that part is 6692.5 m^2.
3. Prove Equation (10.5): write out $4T(h) - T(2h)$ for $n = 2$ strips of width h and show that dividing by 3 gives Simpson's rule. Why does the argument extend to any even n?
4. Repeat the order check of Section 10.6 with $\int_0^1 \frac{4}{1+x^2}\,dx = \pi$. You will find that the Simpson ratios approach 64, not 16. Explain this with Equation (10.3).
5. Derive the three-point Gauss–Legendre rule on $[-1,1]$ by requiring exactness for $1, x^2$ and x^4 with points $0, \pm t$ and weights w_0, w_1. Apply it by hand to $\ln 2 = \int_0^1 d\,x/(1+x)$.
6. A rain gauge on Anto's house records the rainfall intensity every ten minutes during an afternoon storm: 0, 6, 18, 42, 30, 12, 6 and 0 mm per hour. Estimate the total rainfall in millimeters with the trapezoidal rule and with Simpson's rules, and the liters that reach the family's water tank from a 60 m^2 roof if 80% of the rain is collected.

10.16 Solutions to the Exercises

1. With $h = 10$: trapezoid $T = 10 \times (38.6/2 + 40.0 + 41.2 + 42.2 + 43.1/2) = 10 \times 164.25 = 1642.5$ m^2. Simpson $S = \frac{10}{3}(38.6 + 4 \times 40.0 + 2 \times 41.2 + 4 \times 42.2 + 43.1) = \frac{10}{3} \times 492.9 = 1643.0$ m^2. With every other offset (step 20), $T(2h) = 20 \times (38.6/2 + 41.2 + 43.1/2) = 20 \times 82.05 = 1641.0$, so by Equation (10.7) the trapezoid's error is about $(1642.5 - 1641.0)/3 = 0.5$ m^2, and indeed $1642.5 + 0.5 = 1643.0$ is Simpson's value. The two rules agree to half a square meter on this nearly straight stretch.

2. Simpson's 1/3 rule on 0 to 120 m (twelve strips, offsets 38.6 to 47.2) gives 5480.0 m^2. The 3/8 rule on 120 to 150 m gives $\frac{3\times10}{8}(47.2 + 3 \times 42.2 + 3 \times 38.2 + 35.0) = 3.75 \times 323.4 = 1212.75$ m^2. The total is 6692.75 m^2, within 0.3 m^2 of the true 6692.5. The trapezoidal rule on all fifteen strips gives 6689.0 m^2, 3.5 m^2 short.

3. For $n = 2$, $T(h) = h\left(\frac{f_0}{2} + f_1 + \frac{f_2}{2}\right)$ and $T(2h) = 2h \cdot \frac{f_0+f_2}{2} = h(f_0 + f_2)$. Then $4T(h) - T(2h) = h(2f_0 + 4f_1 + 2f_2 - f_0 - f_2) = h(f_0 + 4f_1 + f_2)$, and dividing by 3 gives $\frac{h}{3}(f_0 + 4f_1 + f_2)$, Simpson's rule. For any even n both $T(h)$ and $T(2h)$ are sums over consecutive pairs of strips, and the identity holds pair by pair; adding the pairs gives the composite Simpson rule Equation (10.4).

4. Changing `f` and the exact value in `quadrature.py` gives Simpson errors of 2.365×10^{-9} for $n = 16$ and 3.696×10^{-11} for $n = 32$, a ratio of 64.0, and the same ratio for the next doubling. Simpson's rule is $\frac{4T(h)-T(2h)}{3}$, which removes the h^2 term of Equation (10.3); its leading error is then the h^4 term, whose coefficient is proportional to $f'''(1) - f'''(0)$. For $f(x) = 4/(1+x^2)$, $f'''(x) = 96x(1-x^2)/(1+x^2)^4$, which is zero at both $x = 0$ and $x = 1$. The h^4 term vanishes, the leading error is the h^6 term, and the ratio is $2^6 = 64$. The rule did not become better in general; this integrand is simply lucky at its end points. The trapezoidal ratios stay at 4 because $f'(1) - f'(0) = -2 - 0 \neq 0$.

5. By symmetry the outer weights are equal and odd powers are exact. Exactness for 1: $w_0 + 2w_1 = 2$. For x^2: $2w_1t^2 = \frac{2}{3}$. For x^4: $2w_1t^4 = \frac{2}{5}$. Dividing the last two gives $t^2 = \frac{3}{5}$, so $t = \sqrt{3/5} \approx 0.774597$; then $w_1 = \frac{1}{3t^2} = \frac{5}{9}$ and $w_0 = 2 - \frac{10}{9} = \frac{8}{9}$. For ln2 the nodes map to $x = 0.5 - 0.387298 =$

0.112702, 0.5 and 0.887298, where $f = 0.898715$, 0.666667 and 0.529858. The rule gives $\frac{1}{2}\left(\frac{5}{9}\times 0.898715 + \frac{8}{9}\times 0.666667 + \frac{5}{9}\times 0.529858\right) = \frac{1}{2}(0.499286 + 0.592593 + 0.294366) = 0.693122$, in agreement with the program's 0.693121693 and an error of 2.5×10^{-5}.

6. The step is $h = 10$ min, that is $\frac{1}{6}$ h, and there are seven strips. Trapezoid: $\frac{1}{6}(0/2 + 6 + 18 + 42 + 30 + 12 + 6 + 0/2) = \frac{114}{6} = 19.0$ mm. Simpson: seven strips are odd, so use the 1/3 rule on the first four strips and the 3/8 rule on the last three: $\frac{1}{18}(0 + 4\times 6 + 2\times 18 + 4\times 42 + 30) = \frac{258}{18} = 14.33$ mm, plus $\frac{3}{8}\cdot\frac{1}{6}(30 + 3\times 12 + 3\times 6 + 0) = \frac{84}{16} = 5.25$ mm, total 19.58 mm. One millimeter on one square meter is one liter, so the roof receives $19.58 \times 60 = 1175$ liters and the tank about $0.8 \times 1175 = 940$ liters (the trapezoidal answer gives 912 liters). With readings only every ten minutes, the difference between the two rules, about 3%, is a fair measure of the uncertainty; the rain gauge's own accuracy is likely worse.

References

[1] S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers*, 8th ed. New York, NY, USA: McGraw-Hill Education, 2021.

[2] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical Analysis*, 10th ed. Boston, MA, USA: Cengage Learning, 2016.

[3] P. J. Davis and P. Rabinowitz, *Methods of Numerical Integration*, 2nd ed. Orlando, FL, USA: Academic Press, 1984; reprinted Mineola, NY, USA: Dover, 2007.

[4] W. Romberg, "Vereinfachte numerische Integration," *Det Kongelige Norske Videnskabers Selskab Forhandlinger*, vol. 28, no. 7, pp. 30–36, 1955.

[5] G. H. Golub and J. H. Welsch, "Calculation of Gauss quadrature rules," *Mathematics of Computation*, vol. 23, no. 106, pp. 221–230, 1969.

[6] W. M. McKeeman, "Algorithm 145: Adaptive numerical integration by Simpson's rule," *Communications of the ACM*, vol. 5, no. 12, p. 604, 1962.

[7] W. Gander and W. Gautschi, "Adaptive quadrature—revisited," *BIT Numerical Mathematics*, vol. 40, no. 1, pp. 84–101, 2000.

[8] SciPy Developers, "scipy 1.18.1," Python Package Index, Aug. 21, 2026. [Online]. Available: https://pypi.org/project/scipy/

[9] NumPy Developers, "numpy 2.5.3," Python Package Index, Sep. 6, 2026. [Online]. Available: https://pypi.org/project/numpy/

[10] R. Piessens, E. de Doncker-Kapenga, C. W. Überhuber, and D. K. Kahaner, *QUADPACK: A Subroutine Package for Automatic Integration*. Berlin, Germany: Springer, 1983.

[11] SciPy Developers, "Integration and ODEs (scipy.integrate)," SciPy v1.18 Manual, accessed Sep. 2026. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/integrate.html

[12] NumPy Developers, "NumPy 2.4.0 Release Notes," accessed Sep. 2026. [Online]. Available: https://numpy.org/doc/stable/release/2.4.0-notes.html

[13] SciPy Developers, "SciPy 1.15.0 Release Notes," accessed Sep. 2026. [Online]. Available: https://docs.scipy.org/doc/scipy/release/1.15.0-notes.html

Chapter 11

Ordinary Differential Equations: Euler to Runge-Kutta

Many everyday questions are about how something changes when we know what drives the change: how fast a water tank empties when the tap is open, how a motorbike seat settles after a speed bump, how a savings balance grows with interest. The rule for the *rate* of change is often easy to write down; the quantity itself, at a later time, is not. An ordinary differential equation states the rule, and a numerical method walks forward in time step by step to find the quantity. By the end of this chapter you will be able to write a real situation as an initial value problem and, where possible, solve it exactly to have something to check against; derive Euler's method and bound its error; derive second-order Runge–Kutta methods from Taylor's theorem and use the classical fourth-order method; measure the order of each method from its own output; simulate a morning of an orchard's drip-irrigation tank in Java and in Python, with a pump, a valve, an overflow and running dry; plan the pump with the bisection method of Chapter 2 wrapped around the simulation; turn a second-order equation into a system and see why Euler's method can make a stable system blow up; estimate the error of a simulation from the simulation itself; and check the answers with SciPy's `solve_ivp`.

11.1 Will the Tank Last Until Ten?

The dry season has come to the fruit and durian orchard of Chapter 10. Anto, who now runs it, has planted seventy young durian trees, and young durian trees must not go thirsty. He has built a simple drip-irrigation system: a 1,000-liter plastic water tank on a stand at the top of the slope, a hose down the rows, and a small dripper at the foot of each tree. No electricity is needed for the watering itself; gravity pushes the water through. The only machine is an electric pump that lifts water from the well into the tank.

Every morning the same routine: at six o'clock he switches the pump on, at half past six he opens the valve to the drip line, and later, when he remembers, he switches the pump off. Two things go wrong. Some mornings he forgets the pump, the tank fills up, and clean well water pours out of the overflow pipe onto the ground, together with the electricity that pumped it. Other mornings the tank runs dry before ten, and the trees at the far end of the rows get too little. He would like to know: *if I switch the pump off at a given time, how high will the tank get, how much will overflow, and when will the drippers stop?* And the real question behind it: *when should I switch the pump off?*

He could find out by trial, one morning at a time. But the tank obeys simple physics, and simple physics can be written as an equation. The level of the water changes because water comes in (the pump) and goes out (the drippers). The flow out through the drippers depends on how high the water stands above them: a full tank pushes harder than an almost empty one. So the *rate* of change of the level depends on the level itself. That is a **differential equation**, and this chapter is about solving such equations numerically, by stepping forward in time: from the level at six o'clock, the level a minute later; from that, the level a minute after that; and so on, until ten o'clock and beyond.

The same stepping lies behind weather forecasts, the orbits of satellites, the spread of a disease in a population, the charging curve of a phone battery, and every physics engine in a video game. We start with the simplest method, understand exactly how wrong it is, and improve it until one morning of the tank can be simulated to better than a millimeter in a fraction of a second.

11.2 The Idea in Plain Words

Suppose the tank is full, the pump is off, and the valve is open. Anto knows from experience that a full tank drips out about ten liters a minute. His tank has a floor area of one square meter, so ten liters (0.01 m^3) a minute lowers the level by one centimeter a minute. If the flow stayed at ten liters a minute, the level after ten minutes would be $1.00 - 10 \times 0.01 = 0.90$ m.

But the flow does not stay the same. As the level drops, the pressure at the drippers drops, and the flow slows down. So after ten minutes, at 0.90 m, we should recompute the flow before taking the next step. For the drippers, the flow is proportional to the square root of the water height (Section 11.3 explains why), so at 0.90 m it is $10 \times \sqrt{0.90} = 9.49$ liters a minute, and in the next ten minutes the level drops by about 9.49 cm, to $0.90 - 0.0949 = 0.805$ m. Step, recompute the rate, step again: that is **Euler's method**, the first and simplest method for differential equations, published by Leonhard Euler in 1768 [1].

We can already see its weakness. During the first ten minutes we used the flow at the *start* of the ten minutes, 10 L/min, although the flow was falling the whole time, to 9.49 L/min at the end. So we took out a little too much water, and the level we computed, 0.900 m, is a little too low. The true level after ten minutes, from the exact formula of Section 11.3, is 0.9025 m. The error, 2.5 mm, is small, but it is repeated at every step, and each step starts from a level that is already slightly wrong. Section 11.4 measures how these errors add up; Section 11.5 fixes them by looking at the rate not only at the start of each step but also in the middle and at the end.

> **KEY INSIGHT:** A differential equation does not tell you where you are; it tells you which way to go from wherever you are. Every numerical method for it is a strategy for walking: look at the direction, take a step, look again. The methods differ in how carefully they look before each step.

11.3 The Model and Its Exact Solution

11.3.1 The mass balance

Let $h(t)$ be the water level in meters at time t in minutes, and A the floor area of the tank in square meters. The volume of water is Ah, and its rate of change is the flow in minus the flow out:

$$A\frac{dh}{dt} = Q_{\text{in}}(t) - k\sqrt{h}. \tag{11.1}$$

Here Q_{in} is the pump's flow in cubic meters per minute (0.02 for 20 L/min, or 0 when the pump is off) and $k\sqrt{h}$ is the flow out through the drip line. The square root comes from Torricelli's law: water leaving an opening under a head h has speed about $\sqrt{2gh}$, where g is the acceleration of gravity, so the flow through a fixed set of openings is proportional to $\sqrt{h}$ [2]. The constant k lumps together the number of drippers, their size, and $\sqrt{2g}$; Anto found it the practical way, from the observation that a full tank ($h = 1$) gives 10 L/min, so $k = 0.01$. His tank has $A = 1$ m^2 and is 1 m high, so it holds 1,000 liters and the level in meters equals the volume in cubic meters.

Equation (11.1) is an **ordinary differential equation** (ODE): an equation for an unknown function $h(t)$ that involves its derivative. *Ordinary* means the function depends on one variable, time; equations with several variables, such as the temperature across a metal plate, are *partial* differential equations and are not treated in this book. Together with a starting value, $h(0) = h_0$, it forms an **initial value problem**. In general we write

$$y'(t) = f(t, y(t)), \qquad y(t_0) = y_0, \tag{11.2}$$

where f is a known function that gives the slope for any time and any value. For the tank, $y = h$ and $f(t, h) = \left(Q_{\text{in}}(t) - k\sqrt{h}\right)/A$. Under mild conditions on f (it should be continuous and satisfy a Lipschitz condition, $f(t, y) - f(t, z)L y - z$) the problem has exactly one solution [3]. The square root violates the Lipschitz condition at $h = 0$, which is why an empty tank needs a little care in the programs.

11.3.2 An exact solution to check against

With the pump off, Equation (11.1) can be solved by hand, which gives us a yardstick for the numerical methods. Separate the variables, putting everything with h on one side:

$$\frac{dh}{\sqrt{h}} = -\frac{k}{A}\,dt.$$

The left side is the derivative of $2\sqrt{h}$; integrating from time 0, when the level is h_0, to time t gives $2\sqrt{h} - 2\sqrt{h_0} = -\frac{k}{A}t$, so

$$h(t) = \left(\sqrt{h_0} - \frac{kt}{2A}\right)^2, \qquad 0 \le t \le \frac{2A\sqrt{h_0}}{k}. \tag{11.3}$$

For Anto's full tank, $h_0 = 1$, $k = 0.01$ and $A = 1$, so $h(t) = (1 - 0.005\,t)^2$: 0.9025 m after 10 minutes, 0.81 m after 20, 0.49 m after an hour, and empty after exactly 200 minutes. This formula is what every method in the chapter will be tested against. Most real problems have no such formula, which is the reason numerical methods exist, but a method that fails on a problem with a known answer will not do better on one without.

11.4 Euler's Method

11.4.1 Derivation

Divide time into steps of length h (the letter h is traditional for the step; in the programs the water level is `h` and the step is `step`, to avoid confusion), with $t_n = t_0 + nh$, and let y_n be our approximation of $y(t_n)$. Taylor's theorem (Chapter 9) gives

$$\begin{aligned} y(t_{n+1}) &= y(t_n) + hy'(t_n) + \frac{h^2}{2}y''(\xi_n) \\ &= y(t_n) + hf(t_n, y(t_n)) + \frac{h^2}{2}y''(\xi_n). \end{aligned}$$

Dropping the last term and using our approximation y_n in place of the unknown $y(t_n)$ gives **Euler's method**:

$$y_{n+1} = y_n + hf(t_n, y_n). \tag{11.4}$$

Geometrically, each step follows the tangent line of the solution through the current point for a time h. For the tank with a step of 10 minutes, the first two steps are exactly the hand computation of Section 11.2: $h_1 = 1 + 10 \times \left(-0.01\sqrt{1}\right) = 0.9$ and $h_2 = 0.9 + 10 \times \left(-0.01\sqrt{0.9}\right) = 0.805132$.

11.4.2 Local and global error

The term we dropped, $\frac{h^2}{2}y''$, is the **local truncation error**: the error made in one step if that step started from the exact value. It is $O(h^2)$. But we take $(t_{\text{end}} - t_0)/h$ steps, and the errors accumulate. A careful argument [3], [4] shows that if $|y''| \le M$ and f has Lipschitz constant L, the **global error** at time t_n satisfies

$$|y(t_n) - y_n| \le \frac{hM}{2L}\left(e^{L(t_n - t_0)} - 1\right). \tag{11.5}$$

The important part is the factor h: the global error of Euler's method is $O(h)$, one order lower than the local error, because $O(1/h)$ steps each contribute $O(h^2)$. Euler's method is a *first-order* method: halve the step, halve the error. The exponential factor warns that errors can grow over long times when the equation itself amplifies differences; for the draining tank the opposite happens (two tanks at different levels approach each other as they drain), and the bound is pessimistic.

11.4.3 The first program

Listing 11.1 applies Equation (11.4) to the draining tank with a step of 10 minutes, prints each step next to the exact solution Equation (11.3), and then repeats the computation with steps of 5, 2.5 and 1.25 minutes to see how the error at one hour changes.

Listing 11.1 `tank_euler.py`: Euler's method for the draining tank, compared with the exact solution.

```python
"""Euler's method for a draining water tank: dh/dt = -k*sqrt(h)/A.
Full tank h = 1 m, area A = 1 m^2, drip-line constant k = 0.01; exact
solution h(t) = (1 - 0.005 t)^2 with t in minutes."""
import math

K, AREA = 0.01, 1.0


def slope(t, h):
    return -K * math.sqrt(max(h, 0.0)) / AREA


def exact(t):
    return (1.0 - K * t / (2 * AREA)) ** 2


def euler(f, t0, y0, step, t_end):
    t, y = t0, y0
    rows = [(t, y)]
    while t < t_end - 1e-9:
        y = y + step * f(t, y)          # follow the tangent for one step
        t = t + step
        rows.append((t, y))
    return rows


def main():
    print("  t(min)   Euler h(m)   exact h(m)     error")
    for t, y in euler(slope, 0.0, 1.0, 10.0, 60.0):
        print(f"{t:8.0f}   {y:10.6f}   {exact(t):10.6f}   {y - exact(t):9.6f}")
    print("error at t = 60 min for smaller steps:")
    old = None
    for step in (10.0, 5.0, 2.5, 1.25):
        err = euler(slope, 0.0, 1.0, step, 60.0)[-1][1] - exact(60.0)
        ratio = f"  ratio {old / err:.2f}" if old else ""
        print(f"  step {step:5.2f} min   error {err:.6f}{ratio}")
        old = err


if __name__ == "__main__":
    main()
```

The function `euler` does not know about tanks: it takes any slope function `f(t, y)`, a start, a step and an end time, which is the general form Equation (11.2). The slope uses `max(h, 0.0)` so that a level that dips a hair below zero by rounding does not make the square root fail. The loop condition `t < t_end - 1e-9` guards against a step count that is off by one because of rounding in `t`. **Listing 11.2** is the Java version, with the slope passed as a `DoubleBinaryOperator`.

Listing 11.2 `TankEuler.java`: the same in Java.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;
import java.util.function.DoubleBinaryOperator;

/** Euler's method for a draining water tank: dh/dt = -k*sqrt(h)/A.
 *  Full tank h = 1 m, area A = 1 m^2, drip-line constant k = 0.01; exact
 *  solution h(t) = (1 - 0.005 t)^2 with t in minutes. */
public class TankEuler {
    static final double K = 0.01, AREA = 1.0;

    static double slope(double t, double h) {
        return -K * Math.sqrt(Math.max(h, 0.0)) / AREA;
    }

    static double exact(double t) {
        double r = 1.0 - K * t / (2 * AREA);
        return r * r;
    }

    /** Returns rows {t, y}. */
    static List<double[]> euler(DoubleBinaryOperator f, double t0, double y0,
                                double step, double tEnd) {
        double t = t0, y = y0;
        List<double[]> rows = new ArrayList<>();
        rows.add(new double[] {t, y});
        while (t < tEnd - 1e-9) {
            y = y + step * f.applyAsDouble(t, y);   // follow the tangent one step
            t = t + step;
            rows.add(new double[] {t, y});
        }
        return rows;
    }

    public static void main(String[] args) {
        System.out.println("  t(min)   Euler h(m)   exact h(m)     error");
        for (double[] r : euler(TankEuler::slope, 0.0, 1.0, 10.0, 60.0)) {
            System.out.println(String.format(Locale.ROOT,
                "%8.0f   %10.6f   %10.6f   %9.6f",
                r[0], r[1], exact(r[0]), r[1] - exact(r[0])));
        }
        System.out.println("error at t = 60 min for smaller steps:");
        double old = 0;
        for (double step : new double[] {10.0, 5.0, 2.5, 1.25}) {
            List<double[]> rows = euler(TankEuler::slope, 0.0, 1.0, step, 60.0);
            double err = rows.get(rows.size() - 1)[1] - exact(60.0);
```

Listing 11.2 (Continued) *TankEuler.java: the same in Java.*

```
            String ratio = old != 0
                ? String.format(Locale.ROOT, "  ratio %.2f", old / err) : "";
            System.out.println(String.format(Locale.ROOT,
                "  step %5.2f min   error %.6f%s", step, err, ratio));
            old = err;
        }
    }
}
```

Both print:

```
  t(min)   Euler h(m)   exact h(m)     error
       0     1.000000     1.000000    0.000000
      10     0.900000     0.902500   -0.002500
      20     0.805132     0.810000   -0.004868
      30     0.715403     0.722500   -0.007097
      40     0.630821     0.640000   -0.009179
      50     0.551397     0.562500   -0.011103
      60     0.477141     0.490000   -0.012859
error at t = 60 min for smaller steps:
  step 10.00 min   error -0.012859
  step  5.00 min   error -0.006335  ratio 2.03
  step  2.50 min   error -0.003144  ratio 2.01
  step  1.25 min   error -0.001566  ratio 2.01
```

Every Euler value is below the exact one, and the error grows steadily, to 1.3 cm after an hour. The reason is the one we saw by hand: the outflow is always taken at the start of the step, when the level, and therefore the flow, is highest, so each step drains too much. The second table confirms the order: each halving of the step halves the error (ratio 2.0). **Figure 11.1** shows the Euler points falling slowly below the exact curve, next to the points of the RK4 method of the next section, which cannot be told apart from the curve.

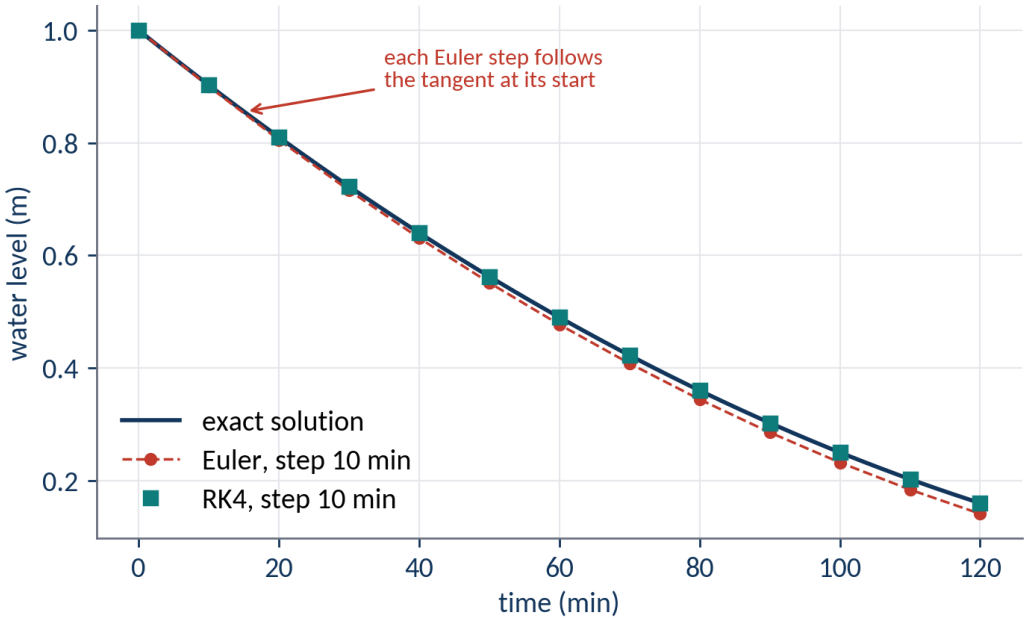


Figure 11.1 The draining tank: the exact solution, Euler's method with a 10-minute step (each step follows the tangent at its start and drains a little too much), and the classical Runge–Kutta method with the same step.

For Anto, 1.3 cm is 13 liters, not a disaster. But first order is expensive: to get the error down to a millimeter, Euler needs steps of about one minute; for a tenth of a millimeter, about five seconds. We can do far better with the same number of slope evaluations by looking more carefully within each step.

11.5 Runge–Kutta Methods

11.5.1 Second order: Heun and midpoint

The trouble with Euler is that it uses only the slope at the start of the step. The average slope over the step would be exactly right, but it is unknown. We can estimate it. **Heun's method** first takes

a trial Euler step to the end of the interval, evaluates the slope there, and then steps with the average of the two slopes:

$$k_1 = f(t_n, y_n), \qquad k_2 = f(t_n + h,\ y_n + hk_1), \qquad y_{n+1} = y_n + \frac{h}{2}(k_1 + k_2).$$

The **midpoint method** instead takes a half Euler step, evaluates the slope in the middle, and uses that for the whole step: $k_2 = f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right)$ and $y_{n+1} = y_n + hk_2$. Both use two slope evaluations per step, twice Euler's work. Both are second order. Here is why, for the general two-stage method

$$k_1 = f(t_n, y_n), \quad k_2 = f(t_n + ch,\ y_n + ahk_1), \quad y_{n+1} = y_n + h(b_1k_1 + b_2k_2). \tag{11.6}$$

By the chain rule, the exact solution satisfies $y'' = f_t + ff_y$, where f_t and f_y are the partial derivatives of f, so its Taylor expansion is $y(t+h) = y + hf + \frac{h^2}{2}(f_t + ff_y) + O(h^3)$. Expanding k_2 in Taylor series in two variables gives $k_2 = f + chf_t + ahk_1f_y + O(h^2)$, so the method gives $y + h(b_1 + b_2)f + h^2b_2(cf_t + aff_y) + O(h^3)$. Matching the two expansions term by term requires

$$b_1 + b_2 = 1, \qquad b_2c = \frac{1}{2}, \qquad b_2a = \frac{1}{2}.$$

Three equations for four unknowns: a one-parameter family of second-order methods. Heun is $b_1 = b_2 = \frac{1}{2}$, $a = c = 1$; midpoint is $b_1 = 0$, $b_2 = 1$, $a = c = \frac{1}{2}$. Their local error is $O(h^3)$ and their global error $O(h^2)$. For the tank's first step of 10 minutes, Heun gives $k_1 = -0.01$, $k_2 = -0.01\sqrt{0.9} = -0.0094868$, and $h_1 = 1 + 5 \times (-0.0194868) = 0.902566$, only 0.07 mm above the exact 0.9025, against Euler's 2.5 mm below.

11.5.2 The classical fourth-order method

The same idea with four slope evaluations gives the method that Carl Runge and Wilhelm Kutta developed around 1900 [5] and that is still the default choice of countless programs, the **classical Runge–Kutta method**, or **RK4**:

$$\begin{aligned} k_1 &= f(t_n, y_n), & k_2 &= f\left(t_n + \frac{h}{2},\ y_n + \frac{h}{2}k_1\right), \\ k_3 &= f\left(t_n + \frac{h}{2},\ y_n + \frac{h}{2}k_2\right), & k_4 &= f(t_n + h,\ y_n + hk_3), \\ y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4). \end{aligned} \tag{11.7}$$

It samples the slope at the start (k_1), twice in the middle (k_2 from a half step with k_1, k_3 from a half step with k_2) and at the end (k_4), and averages them with weights 1,2,2,1. Matching Taylor expansions to fourth order requires eight conditions, which these weights satisfy [4], [6]; the local error is $O(h^5)$ and the global error $O(h^4)$.

The weights should look familiar. If f depends only on t, the differential equation is just $y' = f(t)$ and its solution is an integral; then $k_2 = k_3 = f(t_n + h/2)$ and RK4 becomes $y_{n+1} = y_n + \frac{h}{6}\left(f(t_n) + 4f\left(t_n + \frac{h}{2}\right) + f(t_n + h)\right)$, which is Simpson's rule from Chapter 10 on the interval $[t_n, t_n + h]$. RK4 is Simpson's rule generalized to slopes that depend on the solution itself.

One RK4 step for the tank by hand, with a step of 10 minutes from $h = 1$: $k_1 = -0.01\sqrt{1} = -0.01$; $k_2 = -0.01\sqrt{1 + 5k_1} = -0.01\sqrt{0.95} = -0.0097468$; $k_3 = -0.01\sqrt{1 + 5k_2} = -0.01\sqrt{0.951266} = -0.0097533$; $k_4 = -0.01\sqrt{1 + 10k_3} = -0.01\sqrt{0.902467} = -0.0094998$. Then $h_1 = 1 + \frac{10}{6}(-0.01 - 0.0194936 - 0.0195066 - 0.0094998) = 1 + \frac{10}{6} \times (-0.0585000) = 0.9025000$, equal to the exact 0.9025 to seven decimals, with a step of ten minutes.

11.5.3 Measuring the orders

Listing 11.3 implements all four methods as step functions with the same signature, so that one `solve` function can drive any of them, and prints the error at one hour for halved steps.

Listing 11.3 `ode_orders.py`: Euler, Heun, midpoint and RK4 as interchangeable step functions, with the error ratios for halved steps.

```
"""Euler, Heun, midpoint and classical Runge-Kutta (RK4) on the draining tank
of tank_euler.py: error at t = 60 min for halved steps."""
from tank_euler import slope, exact


def euler_step(f, t, y, h):
    return y + h * f(t, y)


def heun_step(f, t, y, h):
    k1 = f(t, y)
    k2 = f(t + h, y + h * k1)           # slope at the Euler prediction
    return y + h * (k1 + k2) / 2


def midpoint_step(f, t, y, h):
    k1 = f(t, y)
    k2 = f(t + h / 2, y + h / 2 * k1)   # slope half-way along
    return y + h * k2


def rk4_step(f, t, y, h):
    k1 = f(t, y)
    k2 = f(t + h / 2, y + h / 2 * k1)
    k3 = f(t + h / 2, y + h / 2 * k2)
    k4 = f(t + h, y + h * k3)
    return y + h * (k1 + 2 * k2 + 2 * k3 + k4) / 6


def solve(step_fn, f, t0, y0, h, t_end):
    t, y = t0, y0
    for _ in range(round((t_end - t0) / h)):
        y = step_fn(f, t, y, h)
        t += h
    return y
```

Listing 11.3 (Continued) *ode_orders.py: Euler, Heun, midpoint and RK4 as interchangeable step functions, with the error ratios for halved steps.*

```
def main():
    methods = [("Euler", euler_step), ("Heun", heun_step),
               ("midpoint", midpoint_step), ("RK4", rk4_step)]
    for name, fn in methods:
        print(f"{name}:")
        old = None
        for h in (10.0, 5.0, 2.5, 1.25):
            err = abs(solve(fn, slope, 0.0, 1.0, h, 60.0) - exact(60.0))
            ratio = f"  ratio {old / err:6.2f}" if old else ""
            print(f"  step {h:5.2f}   error {err:.3e}{ratio}")
            old = err


if __name__ == "__main__":
    main()
```

In Java, **Listing 11.4** declares a small interface `Stepper` so that the four step methods can be stored in an array and passed to `solve` as method references.

Listing 11.4 `OdeOrders.java`: the same comparison in Java.

```
import java.util.Locale;
import java.util.function.DoubleBinaryOperator;

/** Euler, Heun, midpoint and classical Runge-Kutta (RK4) on the draining tank
 *  of TankEuler: error at t = 60 min for halved steps. */
public class OdeOrders {
    interface Stepper {
        double step(DoubleBinaryOperator f, double t, double y, double h);
    }

    static double eulerStep(DoubleBinaryOperator f, double t, double y, double h) {
        return y + h * f.applyAsDouble(t, y);
    }

    static double heunStep(DoubleBinaryOperator f, double t, double y, double h) {
        double k1 = f.applyAsDouble(t, y);
        double k2 = f.applyAsDouble(t + h, y + h * k1);  // at the Euler prediction
        return y + h * (k1 + k2) / 2;
    }

    static double midpointStep(DoubleBinaryOperator f, double t, double y,
                               double h) {
        double k1 = f.applyAsDouble(t, y);
        double k2 = f.applyAsDouble(t + h / 2, y + h / 2 * k1);  // half-way
        return y + h * k2;
    }

    static double rk4Step(DoubleBinaryOperator f, double t, double y, double h) {
        double k1 = f.applyAsDouble(t, y);
        double k2 = f.applyAsDouble(t + h / 2, y + h / 2 * k1);
        double k3 = f.applyAsDouble(t + h / 2, y + h / 2 * k2);
        double k4 = f.applyAsDouble(t + h, y + h * k3);
        return y + h * (k1 + 2 * k2 + 2 * k3 + k4) / 6;
    }

    static double solve(Stepper s, DoubleBinaryOperator f, double t0, double y0,
                        double h, double tEnd) {
        double t = t0, y = y0;
        long n = Math.round((tEnd - t0) / h);
        for (long i = 0; i < n; i++) {
            y = s.step(f, t, y, h);
            t += h;
        }
        return y;
    }
```

Listing 11.4 (Continued) *OdeOrders.java: the same comparison in Java.*

```
    public static void main(String[] args) {
        String[] names = {"Euler", "Heun", "midpoint", "RK4"};
        Stepper[] steppers = {OdeOrders::eulerStep, OdeOrders::heunStep,
                              OdeOrders::midpointStep, OdeOrders::rk4Step};
        for (int m = 0; m < names.length; m++) {
            System.out.println(names[m] + ":");
            double old = 0;
            for (double h : new double[] {10.0, 5.0, 2.5, 1.25}) {
                double err = Math.abs(solve(steppers[m], TankEuler::slope, 0.0,
                                            1.0, h, 60.0) - TankEuler.exact(60.0));
                String ratio = old != 0
                    ? String.format(Locale.ROOT, "  ratio %6.2f", old / err) : "";
                System.out.println(String.format(Locale.ROOT,
                    "  step %5.2f   error %.3e%s", h, err, ratio));
                old = err;
            }
        }
    }
}
```

Both print:

```
Euler:
  step 10.00   error 1.286e-02
  step  5.00   error 6.335e-03  ratio   2.03
  step  2.50   error 3.144e-03  ratio   2.01
  step  1.25   error 1.566e-03  ratio   2.01
Heun:
  step 10.00   error 3.988e-04
  step  5.00   error 9.666e-05  ratio   4.13
  step  2.50   error 2.380e-05  ratio   4.06
  step  1.25   error 5.904e-06  ratio   4.03
midpoint:
  step 10.00   error 1.932e-04
  step  5.00   error 4.759e-05  ratio   4.06
  step  2.50   error 1.181e-05  ratio   4.03
  step  1.25   error 2.941e-06  ratio   4.02
RK4:
  step 10.00   error 1.535e-07
  step  5.00   error 9.345e-09  ratio  16.42
  step  2.50   error 5.762e-10  ratio  16.22
  step  1.25   error 3.577e-11  ratio  16.11
```

The ratios tell the orders at a glance: 2 for Euler (first order), 4 for Heun and midpoint (second order), 16 for RK4 (fourth order). The absolute numbers tell the practical story. With a step of 10 minutes, six steps in the hour, RK4's error is 1.5×10^{-7} m, a sixth of a micrometer, using 24 slope evaluations. Euler with 48 steps of 1.25 minutes, 48 evaluations, is still wrong by 1.6 mm, ten thousand times worse for twice the work. That is why RK4 is the workhorse of this chapter and of much of practical computing.

11.6 Building the Morning Simulator

Now the real morning. At 06:00 the tank holds 25 cm of water left from yesterday. The pump adds 20 L/min from 06:00 until Anto switches it off. The drip valve opens at 06:30 and stays open. The tank is 1 m high; anything above that leaves through the overflow pipe. When the level falls below 1 cm the drippers effectively stop. The model is Equation (11.1) with a pump term that switches off and a drip term that switches on.

Switches need care. RK4's accuracy comes from Taylor expansions, which assume that f is smooth *within* each step. If the pump stops in the middle of a step, some of the four slopes see the pump and some do not, and the step is only first-order accurate. The cure is simple: never let a step straddle a switch. **Listing 11.5** splits any step that contains a switching time into two pieces, one up to the switch and one after it, and decides whether the pump and valve are on from the *start* of each piece. It also caps the level at the overflow and counts the spilled liters, stops the square root from seeing a negative level, records the highest level and the moment the drippers stop, and finally runs the whole morning again with half the step, as a check.

Listing 11.5 `morning.py`: one morning of the drip-irrigation tank with RK4, including the pump switch, overflow and running dry.

```
"""One morning of the orchard's drip-irrigation tank, simulated with RK4.
The pump fills the tank at 20 L/min from 06:00 until the pump is switched off;
the drip valve opens at 06:30. Usage: python morning.py [pump-off minute]"""
import math
import sys
from ode_orders import rk4_step

AREA, HEIGHT, K = 1.0, 1.0, 0.01     # m^2, m, drip constant (m^2.5/min)
PUMP, DRIP_OPEN = 0.02, 30.0         # m^3/min, minutes after 06:00
DRY = 0.01                           # m: below 1 cm the drippers stop


def make_slope(pump_on, drip_on):
    """Pump and valve stay as they are during one step."""
    def slope(t, h):
        inflow = PUMP if pump_on else 0.0
        outflow = K * math.sqrt(max(h, 0.0)) if drip_on else 0.0
        return (inflow - outflow) / AREA
    return slope


def simulate(pump_off, h0=0.25, step=1.0, t_end=300.0):
    """Returns rows (t, h), highest level, liters spilled, minute run dry."""
    t, h, top, spilled, dry_at = 0.0, h0, h0, 0.0, None
    rows = [(t, h)]
    for _ in range(round(t_end / step)):
        cuts = [c for c in (pump_off, DRIP_OPEN) if t < c < t + step]
        start = t
        for stop in sorted(cuts) + [t + step]:   # split the step at a switch
            f = make_slope(start < pump_off, start >= DRIP_OPEN)
            h = max(rk4_step(f, start, h, stop - start), 0.0)
            start = stop
        t += step
        if h > HEIGHT:                   # the overflow pipe takes the excess
            spilled += (h - HEIGHT) * AREA * 1000
            h = HEIGHT
        top = max(top, h)
        if dry_at is None and t > pump_off and h < DRY:
            dry_at = t
        rows.append((t, h))
    return rows, top, spilled, dry_at
```

Listing 11.5 (Continued) *morning.py: one morning of the drip-irrigation tank with RK4, including the pump switch, overflow and running dry.*

```
def clock(minutes):
    m = int(minutes)                # whole minutes, rounded down
    return f"{6 + m // 60:02d}:{m % 60:02d}"


def main(args):
    pump_off = float(args[0]) if args else 90.0
    rows, top, spilled, dry_at = simulate(pump_off)
    print(f"pump off at {clock(pump_off)}, drip valve open from {clock(DRIP_OPEN)}")
    print(" time   level(cm)  liters")
    for t, h in rows[::30]:
        print(f"{clock(t)}   {100 * h:8.1f}  {1000 * h * AREA:6.0f}")
    print(f"highest level {100 * top:.1f} cm, spilled {spilled:.0f} liters")
    print(f"drippers stop at {clock(dry_at)}" if dry_at else "never runs dry")
    half = simulate(pump_off, step=0.5)
    print(f"check with half the step: highest {100 * half[1]:.1f} cm,"
          f" spilled {half[2]:.0f} liters, dry at {clock(half[3])}")


if __name__ == "__main__":
    main(sys.argv[1:])
```

Listing 11.6 is the Java version. A small class `Day` carries the results that the Python version returns as a tuple, and `dryAt` is a `Double` object so that it can be `null` when the tank never runs dry.

Listing 11.6 `Morning.java`: the same simulator in Java.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Locale;
import java.util.function.DoubleBinaryOperator;

/** One morning of the orchard's drip-irrigation tank, simulated with RK4.
 *  The pump fills the tank at 20 L/min from 06:00 until the pump is switched
 *  off; the drip valve opens at 06:30. Usage: java Morning [pump-off minute] */
public class Morning {
    static final double AREA = 1.0, HEIGHT = 1.0, K = 0.01;   // m^2, m, drip
    static final double PUMP = 0.02, DRIP_OPEN = 30.0;        // m^3/min, minute
    static final double DRY = 0.01;          // m: below 1 cm the drippers stop

    /** Pump and valve stay as they are during one step. */
    static DoubleBinaryOperator makeSlope(boolean pumpOn, boolean dripOn) {
        return (t, h) -> {
            double inflow = pumpOn ? PUMP : 0.0;
            double outflow = dripOn ? K * Math.sqrt(Math.max(h, 0.0)) : 0.0;
            return (inflow - outflow) / AREA;
        };
    }

    /** Result of one simulated morning. */
    static class Day {
        List<double[]> rows = new ArrayList<>();
        double top, spilled;
        Double dryAt;                        // null when it never runs dry
    }

    static Day simulate(double pumpOff, double h0, double step, double tEnd) {
        Day d = new Day();
        double t = 0.0, h = h0;
        d.top = h0;
        d.rows.add(new double[] {t, h});
        long n = Math.round(tEnd / step);
        for (long i = 0; i < n; i++) {
            List<Double> stops = new ArrayList<>();
            for (double c : new double[] {pumpOff, DRIP_OPEN}) {
                if (t < c && c < t + step) {
                    stops.add(c);
                }
            }
            stops.sort(null);
            stops.add(t + step);
            double start = t;
```

Listing 11.6 (Continued) *Morning.java: the same simulator in Java.*

```
            for (double stop : stops) {             // split the step at a switch
                DoubleBinaryOperator f = makeSlope(start < pumpOff,
                                                   start >= DRIP_OPEN);
                h = Math.max(OdeOrders.rk4Step(f, start, h, stop - start), 0.0);
                start = stop;
            }
            t += step;
            if (h > HEIGHT) {                       // the overflow pipe takes it
                d.spilled += (h - HEIGHT) * AREA * 1000;
                h = HEIGHT;
            }
            d.top = Math.max(d.top, h);
            if (d.dryAt == null && t > pumpOff && h < DRY) {
                d.dryAt = t;
            }
            d.rows.add(new double[] {t, h});
        }
        return d;
    }

    static Day simulate(double pumpOff) {
        return simulate(pumpOff, 0.25, 1.0, 300.0);
    }

    static String clock(double minutes) {
        long m = (long) Math.floor(minutes);      // whole minutes, rounded down
        return String.format(Locale.ROOT, "%02d:%02d", 6 + m / 60, m % 60);
    }

    public static void main(String[] args) {
        double pumpOff = args.length > 0 ? Double.parseDouble(args[0]) : 90.0;
        Day d = simulate(pumpOff);
        System.out.println("pump off at " + clock(pumpOff)
                           + ", drip valve open from " + clock(DRIP_OPEN));
        System.out.println(" time   level(cm)  liters");
        for (int i = 0; i < d.rows.size(); i += 30) {
            double[] r = d.rows.get(i);
            System.out.println(String.format(Locale.ROOT, "%s   %8.1f  %6.0f",
                clock(r[0]), 100 * r[1], 1000 * r[1] * AREA));
        }
        System.out.println(String.format(Locale.ROOT,
            "highest level %.1f cm, spilled %.0f liters", 100 * d.top, d.spilled));
        System.out.println(d.dryAt != null ? "drippers stop at " + clock(d.dryAt)
                                           : "never runs dry");
        Day half = simulate(pumpOff, 0.25, 0.5, 300.0);
        System.out.println(String.format(Locale.ROOT,
```

Listing 11.6 (Continued) *Morning.java: the same simulator in Java.*

```
            "check with half the step: highest %.1f cm, spilled %.0f liters,"
            + " dry at %s", 100 * half.top, half.spilled, clock(half.dryAt)));
    }
}
```

With the pump left on until 07:30, as on a forgetful morning, both print:

```
pump off at 07:30, drip valve open from 06:30
 time   level(cm)  liters
06:00       25.0     250
06:30       85.0     850
07:00      100.0    1000
07:30      100.0    1000
08:00       72.3     723
08:30       49.0     490
09:00       30.3     303
09:30       16.0     160
10:00        6.3      63
10:30        1.0      10
11:00        0.0       0
highest level 100.0 cm, spilled 454 liters
drippers stop at 10:31
check with half the step: highest 100.0 cm, spilled 455 liters, dry at 10:30
```

The tank is full at about 06:44 and stays at the brim for three quarters of an hour while 454 liters pour out of the overflow, nearly half a tankful, together with the electricity used to pump it. After the pump stops, the level falls along the curve of Equation (11.3), and the drippers stop at about 10:30.

We can check that last number exactly. From 07:30 the pump is off and the tank starts full, so Equation (11.3) applies with $h_0 = 1$ and the clock started at 07:30: $h = (1 - 0.005\,s)^2$ where s is the time since 07:30. The level reaches 1 cm when $1 - 0.005\,s = 0.1$, that is $s = 180$ minutes: at 10:30 exactly. The simulation with one-minute steps reports 10:31, because it checks the level only at whole minutes and at 10:30 the level is a hair above 1 cm; with half-minute steps it reports 10:30. The spilled volume, 454 or 455 liters, differs slightly for the same reason: the overflow is handled at the end of each step. This is the useful habit to learn here: **run the simulation again with half the step**. If the answers you care about do not change, the step is small enough; if they do, the difference is an estimate of the error, exactly as in Chapters 9 and 10.

A reported time that jumps between 10:30 and 10:31 also shows how to do better when the *moment* of an event matters. Instead of asking only "is the level below 1 cm at this whole minute?", the program could notice the minute in which the level crosses 1 cm and then locate the crossing inside that minute, by linear interpolation between the two levels or, more precisely, by bisection on a partial RK4 step, exactly as in Chapter 2. SciPy's `solve_ivp` does this for you through its `events` argument: you give it a function that changes sign at the event, such as the level minus 0.01, and it returns the time of the crossing to full accuracy.

The command-line argument lets Anto try other plans; `python3 morning.py 45` switches the pump off at 06:45:

```
pump off at 06:45, drip valve open from 06:30
 time   level(cm)  liters
06:00       25.0     250
06:30       85.0     850
07:00       85.6     856
07:30       60.1     601
08:00       39.1     391
08:30       22.6     226
09:00       10.6     106
09:30        3.1      31
10:00        0.1       1
10:30        0.0       0
11:00        0.0       0
highest level 100.0 cm, spilled 5 liters
drippers stop at 09:46
check with half the step: highest 100.0 cm, spilled 5 liters, dry at 09:45
```

Now only 5 liters overflow, but the drippers stop at 09:46, three quarters of an hour earlier. **Figure 11.2** draws the two mornings. The planned curve of the next section lies between them.

11.7 Making It Better: Planning the Pump

Anto's real question is an inverse one: *at what time should the pump stop so that the tank fills to 98 cm, two centimeters below the overflow, but no higher?* The simulator answers the forward question, "given the pump-off time, how high does the tank get?", so we need a root-finding method around it. The function $g(T) = \text{top}(T) - 0.98$, where $\text{top}(T)$ is the highest level when the pump stops at minute T, is negative for an early pump-off and positive for a late one. That is exactly the setting of the bisection method of Chapter 2, which needs nothing but the sign of g. Each evaluation of g is a whole RK4 simulation of the morning, which takes a few milliseconds. **Listing 11.7** runs twelve bisection steps between 06:30 and 07:30.

Listing 11.7 `pump_plan.py`: bisection on the pump-off time, with the RK4 simulation as the function whose root is sought.

```
"""When should the pump be switched off so that the tank fills to 98 cm but
never overflows? Bisection (Chapter 2) around the RK4 simulation."""
from morning import simulate, clock

TARGET = 0.98                          # m: leave 2 cm below the overflow


def top_level(pump_off):
    return simulate(pump_off)[1]


def main():
    lo, hi = 30.0, 90.0                # 06:30 is too early, 07:30 too late
    print(" step   pump off   highest level(cm)")
    for i in range(1, 13):
        mid = (lo + hi) / 2
        top = top_level(mid)
        print(f"{i:5d}   {mid:8.3f}   {100 * top:10.2f}")
        if top > TARGET:
            hi = mid
        else:
            lo = mid
    best = (lo + hi) / 2
    rows, top, spilled, dry_at = simulate(best)
    print(f"switch the pump off after {best:.1f} min, at about {clock(best)}:"
          f" top {100 * top:.1f} cm, spilled {spilled:.0f} liters,"
          f" drippers stop at {clock(dry_at)}")


if __name__ == "__main__":
    main()
```

Listing 11.8 `PumpPlan.java`: the same planner in Java.

```
import java.util.Locale;

/** When should the pump be switched off so that the tank fills to 98 cm but
 *  never overflows? Bisection (Chapter 2) around the RK4 simulation. */
public class PumpPlan {
    static final double TARGET = 0.98;       // m: leave 2 cm below the overflow

    public static void main(String[] args) {
        double lo = 30.0, hi = 90.0;         // 06:30 is too early, 07:30 too late
        System.out.println(" step   pump off   highest level(cm)");
        for (int i = 1; i <= 12; i++) {
            double mid = (lo + hi) / 2;
            double top = Morning.simulate(mid).top;
            System.out.println(String.format(Locale.ROOT, "%5d   %8.3f   %10.2f",
                i, mid, 100 * top));
            if (top > TARGET) {
                hi = mid;
            } else {
                lo = mid;
            }
        }
        double best = (lo + hi) / 2;
        Morning.Day d = Morning.simulate(best);
        System.out.println(String.format(Locale.ROOT,
            "switch the pump off after %.1f min, at about %s: top %.1f cm,"
            + " spilled %.0f liters, drippers stop at %s", best,
            Morning.clock(best), 100 * d.top, d.spilled, Morning.clock(d.dryAt)));
    }
}
```

Listing 11.8 prints the same search:

```
 step   pump off   highest level(cm)
    1     60.000        100.00
    2     45.000        100.00
    3     37.500         92.44
    4     41.250         96.52
    5     43.125         98.54
    6     42.188         97.53
    7     42.656         97.86
    8     42.891         98.33
    9     42.773         98.09
   10     42.715         97.97
   11     42.744         98.03
   12     42.729         98.00
switch the pump off after 42.7 min, at about 06:42: top 98.0 cm, spilled 0 liters, drippers
stop at 09:41
```

After twelve halvings the interval is $60/2^{12} \approx 0.015$ minutes wide, far finer than anyone switches a pump. The answer is to switch the pump off 42.7 minutes after six, at about 06:42. The tank then peaks at 98 cm, nothing overflows, and the drippers run until 09:41. Splitting the step exactly at the switching time, as `simulate` does, matters here: without it, every pump-off time between two whole minutes would behave like the next whole minute, the function g would move in jumps, and the bisection would converge to a step boundary instead of the true answer.

The planner also exposes a trade-off that Anto had only felt. With a 20 L/min pump, stopping at 06:42 avoids waste but ends the watering at 09:41; running the pump longer lets the tank last longer but spills water. The way out is a pump that delivers less water for longer, for instance by partly closing a valve on the pump outlet. Exercise 11.4 finds the flow that fills the tank to 98 cm with the pump running until 07:30, and shows that the drippers then run until almost 10:30 with no overflow at all. **Figure 11.2** shows the planned morning against the forgetful one.

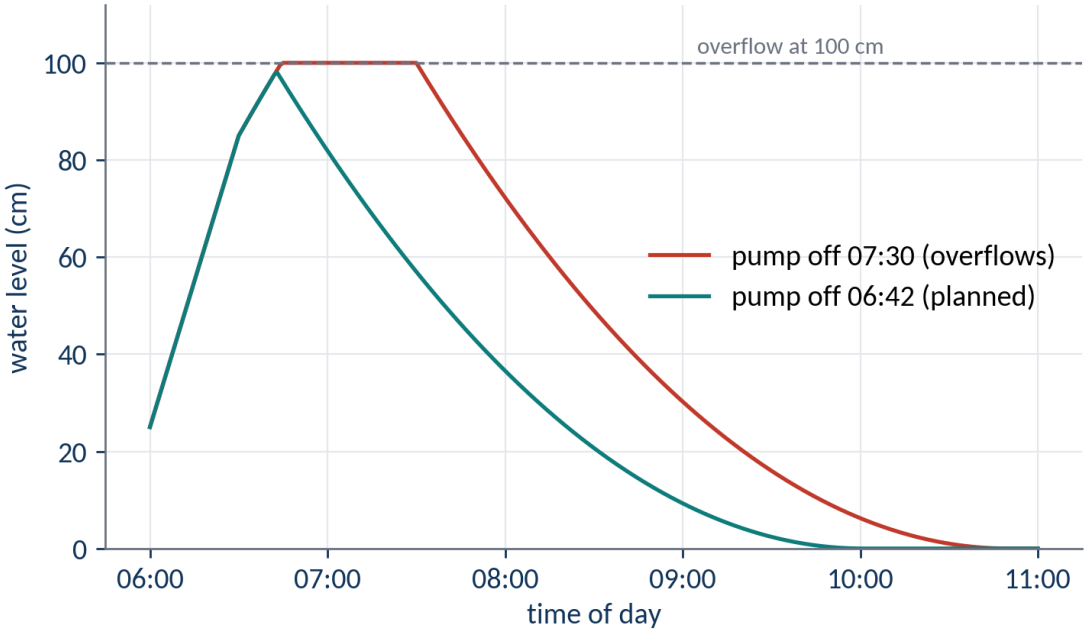


Figure 11.2 Water level on two mornings: with the pump left on until 07:30 the tank overflows for three quarters of an hour; with the pump switched off at 06:42, as planned by bisection, it peaks at 98 cm and never overflows.

11.8 Systems and Stability: The Motorbike Seat

11.8.1 From one equation to a system

Many problems involve several quantities that change together, or a second derivative. Think of Mas Dimas's motorbike from Chapter 9 riding over a speed bump. The rear suspension is a spring with a shock absorber (a damper). If $x(t)$ is how far the seat is pushed down from its resting position, Newton's second law gives

$$mx'' + cx' + kx = 0,$$

where m is the mass carried by the suspension (about 100 kg of bike and rider), k the spring stiffness (about 20,000 N/m) and c the damping of the shock absorber: about 1,000 N·s/m for a good one, much less for a worn one, say 150 N·s/m. Every numerical method of this chapter solves *first-order* equations. The trick is to introduce the speed $v = x'$ as a second unknown:

$$x' = v, \qquad v' = -\frac{cv + kx}{m}. \tag{11.8}$$

That is a **system** of two first-order equations for the vector $y = (x, v)$, of exactly the form Equation (11.2), with f now returning a vector. Euler and RK4 apply unchanged, component by component; every k_i in Equation (11.7) becomes a vector. Any higher-order equation can be turned into a first-order system the same way.

Listing 11.9 simulates two seconds after the bump, starting with the seat 5 cm down and at rest, for both shock absorbers, with Euler and with RK4 at a step of 0.01 s, and with RK4 at 0.001 s as a reference.

Listing 11.9 `bump.py`: the seat after a speed bump as a system of two equations, with Euler and RK4.

```
"""A motorbike seat after a speed bump: m x'' + c x' + k x = 0 as a system of
two first-order equations, solved with Euler and with RK4."""

MASS, SPRING = 100.0, 20000.0        # kg, N/m
SHOCKS = [("good shock", 1000.0), ("worn shock", 150.0)]   # c in N s/m


def make_rhs(damping):
    def rhs(t, y):
        x, v = y
        return [v, -(damping * v + SPRING * x) / MASS]
    return rhs


def euler_step(f, t, y, h):
    k1 = f(t, y)
    return [y[i] + h * k1[i] for i in range(len(y))]


def rk4_step(f, t, y, h):
    k1 = f(t, y)
    k2 = f(t + h / 2, [y[i] + h / 2 * k1[i] for i in range(len(y))])
    k3 = f(t + h / 2, [y[i] + h / 2 * k2[i] for i in range(len(y))])
    k4 = f(t + h, [y[i] + h * k3[i] for i in range(len(y))])
    return [y[i] + h * (k1[i] + 2 * k2[i] + 2 * k3[i] + k4[i]) / 6
            for i in range(len(y))]


def trace(step_fn, f, h, t_end=2.0, every=0.5):
    """Seat position (mm) at t = every, 2*every, ... up to t_end."""
    y, out = [0.05, 0.0], []         # 5 cm down, at rest, just after the bump
    per = round(every / h)
    for n in range(1, round(t_end / h) + 1):
        y = step_fn(f, (n - 1) * h, y, h)
        if n % per == 0:
            out.append(1000 * y[0])
    return out
```

Listing 11.9 (Continued) *bump.py: the seat after a speed bump as a system of two equations, with Euler and RK4.*

```
def main():
    print("seat position (mm) at t =        0.5 s     1.0 s     1.5 s     2.0 s")
    for name, c in SHOCKS:
        f = make_rhs(c)
        for label, fn, h in (("Euler h=0.01 ", euler_step, 0.01),
                             ("RK4   h=0.01 ", rk4_step, 0.01),
                             ("RK4   h=0.001", rk4_step, 0.001)):
            cells = "".join(f"{x:10.3f}" for x in trace(fn, f, h))
            print(f"{name}, {label}    {cells}")


if __name__ == "__main__":
    main()
```

Listing 11.10 `Bump.java`: the same system in Java, with the vector operations written out in a helper `axpy`.

```java
import java.util.Locale;

/** A motorbike seat after a speed bump: m x'' + c x' + k x = 0 as a system of
 *  two first-order equations, solved with Euler and with RK4. */
public class Bump {
    static final double MASS = 100.0, SPRING = 20000.0;      // kg, N/m
    static final String[] NAMES = {"good shock", "worn shock"};
    static final double[] DAMPING = {1000.0, 150.0};         // c in N s/m

    interface Rhs {
        double[] f(double t, double[] y);
    }

    interface Stepper {
        double[] step(Rhs f, double t, double[] y, double h);
    }

    static Rhs makeRhs(double damping) {
        return (t, y) -> new double[] {y[1],
                                       -(damping * y[1] + SPRING * y[0]) / MASS};
    }

    /** y + a * k, element by element. */
    static double[] axpy(double[] y, double a, double[] k) {
        double[] r = new double[y.length];
        for (int i = 0; i < y.length; i++) {
            r[i] = y[i] + a * k[i];
        }
        return r;
    }

    static double[] eulerStep(Rhs f, double t, double[] y, double h) {
        return axpy(y, h, f.f(t, y));
    }

    static double[] rk4Step(Rhs f, double t, double[] y, double h) {
        double[] k1 = f.f(t, y);
        double[] k2 = f.f(t + h / 2, axpy(y, h / 2, k1));
        double[] k3 = f.f(t + h / 2, axpy(y, h / 2, k2));
        double[] k4 = f.f(t + h, axpy(y, h, k3));
        double[] r = new double[y.length];
        for (int i = 0; i < y.length; i++) {
            r[i] = y[i] + h * (k1[i] + 2 * k2[i] + 2 * k3[i] + k4[i]) / 6;
        }
        return r;
    }
```

Listing 11.10 (Continued) *Bump.java: the same system in Java, with the vector operations written out in a helper axpy.*

```
    /** Seat position (mm) at t = 0.5, 1.0, 1.5 and 2.0 s. */
    static double[] trace(Stepper s, Rhs f, double h) {
        double[] y = {0.05, 0.0};          // 5 cm down, at rest, after the bump
        double[] out = new double[4];
        long per = Math.round(0.5 / h), total = Math.round(2.0 / h);
        for (long n = 1; n <= total; n++) {
            y = s.step(f, (n - 1) * h, y, h);
            if (n % per == 0) {
                out[(int) (n / per) - 1] = 1000 * y[0];
            }
        }
        return out;
    }

    public static void main(String[] args) {
        System.out.println(
            "seat position (mm) at t =        0.5 s     1.0 s     1.5 s     2.0 s");
        String[] labels = {"Euler h=0.01 ", "RK4   h=0.01 ", "RK4   h=0.001"};
        Stepper[] steppers = {Bump::eulerStep, Bump::rk4Step, Bump::rk4Step};
        double[] steps = {0.01, 0.01, 0.001};
        for (int s = 0; s < NAMES.length; s++) {
            Rhs f = makeRhs(DAMPING[s]);
            for (int m = 0; m < labels.length; m++) {
                StringBuilder cells = new StringBuilder();
                for (double x : trace(steppers[m], f, steps[m])) {
                    cells.append(String.format(Locale.ROOT, "%10.3f", x));
                }
                System.out.println(NAMES[s] + ", " + labels[m] + "    " + cells);
            }
        }
    }
}
```

Listing 11.10 and the Python program print:

```
seat position (mm) at t =        0.5 s     1.0 s     1.5 s     2.0 s
good shock, Euler h=0.01          6.401     0.509     0.003    -0.007
good shock, RK4   h=0.01          4.386     0.344     0.024     0.001
good shock, RK4   h=0.001         4.386     0.344     0.024     0.001
worn shock, Euler h=0.01         42.236     3.606   -48.413   -82.305
worn shock, RK4   h=0.01         25.762     1.608   -10.594   -11.134
worn shock, RK4   h=0.001        25.761     1.607   -10.595   -11.134
```

With the good shock absorber all methods agree roughly, and the seat is still within half a millimeter of rest after one second. With the worn one, RK4 at both steps agrees to a thousandth of a millimeter: the seat is still swinging by about a centimeter after two seconds, which a rider feels as a bouncy, floating ride. Euler says something quite different: the swing *grows*, to more than 8 cm after two seconds, larger than the 5 cm the bump caused. A real spring with a real damper cannot do that; energy is only ever taken out. **Figure 11.3** shows Euler's swing growing while the true one dies away.

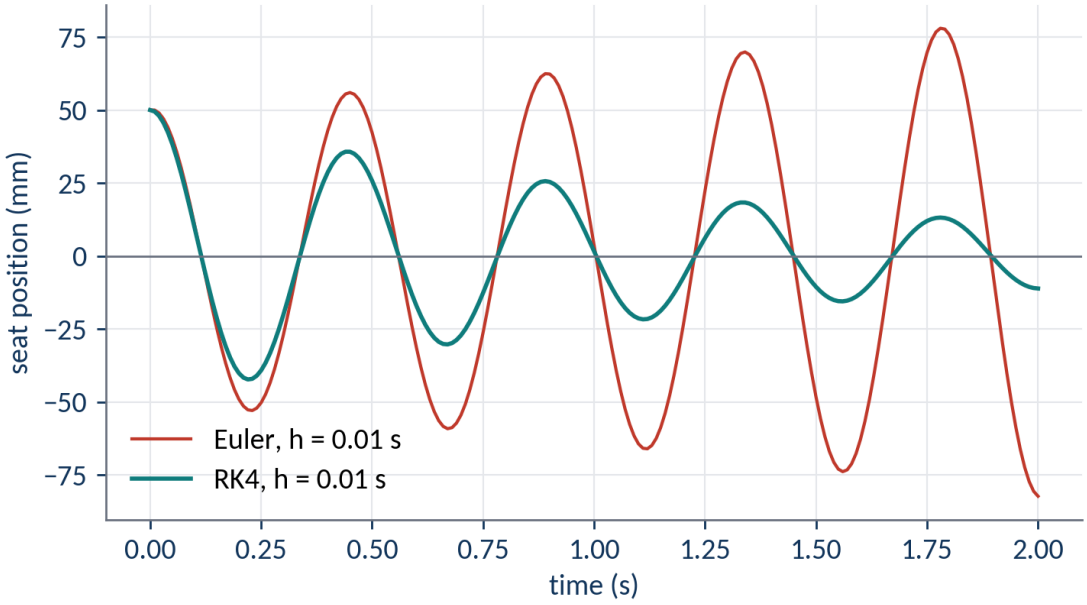


Figure 11.3 The seat with a worn shock absorber after a speed bump: RK4 shows the swing dying away, while Euler's method with the same step makes it grow, an instability of the method, not of the motorbike.

11.8.2 Why Euler blew up: stability

To understand this, apply Euler's method to the simplest equation that decays, the **test equation** $y' = \lambda y$ with a constant λ whose real part is negative. The exact solution $y_0 e^{\lambda t}$ decays to zero. Euler gives $y_{n+1} = y_n + h\lambda y_n = (1 + h\lambda)y_n$, so $y_n = (1 + h\lambda)^n y_0$, which decays only if

$$|1 + h\lambda| \leq 1. \tag{11.9}$$

For real negative λ this means \$h /\$: a step too large makes the numerical solution oscillate and grow, however small the local error estimate suggested. The set of values $h\lambda$ satisfying Equation (11.9) is the **region of absolute stability** of Euler's method: a disc of radius 1 centered at -1 in the complex plane [7].

For the suspension, the system Equation (11.8) behaves like the test equation with the two complex values $\lambda = -\zeta\omega \pm i\,\omega\sqrt{1-\zeta^2}$, where $\omega = \sqrt{k/m}$ is the natural angular frequency and $\zeta = c/\left(2\sqrt{km}\right)$ the damping ratio. Here $\omega = \sqrt{200} = 14.14$ rad/s (the seat bounces about 2.25 times a second), $\zeta = 0.354$ for the good shock and $\zeta = 0.053$ for the worn one. Then

$$|\,1 + h\lambda\,|^2 = (1 - h\zeta\omega)^2 + h^2\omega^2(1 - \zeta^2) = 1 - 2h\zeta\omega + h^2\omega^2,$$

which is at most 1 only if $h \leq 2\zeta/\omega$. For the good shock that is $h \leq 0.05$ s, and our step of 0.01 s is stable, though not accurate. For the worn shock it is $h \leq 0.0075$ s, and 0.01 s is *outside* the stability region: every step multiplies the swing by $\sqrt{1 - 2 \times 0.01 \times 0.75 + 0.02} = \sqrt{1.005} \approx 1.0025$, and 200 steps multiply it by about 1.65. Lightly damped oscillations are the worst case for Euler: the less damping the system has, the smaller the step must be. RK4's stability region is much larger and reaches along the imaginary axis to about $|h\lambda| \leq 2.8$; with $h\omega = 0.14$ it is comfortably inside.

> **WARNING:** An unstable method does not announce itself with an error message; it produces smooth, plausible-looking numbers that are wrong. A simulated suspension that bounces more and more, a simulated tank level that oscillates, a population that goes negative: when a

> simulation does something the physics cannot, suspect the step size before you suspect the physics, and halve the step to see whether the behavior survives.

Some problems contain both very fast and very slow processes, such as a chemical reaction with rates that differ by a factor of a million, or an electrical circuit with a tiny capacitor. Stability then forces an explicit method like Euler or RK4 to take tiny steps for the fast process long after it has died out. Such problems are called **stiff**, and they are solved with *implicit* methods, such as the backward Euler method $y_{n+1} = y_n + hf(t_{n+1}, y_{n+1})$, which require solving an equation (Chapters 2–5) at every step but are stable for any step on decaying problems [7]. SciPy's `solve_ivp` offers them as `method="Radau"`, `"BDF"` and `"LSODA"`.

11.9 Checking Against the Library

SciPy's `scipy.integrate.solve_ivp` is the standard Python tool for initial value problems. The current release, SciPy 1.18.1, came out on August 21, 2026 [8]. Its default method, `"RK45"`, is the Dormand–Prince pair [9]: a fifth-order and a fourth-order Runge–Kutta formula that share their slope evaluations. The difference between the two results estimates the error of each step, and the step is enlarged or reduced automatically to keep that estimate below `rtol` times the solution plus `atol`. It is the idea of our step-halving check, applied at every step and at almost no extra cost. `"DOP853"` is an eighth-order method from the same family for high accuracy [4], [10]. **Listing 11.11** solves the draining tank and the worn-shock seat with it.

Listing 11.11 `compare_scipy.py`: the tank and the seat with `solve_ivp`.

```
"""Check this chapter's answers with scipy.integrate.solve_ivp."""
import numpy as np
import scipy
from scipy.integrate import solve_ivp
from tank_euler import slope, exact
from bump import make_rhs

print(f"NumPy {np.__version__}, SciPy {scipy.__version__}")
for rtol in (1e-3, 1e-6, 1e-9):                       # RK45 is the default
    sol = solve_ivp(lambda t, y: [slope(t, y[0])], (0, 60), [1.0],
                    rtol=rtol, atol=1e-12)
    err = abs(sol.y[0, -1] - exact(60))
    print(f"tank, RK45 rtol={rtol:.0e}: h(60) = {sol.y[0, -1]:.9f},"
          f" error {err:.1e}, {sol.nfev} slope evaluations")
sol = solve_ivp(make_rhs(150.0), (0, 2), [0.05, 0.0], method="DOP853",
                rtol=1e-10, atol=1e-12, t_eval=[0.5, 1.0, 1.5, 2.0])
cells = "  ".join(f"{1000 * x:.3f}" for x in sol.y[0])
print(f"worn shock, DOP853, seat position (mm): {cells}")
```

```
NumPy 2.4.4, SciPy 1.17.1
tank, RK45 rtol=1e-03: h(60) = 0.490000035, error 3.5e-08, 26 slope evaluations
tank, RK45 rtol=1e-06: h(60) = 0.490000020, error 2.0e-08, 38 slope evaluations
tank, RK45 rtol=1e-09: h(60) = 0.490000000, error 5.7e-11, 68 slope evaluations
worn shock, DOP853, seat position (mm): 25.761  1.607  -10.595  -11.134
```

The adaptive solver reaches the exact level of Equation (11.3) at one hour to 3.5×10^{-8} m with 26 slope evaluations at the loosest tolerance, and to 6×10^{-11} m with 68 evaluations at the tightest. The seat positions from `DOP853` agree with our RK4 reference to the last printed digit (25.761, 1.607, −10.595, −11.134 mm). Note the small wrapper around `slope`: `solve_ivp` passes the state as a NumPy array even for a single equation, and expects an array-like slope in return.

The principle behind the adaptive step is worth stating as a formula, because it works for any method of order p, including our own. If y_h and $y_{h/2}$ are the results at the same time with step h and with step $h/2$, then

$$y(t) - y_{h/2} \approx \frac{y_{h/2} - y_h}{2^p - 1}, \tag{11.10}$$

the Richardson estimate of Chapters 9 and 10 once again. For RK4 on the tank at one hour, $y_{10} = 0.4900001535$ and $y_5 = 0.4900000093$, so the estimated error of y_5 is $(0.4900000093 - 0.4900001535)/15 = -9.6 \times 10^{-9}$, against the actual -9.3×10^{-9}.

The code folder also has tests (`test_ode.py`, `OdeTest.java`) that check facts that must hold: RK4 is exact for $y' = 3t^2$ (it reduces to Simpson's rule, exact for cubics); the three methods show their orders on the tank; a seat at rest stays at rest; and with the valve still closed, the tank level rises by exactly the pumped volume.

11.10 Numerical Methods in Practice: Profitina

Differential equations appear in Profitina's world wherever a quantity changes at a rate that depends on the quantity itself. The PROFIT utility coin is designed with a decaying emission: the rate at which new coins are released falls over time, and some coins are burned. A decaying rate is the solution of an equation like $E' = -\lambda E$, and the total supply then obeys $S' = E(t) - B(t)$, emission minus burning. When the schedule is a clean formula, the supply can be computed exactly; when it is combined with real, irregular burns, it is computed by stepping forward in time exactly as the tank was, and the tank's lessons carry over: switch events must not be straddled by a step, and a projection should be checked by halving the step.

A second, less obvious example is smoothing. Many dashboards, including trend displays such as Profitina's weekly score history and a price oracle like the PROFIT↔IDR rate, show a smoothed value that follows the raw measurements with some lag. The common "exponential moving average", $y_{n+1} = y_n + \alpha(x_n - y_n)$, is precisely Euler's method with step h applied to $y' = (x(t) - y)/\tau$ with $\alpha = h/\tau$, a first-order lag. Seen that way, its behavior is no mystery: with α between 0 and 1 it is stable and smooth, and with α above 2 it would oscillate and blow up, which is Equation (11.9) with $\lambda = -1/\tau$. How Profitina computes its own figures is its business; the shape of the mathematics is this chapter's, and the numbers advise while the user decides.

11.11 Using It for Real

- **Write the balance first.** Most practical ODEs are balances: rate of change of a stock equals what comes in minus what goes out. Water in a tank, money in an account, stock in a warehouse, charge in a battery, people in a queue. Write the balance in words, then in symbols, as in Equation (11.1). Check the units of every term: if the left side is cubic meters per minute, every term on the right must be too, and a term that is not is a mistake you have found before running anything.
- **Measure the constants the practical way.** Anto did not compute k from dripper geometry; he timed how fast a full tank drains. One or two such measurements fix the constants of a simple model better than a handbook.
- **Find an exact special case.** Before trusting a simulation, find a situation you can solve by hand (pump off, no leak, constant rate) and check the program against it, as Equation (11.3) checked ours.

- **Use RK4 with a fixed step for simple, smooth problems, and `solve_ivp` for everything else.** Always check by halving the step or tightening the tolerance.
- **Split steps at events.** A pump switching, a valve opening, a price changing: make the step end exactly there.
- **Adapt the tank to your own situation.** A rooftop water tank (*tandon*) during a PDAM water cut is the same model with the pump off and a different k for the household taps; a rice field drained through a sluice gate is the same model with a gate constant; a phone battery draining under a constant load is the simplest case of all, a straight line. Change the constants in `morning.py`, check them against one real observation, and plan.
- **Keep people safe.** A simulation of a vehicle, a structure or a medicine dose is a teaching tool, not a certificate. A suspension that feels floaty, or a model that says it is, is a reason to see a mechanic.

11.12 Common Mistakes, and How to Avoid Them

- **Trusting Euler because the steps are small.** First order means that accuracy is expensive, and for oscillations Euler can be unstable at steps that look small (Section 11.8.2). Use RK4 or an adaptive solver.
- **Straddling a switch.** A step that contains a sudden change in f loses the method's order. Split the step at the switching time, or stop and restart the solver there.
- **Forgetting that the step and the solution share a letter.** In textbooks h is the step; in our tank it was also the level. In code, give them different names.
- **Letting the square root see a negative number.** Rounding can push a level slightly below zero. Clamp it, as `max(h, 0.0)` does, or stop the simulation when the tank is empty.
- **Checking only the last number.** A simulation can end at a plausible number after doing something impossible on the way (an overflowing tank that never spills, a seat that bounces higher). Print or plot the whole trajectory.
- **Not checking with half the step.** It is the cheapest test there is. If halving the step changes the answer you care about, the step was too large.
- **Using `odeint` in new code.** It still works, but `solve_ivp` is the current interface, with a choice of explicit and implicit methods, event detection and dense output.

11.13 Chapter Summary

- An ordinary differential equation with a starting value, an initial value problem (Equation (11.2)), gives the rate of change of a quantity as a function of time and the quantity itself. Anto's tank obeys the mass balance Equation (11.1), and with the pump off it has the exact solution Equation (11.3).
- Euler's method (Equation (11.4)) follows the tangent for one step; its local error is $O(h^2)$ and its global error $O(h)$ (Equation (11.5)).
- Runge–Kutta methods sample the slope several times within each step. Two-stage methods satisfying the conditions derived from Equation (11.6), such as Heun and midpoint, are second order; the classical RK4 (Equation (11.7)) is fourth order and reduces to Simpson's rule when f depends only on t. The measured error ratios, 2, 4 and 16, confirmed the orders.
- The morning simulator split its steps at the pump and valve switches, handled overflow and running dry, and was checked against the exact draining time (10:30) and by halving the step.

Bisection around the simulation found the pump-off time, 06:42, that fills the tank to 98 cm without overflow.

- A second-order equation becomes a first-order system (Equation (11.8)). On the lightly damped suspension, Euler's method with $h = 0.01$ s was unstable, because $h\lambda$ lay outside its stability region (Equation (11.9)); RK4 was accurate.
- Halving the step estimates the error of any method of order p (Equation (11.10)); `solve_ivp` does this automatically with embedded Runge–Kutta pairs and offers implicit methods for stiff problems.

11.14 Exercises

1. By hand, take two steps of Heun's method with a step of 10 minutes for the draining tank from $h = 1$, and compare with the exact level at 20 minutes and with Euler's two steps from Section 11.4.
2. A glass of hot tea at 90 °C cools in a room at 28 °C following Newton's law of cooling, $T' = -0.1\,(T - 28)$ with t in minutes. Compute three Euler steps with step sizes of 5, 15 and 25 minutes. Which step sizes give a physically sensible answer, which give an oscillating one, and which blow up? Explain with Equation (11.9).
3. Show that when f depends only on t, one RK4 step Equation (11.7) is Simpson's rule on $[t_n, t_n + h]$. Why does this mean that RK4 is exact for $y' = 3t^2$ with any step?
4. Instead of stopping the pump early, Anto can partly close a valve on the pump outlet so that the pump delivers less water and runs from 06:00 to 07:30. Which flow, in liters per minute, fills the tank to exactly 98 cm? When do the drippers stop then? (Modify `PUMP` in `morning.py` and use bisection on it.)
5. Use Equation (11.10) with the RK4 results for the draining tank at one hour with steps of 10 and 5 minutes (from the output of Section 11.5.3 or by running the program) to estimate the error of the 5-minute result, and compare with the true error.
6. The *semi-implicit* (or symplectic) Euler method for the seat updates the speed first and then moves with the *new* speed: $v_{n+1} = v_n + h\,a(x_n, v_n)$, $x_{n+1} = x_n + h\,v_{n+1}$. Write it in Python and Java for the worn shock with $h = 0.01$ s and compare with the RK4 reference. Is it stable?

11.15 Solutions to the Exercises

1. First step: $k_1 = -0.01$, $k_2 = -0.01\sqrt{1 - 0.1} = -0.0094868$, $h_1 = 1 + 5 \times (-0.0194868) = 0.9025658$. Second step: $k_1 = -0.01\sqrt{0.9025658} = -0.0095004$; the trial Euler value is $0.9025658 - 0.0950040 = 0.8075618$; $k_2 = -0.01\sqrt{0.8075618} = -0.0089864$; $h_2 = 0.9025658 + 5 \times (-0.0184868) = 0.8101319$. The exact level is 0.81, so Heun is 0.13 mm high, while Euler's two steps gave 0.805132, 4.9 mm low: for twice the slope evaluations, about 37 times more accurate.

2. Write $u = T - 28$, so $u' = -0.1\,u$ with $u_0 = 62$, and Euler gives $u_{n+1} = (1 - 0.1\,h)\,u_n$. With $h = 5$: the factor is 0.5, and T goes 90,59,43.5,35.75, a sensible cooling curve (the exact values are 65.6,50.8,41.8). With $h = 15$: the factor is -0.5, and T goes $90, -3, 43.5, 20.25$: the numbers still approach 28 but jump to either side of it, including a glass of tea at -3 °C, which is physical nonsense. With $h = 25$: the factor is -1.5, and T goes $90, -65, 167.5, -181.25$: unstable. By Equation (11.9) with $\lambda = -0.1$, Euler is stable for $h \le 2/0.1 = 20$ minutes and free of oscillation only for $h \le 10$ minutes, where the factor stays non-negative.

3. If $f = f(t)$, then $k_1 = f(t_n)$, $k_2 = k_3 = f\left(t_n + \frac{h}{2}\right)$ because the second argument is ignored, and $k_4 = f(t_n + h)$. So $y_{n+1} - y_n = \frac{h}{6}\left(f(t_n) + 4f\left(t_n + \frac{h}{2}\right) + f(t_n + h)\right)$, which is Simpson's rule on one panel of width h with midpoint $t_n + \frac{h}{2}$. The exact change is $\int_{t_n}^{t_n+h} f\,(t)\,dt$. Simpson's rule is exact for polynomials up to degree three (Chapter 10), and $3t^2$ is of degree two, so every RK4 step is exact and so is the whole solution, $y = t^3$, whatever the step. The test in `test_ode.py` checks this with a step of 0.5.

4. With `PUMP` as the unknown, bisection between 0 and 20 L/min (0 and 0.02 m^3/min) converges to about 14.2 L/min. The tank then rises more slowly, is at 98 cm when the pump stops at 07:30, never overflows, and the drippers stop at about 10:28, three quarters of an hour later than with the planned full-flow pumping of Section 11.7, and with no water wasted. The same water, delivered more slowly, lasts longer.

5. $y_{10} = 0.4900001535$ and $y_5 = 0.4900000093$. By Equation (11.10) with $p = 4$, $y(60) - y_5 \approx (0.4900000093 - 0.4900001535)/15 = -9.6 \times 10^{-9}$, meaning y_5 is about 9.6×10^{-9} too high. The true error is $y_5 - 0.49 = 9.3 \times 10^{-9}$. The estimate is excellent, because at these steps RK4 is well inside its asymptotic regime (the ratios in Section 11.5.3 are close to 16).

6. **Listing 11.12** and **Listing 11.13** implement the method as a step function that plugs into `trace` from `bump.py`.

Listing 11.12 `semi_implicit.py`: semi-implicit Euler for the worn shock.

```
"""Exercise 11.6: semi-implicit (symplectic) Euler for the worn shock: update
the speed first, then move with the new speed."""
from bump import MASS, SPRING, make_rhs, rk4_step, trace


def semi_implicit_step(damping):
    def step(f, t, y, h):
        x, v = y
        v = v + h * (-(damping * v + SPRING * x) / MASS)
        return [x + h * v, v]            # uses the speed just computed
    return step


def main():
    c = 150.0
    print("seat position (mm) at t =  "
          "         0.5 s     1.0 s     1.5 s     2.0 s")
    runs = (("semi-implicit Euler h=0.01", semi_implicit_step(c), 0.01),
            ("RK4 h=0.001 (reference)   ", rk4_step, 0.001))
    for label, fn, h in runs:
        cells = "".join(f"{x:10.3f}" for x in trace(fn, make_rhs(c), h))
        print(f"{label}{cells}")


if __name__ == "__main__":
    main()
```

Listing 11.13 `SemiImplicit.java`: the same in Java.

```java
import java.util.Locale;

/** Exercise 11.6: semi-implicit (symplectic) Euler for the worn shock: update
 *  the speed first, then move with the new speed. */
public class SemiImplicit {
    static Bump.Stepper semiImplicitStep(double damping) {
        return (f, t, y, h) -> {
            double x = y[0], v = y[1];
            v = v + h * (-(damping * v + Bump.SPRING * x) / Bump.MASS);
            return new double[] {x + h * v, v};   // uses the speed just computed
        };
    }

    public static void main(String[] args) {
        double c = 150.0;
        System.out.println("seat position (mm) at t =  "
                           + "         0.5 s     1.0 s     1.5 s     2.0 s");
        String[] labels = {"semi-implicit Euler h=0.01",
                           "RK4 h=0.001 (reference)   "};
        Bump.Stepper[] steppers = {semiImplicitStep(c), Bump::rk4Step};
        double[] steps = {0.01, 0.001};
        for (int m = 0; m < labels.length; m++) {
            StringBuilder cells = new StringBuilder();
            for (double x : Bump.trace(steppers[m], Bump.makeRhs(c), steps[m])) {
                cells.append(String.format(Locale.ROOT, "%10.3f", x));
            }
            System.out.println(labels[m] + cells);
        }
    }
}
```

Both print:

```
seat position (mm) at t =           0.5 s     1.0 s     1.5 s     2.0 s
semi-implicit Euler h=0.01    23.171    -1.598   -12.393   -10.955
RK4 h=0.001 (reference)       25.761     1.607   -10.595   -11.134
```

The method is stable: the swing stays at the right size and dies away slowly, as the physics says, although the positions are off by a few millimeters because the phase of the oscillation drifts. Using the new speed to move couples the two updates so that the method no longer adds energy to an oscillation; its stability limit for an undamped oscillator is $h\omega \le 2$, far beyond our $h\omega = 0.14$. Methods with this property, called symplectic, are the standard choice in physics engines and molecular simulations, where oscillations must run for millions of steps without growing.

References

[1] L. Euler, *Institutionum Calculi Integralis*, vol. 1. St. Petersburg, Russia: Imperial Academy of Sciences, 1768.

[2] S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers*, 8th ed. New York, NY, USA: McGraw-Hill Education, 2021.

[3] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical Analysis*, 10th ed. Boston, MA, USA: Cengage Learning, 2016.

[4] E. Hairer, S. P. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems*, 2nd rev. ed. Berlin, Germany: Springer, 1993.

[5] W. Kutta, “Beitrag zur näherungsweisen Integration totaler Differentialgleichungen,” *Zeitschrift für Mathematik und Physik*, vol. 46, pp. 435–453, 1901.

[6] J. C. Butcher, *Numerical Methods for Ordinary Differential Equations*, 3rd ed. Chichester, UK: Wiley, 2016.

[7] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, 2nd rev. ed. Berlin, Germany: Springer, 1996.

[8] SciPy Developers, "scipy 1.18.1," Python Package Index, Aug. 21, 2026. [Online]. Available: https://pypi.org/project/scipy/

[9] J. R. Dormand and P. J. Prince, "A family of embedded Runge-Kutta formulae," *Journal of Computational and Applied Mathematics*, vol. 6, no. 1, pp. 19–26, 1980.

[10] SciPy Developers, "scipy.integrate.solve_ivp," SciPy v1.18 Manual, accessed Sep. 2026. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html

Chapter 12

Eigenvalues: Power Method and Its Uses (e.g. PageRank)

Some questions about a system are not about any particular moment but about what the system does *in the long run* or *by its nature*: which share of customers each shop ends up with, however the month started; at which frequencies a building likes to sway; which web pages a random reader lands on most often. Each of these is answered by an eigenvalue and its eigenvector, and all three can be computed by the same humble idea: multiply by the matrix again and again. By the end of this chapter you will be able to explain what eigenvalues and eigenvectors are and compute them by hand for small matrices; model customers switching between shops as a Markov chain and find the long-run shares; prove why the power method converges and how fast, and speed it up with the Rayleigh quotient; find the smallest and any chosen eigenvalue with inverse and shifted inverse iteration, reusing the LU decomposition of Chapter 6; compute the natural frequencies and mode shapes of a three-story shophouse; compute PageRank for a small network of websites, including pages without links; remove a found eigenvalue by deflation; bound eigenvalues with Gershgorin's theorem; and check every result against NumPy.

12.1 Three Laundries in One Kampung

In a kampung in Malang, three laundries compete for the same few hundred households: Laundry Bersih by the mosque, Laundry Kilat on the main road, and Laundry Wangi, the newest, near the school. Bu Endang, who owns Bersih, has noticed that customers are not loyal forever. Some try Kilat because it is on their way to work; some go to Wangi because a neighbor praised its fragrance; some come back. She wonders what this coming and going will do to her business. If it continues as it is, will Bersih keep its share, grow, or slowly lose customers to the newcomer?

A student from the kampung who helps at Bersih on weekends did a small survey for her: two hundred households, asked which laundry they used last month and which this month. The answers give, for each laundry, what fraction of its customers stays and what fraction moves to each competitor within a month. For Bersih, 80% stayed, 10% went to Kilat and 10% to Wangi. For Kilat, 75% stayed, 15% came to Bersih and 10% went to Wangi. For Wangi, the newest, only 60% stayed, 20% went to Bersih and 20% to Kilat.

With these numbers we can play the months forward. This month Bersih has 40% of the customers, Kilat 35% and Wangi 25%. Next month Bersih keeps 80% of its own 40% (32 points), gains 15% of Kilat's 35% (5.25 points) and 20% of Wangi's 25% (5 points): 42.25%. Kilat and Wangi are computed the same way. Repeat for the month after, and the month after that. Do the shares settle down? To what? And how quickly? The answers are an eigenvector, an eigenvalue, and the ratio of two eigenvalues, and the procedure we just described, multiplying by the same table again and again, is the **power method**, the main tool of this chapter.

The same mathematics decides, in quite different settings, how a building sways in an earthquake and how the first web search engine ranked pages. We will meet both. Eigenvalues are sometimes

taught as a purely algebraic topic, a determinant to be expanded; here we meet them as what they really are in applications: the numbers that describe what a system does when it is left to itself.

12.2 The Idea in Plain Words

12.2.1 Directions that a matrix only stretches

A square matrix A turns a vector x into a new vector Ax. Usually the new vector points in a different direction. But for most matrices there are a few special directions that the matrix does not turn at all, only stretches or shrinks. A nonzero vector v with

$$Av = \lambda v \tag{12.1}$$

is an **eigenvector** of A, and the number λ is its **eigenvalue**: the factor by which that direction is stretched. (The German *eigen* means "own": the matrix's own directions and own values.) An $n \times n$ matrix has n eigenvalues, counted with multiplicity, possibly complex. They are the roots of the **characteristic polynomial** $\det(A - \lambda I) = 0$, because Equation (12.1) says that $(A - \lambda I)v = 0$ has a nonzero solution, which happens exactly when $A - \lambda I$ is singular [1], [2], [3].

A small example makes it concrete. Take

$$A = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}.$$

Then $\det(A - \lambda I) = (2 - \lambda)^2 - 1 = 0$ gives $\lambda = 3$ and $\lambda = 1$. For $\lambda = 3$, $(A - 3I)v = 0$ means $-v_1 + v_2 = 0$, so $v = (1,1)$; indeed $A(1,1) = (3,3) = 3(1,1)$. For $\lambda = 1$, $v = (1,-1)$, and $A(1,-1) = (1,-1)$. The matrix stretches the diagonal direction by 3 and leaves the anti-diagonal direction alone.

12.2.2 Why repeated multiplication finds the largest one

Now multiply a vector that is *not* an eigenvector by A several times, say $x = (1,0)$. We get (2,1), then (5,4), then (14,13), then (41,40). The vectors grow, and their direction swings toward (1,1): the ratios of the two components are 2, 1.25, 1.077, 1.025. The reason is simple. Write $(1,0) = \frac{1}{2}(1,1) + \frac{1}{2}(1,-1)$, half of each eigenvector. Each multiplication multiplies the first half by 3 and the second by 1. After k multiplications the first part is 3^k times bigger and the second is unchanged, so the first part dominates more and more. The ratio of successive lengths approaches 3, the largest eigenvalue: $41/14 = 2.93$, already close.

That is the power method in one paragraph: *keep multiplying, and the direction with the largest stretch wins.* How fast it wins depends on how much larger the largest stretch is than the second largest: here 3 against 1, so the unwanted part shrinks by a factor of 3 per step.

> **KEY INSIGHT:** Repeated multiplication by a matrix is what many real processes do by themselves: customers switching month after month, a random reader clicking link after link, a vibration passing from floor to floor. Such processes forget where they started and settle into the matrix's dominant eigenvector. The power method simply lets the process run on the computer.

12.3 The Laundry Market as a Markov Chain

12.3.1 The switching matrix

Let $x_k = (x_{k,1}, x_{k,2}, x_{k,3})$ be the shares of Bersih, Kilat and Wangi in month k, as fractions that add up to 1. The survey gives the **switching matrix** P, whose entry P_{ij} is the fraction of laundry j's customers who use laundry i next month:

$$P = \begin{pmatrix} 0.80 & 0.15 & 0.20 \\ 0.10 & 0.75 & 0.20 \\ 0.10 & 0.10 & 0.60 \end{pmatrix}, \qquad x_{k+1} = P x_k. \tag{12.2}$$

Column j describes where laundry j's customers go, so each column adds up to 1: every customer goes somewhere. A process in which the next state depends only on the current one through such a matrix is a **Markov chain**, and P is its (column-)stochastic transition matrix [1]. By Equation (12.2), $x_k = P^k x_0$: the shares after k months are the starting shares multiplied k times by P, the power method exactly.

12.3.2 What the eigenvalues say

Because every column of P adds up to 1, the row vector (1,1,1) satisfies $(1,1,1)P = (1,1,1)$. So 1 is an eigenvalue of P^{T}, and since a matrix and its transpose have the same eigenvalues, 1 is an eigenvalue of P. Its eigenvector s with $Ps = s$, scaled to add up to 1, is a set of shares that *does not change* from month to month: the **stationary distribution**. For a matrix with all entries positive, as ours, the Perron–Frobenius theorem guarantees that this eigenvector is unique and has all components positive, and that every other eigenvalue satisfies $|\lambda| < 1$ [2]. Writing the starting shares as $x_0 = s + c_2 v_2 + c_3 v_3$ in the eigenvector basis,

$$x_k = P^k x_0 = s + c_2 \lambda_2^k v_2 + c_3 \lambda_3^k v_3 \to s, \tag{12.3}$$

and the distance to the stationary shares shrinks like $|\,\lambda_2\,|^k$, where λ_2 is the eigenvalue second largest in absolute value. (Here the coefficient of s is exactly 1, because x_0, s and every $P^k x_0$ add up to 1 while the other eigenvectors of a stochastic matrix add up to 0.) The same argument for a general matrix, with λ_1 in place of 1, is the convergence proof of the power method: the unwanted components shrink by $_2 / _1$ per step.

12.3.3 The program

Listing 12.1 plays fifteen months forward from 40%, 35% and 25%, prints the shares and the total change from the previous month, and estimates the convergence factor from the last two changes.

Listing 12.1 `market.py`: the laundry shares month by month, the power method on the switching matrix.

```
"""Where do the laundry customers of a kampung end up? Month after month the
shares are multiplied by the switching matrix: the power method at work."""

NAMES = ["Bersih", "Kilat", "Wangi"]
# P[i][j]: share of laundry j's customers who use laundry i next month.
P = [[0.80, 0.15, 0.20],
     [0.10, 0.75, 0.20],
     [0.10, 0.10, 0.60]]


def mat_vec(a, x):
    return [sum(a[i][j] * x[j] for j in range(len(x))) for i in range(len(a))]


def main():
    x = [0.40, 0.35, 0.25]                  # shares this month
    print("month  " + "  ".join(f"{n:>7}" for n in NAMES) + "     change")
    print(f"{0:5d}  " + "  ".join(f"{v:7.4f}" for v in x))
    changes = []
    for month in range(1, 16):
        new = mat_vec(P, x)
        change = sum(abs(new[i] - x[i]) for i in range(3))
        changes.append(change)
        x = new
        print(f"{month:5d}  " + "  ".join(f"{v:7.4f}" for v in x)
              + f"  {change:9.2e}")
    ratio = changes[-1] / changes[-2]
    print(f"change shrinks by a factor {ratio:.4f} per month")
    print("long-run shares: " + ", ".join(
        f"{n} {100 * v:.1f}%" for n, v in zip(NAMES, x)))


if __name__ == "__main__":
    main()
```

Listing 12.2 `Market.java`: the same in Java.

```
import java.util.Locale;

/** Where do the laundry customers of a kampung end up? Month after month the
 *  shares are multiplied by the switching matrix: the power method at work. */
public class Market {
    static final String[] NAMES = {"Bersih", "Kilat", "Wangi"};
    /** P[i][j]: share of laundry j's customers who use laundry i next month. */
    static final double[][] P = {{0.80, 0.15, 0.20},
                                 {0.10, 0.75, 0.20},
                                 {0.10, 0.10, 0.60}};

    static double[] matVec(double[][] a, double[] x) {
        double[] r = new double[a.length];
        for (int i = 0; i < a.length; i++) {
            for (int j = 0; j < x.length; j++) {
                r[i] += a[i][j] * x[j];
            }
        }
        return r;
    }

    static String row(double[] x) {
        StringBuilder s = new StringBuilder();
        for (int i = 0; i < x.length; i++) {
            s.append(i > 0 ? "  " : "")
             .append(String.format(Locale.ROOT, "%7.4f", x[i]));
        }
        return s.toString();
    }

    public static void main(String[] args) {
        double[] x = {0.40, 0.35, 0.25};          // shares this month
        StringBuilder head = new StringBuilder("month  ");
        for (int i = 0; i < NAMES.length; i++) {
            head.append(i > 0 ? "  " : "").append(String.format("%7s", NAMES[i]));
        }
        System.out.println(head + "     change");
        System.out.println(String.format(Locale.ROOT, "%5d  ", 0) + row(x));
        double[] changes = new double[16];
        for (int month = 1; month <= 15; month++) {
            double[] next = matVec(P, x);
            double change = 0;
            for (int i = 0; i < 3; i++) {
                change += Math.abs(next[i] - x[i]);
            }
            changes[month] = change;
```

Listing 12.2 (Continued) *Market.java: the same in Java.*

```
            x = next;
            System.out.println(String.format(Locale.ROOT, "%5d  ", month) + row(x)
                               + String.format(Locale.ROOT, "  %9.2e", change));
        }
        System.out.println(String.format(Locale.ROOT,
            "change shrinks by a factor %.4f per month",
            changes[15] / changes[14]));
        StringBuilder s = new StringBuilder("long-run shares: ");
        for (int i = 0; i < 3; i++) {
            s.append(i > 0 ? ", " : "").append(NAMES[i])
             .append(String.format(Locale.ROOT, " %.1f%%", 100 * x[i]));
        }
        System.out.println(s);
    }
}
```

Listing 12.2 prints the same table:

```
month    Bersih     Kilat     Wangi      change
    0    0.4000    0.3500    0.2500
    1    0.4225    0.3525    0.2250    5.00e-02
    2    0.4359    0.3516    0.2125    2.68e-02
    3    0.4439    0.3498    0.2063    1.61e-02
    4    0.4489    0.3480    0.2031    9.86e-03
    5    0.4519    0.3465    0.2016    6.10e-03
    6    0.4538    0.3454    0.2008    3.81e-03
    7    0.4550    0.3446    0.2004    2.40e-03
    8    0.4558    0.3440    0.2002    1.52e-03
    9    0.4563    0.3436    0.2001    9.68e-04
   10    0.4566    0.3434    0.2000    6.19e-04
   11    0.4568    0.3432    0.2000    3.98e-04
   12    0.4569    0.3431    0.2000    2.56e-04
   13    0.4570    0.3430    0.2000    1.65e-04
   14    0.4570    0.3430    0.2000    1.07e-04
   15    0.4571    0.3429    0.2000    6.91e-05
change shrinks by a factor 0.6471 per month
long-run shares: Bersih 45.7%, Kilat 34.3%, Wangi 20.0%
```

The first line after month 0 is the hand computation of Section 12.1: Bersih 42.25%. The shares settle toward Bersih 45.7%, Kilat 34.3% and Wangi 20.0%, whatever the start, and each month the remaining change is about 0.65 times the previous one. **Figure 12.1** shows the three curves approaching their long-run levels.

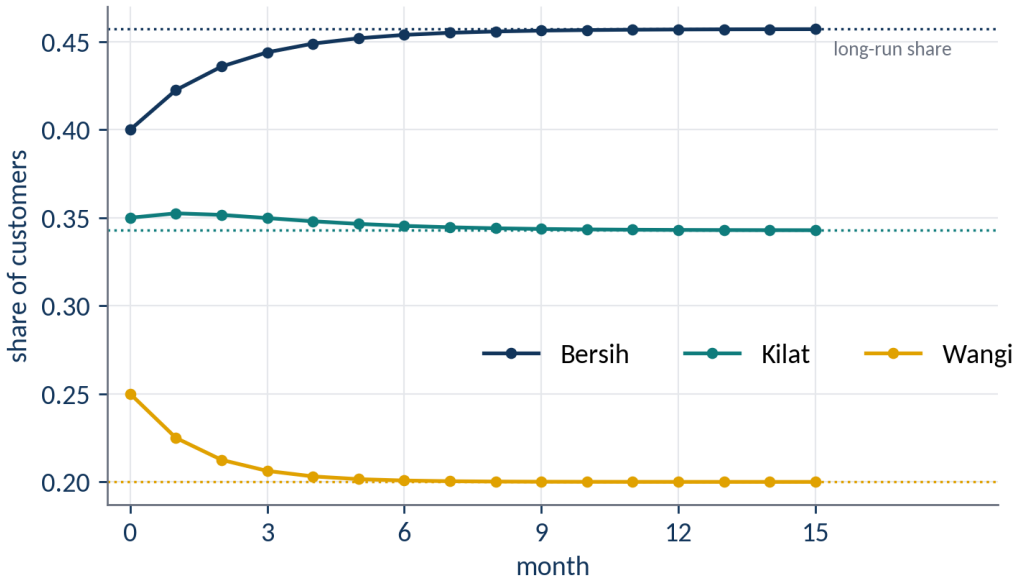


Figure 12.1 Shares of the three laundries month by month from 40%, 35% and 25%; the dotted lines are the stationary shares, the eigenvector of the switching matrix for the eigenvalue 1.

Both numbers can be checked exactly. The stationary shares solve $(P - I)s = 0$ with $s_1 + s_2 + s_3 = 1$. The third row gives $0.1s_1 + 0.1s_2 - 0.4s_3 = 0$, and the first gives $-0.2s_1 + 0.15s_2 + 0.2s_3 = 0$; with the sum condition the solution is $s = (16/35, 12/35, 7/35) = (0.4571, 0.3429, 0.2000)$, exactly what the program converged to. For the eigenvalues, the sum of the eigenvalues equals the trace (the sum of the diagonal), $0.80 + 0.75 + 0.60 = 2.15$, and their product equals the determinant, 0.325. With $\lambda_1 = 1$, the other two satisfy $\lambda_2 + \lambda_3 = 1.15$ and $\lambda_2\lambda_3 = 0.325$, so they are the roots of $\lambda^2 - 1.15\lambda + 0.325 = 0$: $\lambda_2 = 0.65$ and $\lambda_3 = 0.5$. The measured factor 0.6471 is λ_2 seen through the last traces of λ_3.

For Bu Endang the message is encouraging but not dramatic: if customers keep switching as they did this month, Bersih's share will grow from 40% to about 46%, mostly at Wangi's expense, and

within half a year the market will be within one percentage point of that. The number to watch is Wangi's low loyalty, 60%; if Wangi improves it, the eigenvector changes, and the program answers the new question in a second. The model advises; the numbers depend on how customers behave next year, which no survey guarantees.

12.3.4 A fourth state: washing at home

The survey had one more answer that the three-laundry model ignored: a few households stopped using any laundry and washed at home again, and a few home-washers tried a laundry for the first time. Adding "home" as a fourth state keeps the model honest. Suppose each laundry loses 5% of its customers to home washing each month (taken from those who would otherwise have stayed), and each month 6% of home-washers try Bersih, 4% Kilat and 5% Wangi, while 85% keep washing at home. The columns still add up to 1, so 1 is still an eigenvalue, and the stationary vector, computed exactly as before, is Bersih 34.6%, Kilat 25.6%, Wangi 14.8% and home 25.0%. The ranking among the laundries is unchanged, but a quarter of the households end up outside the market altogether, which is information Bu Endang needs: her competition is not only the two other laundries but also the washing machine at home.

The eigenvalues of this four-state matrix are 1, 0.8, 0.61 and 0.46. The second one has grown from 0.65 to 0.8, because the home state changes slowly (85% stay), and the market now takes about $\ln 0.5/\ln 0.8 \approx 3.1$ months to close half of any remaining gap instead of 1.6. A slowly changing group of customers slows the whole market down, and λ_2 measures exactly how much.

12.4 The Power Method in General

12.4.1 The algorithm and the Rayleigh quotient

The method goes back to Richard von Mises and Hilda Pollaczek-Geiringer, who described it in 1929 [4]. For a general matrix the vectors $A^k x_0$ grow or shrink like $|\,\lambda_1\,|^k$ and would soon overflow or underflow, so the power method normalizes after each multiplication:

$$x_{k+1} = \frac{A x_k}{\| A x_k \|}.$$

By Equation (12.3), x_k approaches the dominant eigenvector (up to sign), with an error that shrinks like $|\,\lambda_2/\lambda_1\,|^k$, provided $_1> _2$ and the starting vector has some component along v_1 (a random start almost surely does). The eigenvalue is estimated from the current vector by the **Rayleigh quotient**

$$r(x) = \frac{x^{\mathsf{T}} A x}{x^{\mathsf{T}} x}, \tag{12.4}$$

which equals λ exactly when x is an eigenvector. For a **symmetric** matrix, whose eigenvectors are orthogonal, a vector with an error of size ϵ gives a Rayleigh quotient with an error of size ϵ^2 [2]. The eigenvalue estimate therefore converges like $|\,\lambda_2/\lambda_1\,|^{2k}$, twice as many correct digits per step as the vector. Our programs normalize to length 1, so the denominator of Equation (12.4) is 1 and the quotient is simply $x^{\mathsf{T}} A x$.

The power method needs only matrix–vector products. That is its great strength: for a matrix with billions of rows but only a few nonzero entries per row, as for the web in Section 12.6, a

product costs a few billion operations while any method that factors the matrix would be impossible. Its weaknesses are equally clear: it finds only the dominant eigenvalue, and it is slow when $_2/_1$ is close to 1. The next section overcomes both with one idea.

12.4.2 Inverse and shifted inverse iteration

If $Av = \lambda v$ and σ is any number that is not an eigenvalue, then $(A - \sigma I)v = (\lambda - \sigma)v$ and therefore

$$(A - \sigma I)^{-1} v = \frac{1}{\lambda - \sigma}\, v. \tag{12.5}$$

The matrix $(A - \sigma I)^{-1}$ has the same eigenvectors as A, and its eigenvalues are $1/(\lambda_i - \sigma)$. The eigenvalue of A *closest to* σ becomes the *largest* eigenvalue of $(A - \sigma I)^{-1}$, so the power method applied to $(A - \sigma I)^{-1}$ converges to it, at the rate ${} - / {} - $. With $\sigma = 0$ this is **inverse iteration**, which finds the eigenvalue of smallest absolute value; with σ near a rough guess it is **shifted inverse iteration**, which finds the eigenvalue nearest the guess, and converges very fast when the guess is good.

The inverse is never formed. Each step solves the linear system $(A - \sigma I)y = x_k$, and since the matrix is the same at every step, it is factored once with the LU decomposition of Chapter 6 and each step costs only a forward and a back substitution. When the shift is updated at every step with the current Rayleigh quotient, the method is called Rayleigh quotient iteration and, for symmetric matrices, converges cubically [2].

12.4.3 Inverse iteration by hand

The small matrix of Section 12.2 shows inverse iteration at work. Its eigenvalues are 3 and 1, so inverse iteration ($\sigma = 0$) should find the smaller one, 1, with eigenvector $(1, -1)$. Start again from $x_0 = (1,0)$ and solve $Ay = x_0$: the inverse of A is $\frac{1}{3}\begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix}$, so $y = \frac{1}{3}(2, -1)$. Only the direction matters, so take $x_1 = (2, -1)$ (normalization would divide by $\sqrt{5}$). Its Rayleigh quotient is $x_1^{\mathsf{T}} A x_1 / x_1^{\mathsf{T}} x_1 = (2 \cdot 4 + 2 \cdot 2 \cdot (-1) + 2 \cdot 1)/5 = 6/5 = 1.2$. The next solve gives $y = \frac{1}{3}(5, -4)$, so $x_2 = (5, -4)$, with Rayleigh quotient $(50 - 40 + 32)/41 = 42/41 = 1.024$. Then $x_3 = (14, -13)$ and a quotient of 1.0027. The component along $(1,1)$ is divided by 3 at each step relative to the component along $(1, -1)$, because the eigenvalues of A^{-1} are $\frac{1}{3}$ and 1; and the Rayleigh quotient, as promised for a symmetric matrix, gains about twice as many digits per step as the vector. Compare with Section 12.2.2: the same matrix, the same start, and the same speed, but now converging to the *other* eigenvector.

12.4.4 When the power method fails

The convergence proof needed $_1> _2$, and when that fails, so does the method. The simplest example is the matrix that swaps two components, $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, with eigenvalues 1 and -1 and eigenvectors $(1,1)$ and $(1, -1)$. Starting from $(1,0)$, the power method produces $(0,1)$, $(1,0)$, $(0,1)$, and so on forever: the two components have equal size, neither wins, and the iteration flips instead of settling. A Markov chain can do the same: if every customer of shop A moved to shop B each month and every customer of B moved to A, the shares would swap monthly and never settle. Such a chain is called *periodic*, and it is exactly the situation that the positive entries of the laundry matrix, or the random jump of PageRank, rule out. A rotation of the plane is another

example: its eigenvalues are a complex pair of equal absolute value, and real power iterations simply rotate. When the plain method wanders like this, a shift helps (the eigenvalues $\lambda - \sigma$ no longer have equal size), or one uses the QR algorithm of the libraries, which handles such cases routinely.

12.5 How a Shophouse Sways

12.5.1 The model

Indonesia lies where several tectonic plates meet, and earthquakes are part of life. A three-story *ruko*, the shophouse with a shop on the ground floor and the family living above, is the kind of building many Indonesians live and work in. When the ground shakes, the building does not sway in an arbitrary way; it prefers certain patterns of motion, its **modes**, each with its own natural frequency. If the ground happens to shake near one of those frequencies, the building's motion builds up, which is why engineers must know them. The Indonesian seismic design standard, SNI 1726:2019, asks designers of buildings to determine the fundamental period of vibration of the structure, the time of one sway in the slowest mode [5].

A first model treats each floor slab as a mass m and the columns of each story as a spring of stiffness k connecting it to the floor below. If u_1, u_2, u_3 are the sideways displacements of the three floors, Newton's second law for each floor gives $mu'' = -Ku$ with the **stiffness matrix**

$$K = k\begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{pmatrix}.$$

The first row, for example, says that floor 1 is pulled back by the columns below it ($-ku_1$) and by the columns above it ($-k(u_1 - u_2)$). The roof has columns only below it, hence the 1 in the corner. A solution in which all floors move in step with the same frequency, $u(t) = v\cos(\omega t)$, requires $-\omega^2 mv = -Kv$, that is,

$$\frac{1}{m}Kv = \omega^2 v, \qquad f = \frac{\omega}{2\pi} = \frac{\sqrt{\lambda}}{2\pi}. \tag{12.6}$$

The mode shapes are the eigenvectors of $A = K/m$, and the eigenvalues $\lambda = \omega^2$ give the natural frequencies f in hertz (sways per second) [6]. With $k/m = 2000$ s^{-2}, realistic for a light concrete frame, A is the matrix in the program. It is symmetric, so its eigenvalues are real and its eigenvectors orthogonal, and the Rayleigh quotient doubles the power method's speed.

12.5.2 Three methods, three modes

Listing 12.3 finds all three modes: the highest with the plain power method, the lowest with inverse iteration, and the middle one with shifted inverse iteration around $\sigma = 3000$. A small LU factorization with partial pivoting is included so that the program stands alone; it is the method of Chapter 6. The function `iterate` is shared by all three: it takes the step to repeat (multiply by A, or solve with the factored matrix), normalizes, computes the Rayleigh quotient with the *original* A, and stops when two successive quotients agree to ten digits.

Listing 12.3 `power.py`: power, inverse and shifted inverse iteration for the three modes of the shophouse.

```
"""Natural frequencies of a three-story shophouse (ruko) modeled as three
masses on springs: K v = omega^2 M v. Power method for the highest mode,
inverse and shifted inverse iteration for the lowest and the middle one."""
import math

KM = 2000.0                                  # story stiffness / story mass, 1/s^2
A = [[2 * KM, -KM, 0.0],
     [-KM, 2 * KM, -KM],
     [0.0, -KM, KM]]


def mat_vec(a, x):
    return [sum(a[i][j] * x[j] for j in range(len(x))) for i in range(len(a))]


def dot(x, y):
    return sum(xi * yi for xi, yi in zip(x, y))


def normalize(x):
    norm = math.sqrt(dot(x, x))
    return [xi / norm for xi in x]


def lu_factor(a):
    """LU with partial pivoting (Chapter 6); returns (LU in one matrix, pivots)."""
    n = len(a)
    lu = [row[:] for row in a]
    piv = list(range(n))
    for k in range(n):
        p = max(range(k, n), key=lambda i: abs(lu[i][k]))
        lu[k], lu[p] = lu[p], lu[k]
        piv[k], piv[p] = piv[p], piv[k]
        for i in range(k + 1, n):
            lu[i][k] /= lu[k][k]
            for j in range(k + 1, n):
                lu[i][j] -= lu[i][k] * lu[k][j]
    return lu, piv
```

Listing 12.3 (Continued) *power.py: power, inverse and shifted inverse iteration for the three modes of the shophouse.*

```
def lu_solve(lu, piv, b):
    n = len(lu)
    y = [b[piv[i]] for i in range(n)]
    for i in range(n):                          # forward: L y = P b
        y[i] -= sum(lu[i][j] * y[j] for j in range(i))
    for i in reversed(range(n)):                # backward: U x = y
        y[i] = (y[i] - sum(lu[i][j] * y[j] for j in range(i + 1, n))) / lu[i][i]
    return y


def iterate(step, a, x, tol=1e-10, max_iter=200):
    """Repeat x <- normalize(step(x)); the Rayleigh quotient estimates lambda."""
    lam_old = None
    for k in range(1, max_iter + 1):
        x = normalize(step(x))
        lam = dot(x, mat_vec(a, x))             # Rayleigh quotient, |x| = 1
        if lam_old is not None and abs(lam - lam_old) <= tol * abs(lam):
            return lam, x, k
        lam_old = lam
    return lam, x, max_iter


def report(name, lam, x, k):
    f = math.sqrt(lam) / (2 * math.pi)
    shape = "  ".join(f"{v / x[-1]:6.3f}" for v in x)   # top floor scaled to 1
    print(f"{name:7s} lambda {lam:10.3f}  f {f:5.2f} Hz  iterations {k:3d}"
          f"  shape {shape}")


def main():
    start = [1.0, 1.0, 1.0]
    report("highest", *iterate(lambda v: mat_vec(A, v), A, start))
    lu, piv = lu_factor(A)                      # factor once, solve many times
    report("lowest", *iterate(lambda v: lu_solve(lu, piv, v), A, start))
    shift = 3000.0
    shifted = [[A[i][j] - (shift if i == j else 0.0) for j in range(3)]
               for i in range(3)]
    lu_s, piv_s = lu_factor(shifted)
    report("middle", *iterate(lambda v: lu_solve(lu_s, piv_s, v), A, start))


if __name__ == "__main__":
    main()
```

Listing 12.4 `Power.java`: the same methods in Java; the result of an iteration is a small `record`.

```
import java.util.Locale;
import java.util.function.UnaryOperator;

/** Natural frequencies of a three-story shophouse (ruko) modeled as three
 *  masses on springs: K v = omega^2 M v. Power method for the highest mode,
 *  inverse and shifted inverse iteration for the lowest and the middle one. */
public class Power {
    static final double KM = 2000.0;       // story stiffness / story mass, 1/s^2
    static final double[][] A = {{2 * KM, -KM, 0.0},
                                 {-KM, 2 * KM, -KM},
                                 {0.0, -KM, KM}};

    static double[] matVec(double[][] a, double[] x) {
        double[] r = new double[a.length];
        for (int i = 0; i < a.length; i++) {
            for (int j = 0; j < x.length; j++) {
                r[i] += a[i][j] * x[j];
            }
        }
        return r;
    }

    static double dot(double[] x, double[] y) {
        double s = 0;
        for (int i = 0; i < x.length; i++) {
            s += x[i] * y[i];
        }
        return s;
    }

    static double[] normalize(double[] x) {
        double norm = Math.sqrt(dot(x, x));
        double[] r = new double[x.length];
        for (int i = 0; i < x.length; i++) {
            r[i] = x[i] / norm;
        }
        return r;
    }
```

Listing 12.4 (Continued) *Power.java: the same methods in Java; the result of an iteration is a small record.*

```
    /** LU with partial pivoting (Chapter 6), in place; returns the pivots. */
    static int[] luFactor(double[][] lu) {
        int n = lu.length;
        int[] piv = new int[n];
        for (int i = 0; i < n; i++) {
            piv[i] = i;
        }
        for (int k = 0; k < n; k++) {
            int p = k;
            for (int i = k + 1; i < n; i++) {
                if (Math.abs(lu[i][k]) > Math.abs(lu[p][k])) {
                    p = i;
                }
            }
            double[] tmp = lu[k]; lu[k] = lu[p]; lu[p] = tmp;
            int t = piv[k]; piv[k] = piv[p]; piv[p] = t;
            for (int i = k + 1; i < n; i++) {
                lu[i][k] /= lu[k][k];
                for (int j = k + 1; j < n; j++) {
                    lu[i][j] -= lu[i][k] * lu[k][j];
                }
            }
        }
        return piv;
    }

    static double[] luSolve(double[][] lu, int[] piv, double[] b) {
        int n = lu.length;
        double[] y = new double[n];
        for (int i = 0; i < n; i++) {             // forward: L y = P b
            y[i] = b[piv[i]];
            double s = 0;
            for (int j = 0; j < i; j++) {
                s += lu[i][j] * y[j];
            }
            y[i] -= s;
        }
        for (int i = n - 1; i >= 0; i--) {        // backward: U x = y
            double s = 0;
            for (int j = i + 1; j < n; j++) {
                s += lu[i][j] * y[j];
            }
            y[i] = (y[i] - s) / lu[i][i];
        }
        return y;
    }
```

Listing 12.4 (Continued) *Power.java: the same methods in Java; the result of an iteration is a small record.*

```
    /** Result of an iteration: eigenvalue, eigenvector, iterations used. */
    record Eigen(double lam, double[] x, int k) { }

    /** Repeat x = normalize(step(x)); the Rayleigh quotient estimates lambda. */
    static Eigen iterate(UnaryOperator<double[]> step, double[][] a, double[] x) {
        double lam = 0, lamOld = Double.NaN;
        for (int k = 1; k <= 200; k++) {
            x = normalize(step.apply(x));
            lam = dot(x, matVec(a, x));           // Rayleigh quotient, |x| = 1
            if (!Double.isNaN(lamOld)
                    && Math.abs(lam - lamOld) <= 1e-10 * Math.abs(lam)) {
                return new Eigen(lam, x, k);
            }
            lamOld = lam;
        }
        return new Eigen(lam, x, 200);
    }

    static void report(String name, Eigen e) {
        double f = Math.sqrt(e.lam()) / (2 * Math.PI);
        StringBuilder shape = new StringBuilder();
        double[] x = e.x();
        for (int i = 0; i < x.length; i++) {       // top floor scaled to 1
            shape.append(i > 0 ? "  " : "")
                 .append(String.format(Locale.ROOT, "%6.3f",
                                       x[i] / x[x.length - 1]));
        }
        System.out.println(String.format(Locale.ROOT,
            "%-7s lambda %10.3f  f %5.2f Hz  iterations %3d  shape %s",
            name, e.lam(), f, e.k(), shape));
    }

    static double[][] copy(double[][] a) {
        double[][] c = new double[a.length][];
        for (int i = 0; i < a.length; i++) {
            c[i] = a[i].clone();
        }
        return c;
    }
```

Listing 12.4 (Continued) *Power.java: the same methods in Java; the result of an iteration is a small record.*

```
    public static void main(String[] args) {
        double[] start = {1.0, 1.0, 1.0};
        report("highest", iterate(v -> matVec(A, v), A, start));
        double[][] lu = copy(A);
        int[] piv = luFactor(lu);                 // factor once, solve many times
        report("lowest", iterate(v -> luSolve(lu, piv, v), A, start));
        double shift = 3000.0;
        double[][] luS = copy(A);
        for (int i = 0; i < 3; i++) {
            luS[i][i] -= shift;
        }
        int[] pivS = luFactor(luS);
        report("middle", iterate(v -> luSolve(luS, pivS, v), A, start));
    }
}
```

Listing 12.4 prints the same three lines:

```
highest lambda   6493.959  f 12.83 Hz  iterations  18  shape  1.802  -2.247   1.000
lowest  lambda    396.125  f  3.17 Hz  iterations   7  shape  0.445   0.802   1.000
middle  lambda   3109.916  f  8.88 Hz  iterations   6  shape -1.247  -0.555   1.000
```

The lowest mode, 3.17 Hz, is the slow sway in which all three floors move the same way, the roof most (shape 0.445,0.802,1). The middle mode, 8.88 Hz, bends the building so that the lower floors move against the roof; the highest, 12.83 Hz, makes neighboring floors move in opposite directions. **Figure 12.2** draws the three shapes. The fundamental period is $1/3.17 = 0.32$ s, in line with the rule of thumb of about 0.1 s per story for low buildings.

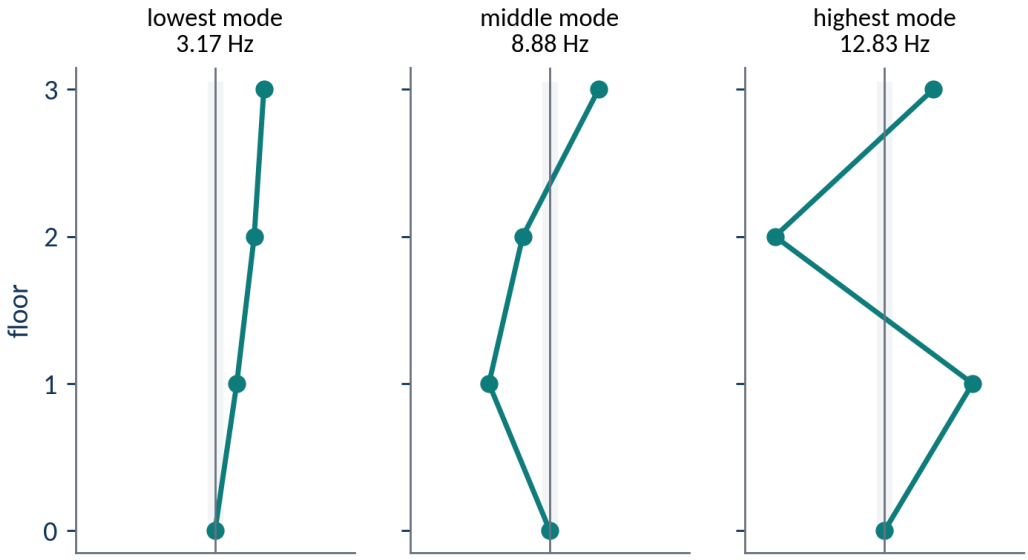


Figure 12.2 The three mode shapes of the shophouse model: sideways displacement of each floor (ground at 0), scaled so that the roof moves one unit, with the natural frequency of each mode.

The iteration counts confirm the theory. The power method needs 18 steps, because the Rayleigh quotient improves by $(\lambda_2/\lambda_1)^2 = (3110/6494)^2 = 0.23$ per step. Inverse iteration needs 7: its ratio is $(396/3110)^2 = 0.016$. Shifted inverse iteration needs 6: the shift 3000 lies 110 from the middle eigenvalue and 2604 from the nearest other one, a ratio of $(110/2604)^2 = 0.0018$ per step. A good shift turns a slow method into a fast one.

12.5.3 A heavier roof

Many ruko owners put water tanks, a laundry drying area or even an extra room on the roof. What does a heavier roof do to the frequencies? With the roof mass $1.5m$ instead of m, the mass matrix is no longer a multiple of the identity, and Equation (12.6) becomes the *generalized* eigenvalue problem $Kv = \omega^2 Mv$ with $M = m\,\mathrm{diag}(1,1,1.5)$. It can be reduced to an ordinary one, $M^{-1}Kv = \omega^2 v$, but $M^{-1}K$ is not symmetric, which costs the Rayleigh quotient's double speed. The better reduction keeps symmetry: with $w = M^{1/2}v$ the problem becomes $M^{-1/2}KM^{-1/2}w = \omega^2 w$, a symmetric matrix, and a diagonal M makes $M^{-1/2}$ trivial. Library routines accept the pair directly: `scipy.linalg.eigh(K, M)` solves the symmetric generalized problem. For the heavier roof it gives the frequencies 2.80, 8.33 and 12.62 Hz. The fundamental frequency drops from 3.17 to 2.80 Hz, and the fundamental period rises from 0.32 to 0.36 s. A heavier top makes a building sway more slowly, which is why engineers must account for everything that will stand on a roof, and why a roof converted to carry heavy loads deserves an engineer's look.

12.5.4 Why the frequencies matter: resonance

Why do engineers care so much about these numbers? Consider one mode as a single mass on a spring with natural frequency f_n and a little damping, with damping ratio ζ as in Chapter 11, shaken at its base with frequency f. Its steady swing is the ground motion multiplied by the amplification factor

$$\frac{1}{\sqrt{(1-r^2)^2+(2\zeta r)^2}}, \qquad r=\frac{f}{f_n}.$$

For concrete buildings ζ is typically about 5% [6]. Ground shaking at 1 Hz, far below the shophouse's fundamental 3.17 Hz, gives $r = 0.32$ and a factor of about 1.1: the building simply rides along. Shaking at 3 Hz gives $r = 0.95$ and a factor of about 7; exactly at 3.17 Hz the factor is $1/(2\zeta) = 10$. The same ground motion can therefore be a gentle rocking for one building and a dangerous swing for another, depending only on an eigenvalue. That is why design standards ask for the fundamental period first, and why the heavier roof of Section 12.5.3, which moves the fundamental frequency from 3.17 to 2.80 Hz, can matter: it moves the building's resonance to a different part of the earthquake's spectrum.

For an engineer, the lowest mode is usually the one that matters most, because it carries most of the building's mass in the same direction and because common earthquake ground motions contain much energy at such frequencies. A real design uses finite-element models with thousands of unknowns and a standard's response spectrum; the three-mass model only shows the idea. Any question about the safety of a real building belongs to a licensed structural engineer.

12.6 PageRank: The Eigenvector of the Web

12.6.1 The random reader

In 1998 Sergey Brin and Lawrence Page described the search engine that became Google, and at its heart an eigenvector problem [7]. Which pages are important? Not simply those with many incoming links, since links are cheap to make, but those linked *from important pages*. That definition is circular, and circular definitions of this kind are eigenvector problems.

Imagine a reader who browses at random. On each page she either follows one of its links, chosen at random, with probability d (the *damping factor*, 0.85 in the original paper), or, with probability $1-d$, gets bored and jumps to a random page anywhere. On a page with no links, a *dangling* page, she always jumps to a random page. The **PageRank** of a page is the fraction of time she spends on it in the long run. If page j has L_j outgoing links, the ranks satisfy

$$r_i = \frac{1-d}{n} + d\sum_{j\to i}\frac{r_j}{L_j} + d\sum_{j\ \text{dangling}}\frac{r_j}{n}, \tag{12.7}$$

where n is the number of pages and $j \to i$ means "page j links to page i". In matrix form this is $r = Gr$ with the **Google matrix** $G = dS + \frac{1-d}{n}\mathbf{1}\mathbf{1}^{\mathsf{T}}$, where S is the column-stochastic link matrix with dangling columns replaced by $1/n$. G is a Markov matrix with all entries positive, exactly like the laundry matrix, so by Perron–Frobenius r is its unique stationary distribution, and the power method finds it. The random jump is what guarantees this: without it, a group of pages that link only to each other could trap the reader, and the ranks would not be unique. It also bounds the

speed: the second eigenvalue of G is at most d in absolute value [8], so the error shrinks at least by a factor 0.85 per step whatever the web looks like.

12.6.2 A village-tourism network

A village in the hills promotes itself as a *desa wisata*, a tourism village, with eight small websites: the village portal, a homestay, the durian orchard tour (Anto's orchard from Chapters 10 and 11 now welcomes visitors in the season), a warung, a batik workshop, a waterfall guide, the ojek-wisata service that drives visitors around, and a travel blog written by a visitor. **Figure 12.3** shows who links to whom. The waterfall guide is a dangling page: it has no links out.

Listing 12.5 applies Equation (12.7) directly, never building the matrix: each page hands its rank, times d, in equal parts to the pages it links to; the rank of dangling pages is spread evenly; and everyone receives the $(1-d)/n$ of the random jump. The iteration stops when the total change is below 10^{-10}.

Listing 12.5 `pagerank.py`: PageRank by the power method, with dangling pages handled explicitly.

```
"""PageRank of the eight websites of a village-tourism network, computed with
the power method on the Google matrix (damping 0.85)."""

PAGES = ["Village portal", "Homestay Melati", "Durian orchard tour",
         "Warung Bu Tini", "Batik workshop", "Waterfall guide",
         "Ojek wisata", "Travel blog"]
LINKS = {0: [1, 2, 3, 4, 5], 1: [0, 2, 3, 6], 2: [0, 1, 3], 3: [0, 2],
         4: [0, 1], 5: [], 6: [1, 2, 5], 7: [2, 1, 5]}   # page 5 has no links
DAMPING = 0.85


def pagerank(links, n, d=DAMPING, tol=1e-10):
    r = [1.0 / n] * n
    for k in range(1, 1000):
        new = [(1 - d) / n] * n
        dangling = sum(r[j] for j in range(n) if not links[j])
        for j in range(n):
            for i in links[j]:                # j shares its rank among its links
                new[i] += d * r[j] / len(links[j])
        for i in range(n):                    # a page without links: to everyone
            new[i] += d * dangling / n
        change = sum(abs(new[i] - r[i]) for i in range(n))
        r = new
        if change < tol:
            return r, k
    return r, k


def main():
    r, k = pagerank(LINKS, len(PAGES))
    print(f"converged after {k} iterations, ranks sum to {sum(r):.12f}")
    order = sorted(range(len(PAGES)), key=lambda i: -r[i])
    for place, i in enumerate(order, 1):
        inbound = sum(i in LINKS[j] for j in LINKS)
        print(f"{place}. {PAGES[i]:20s} {r[i]:.4f}   (linked from {inbound})")


if __name__ == "__main__":
    main()
```

Listing 12.6 `PageRank.java`: the same in Java; the ranking is sorted with a comparator.

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.Locale;

/** PageRank of the eight websites of a village-tourism network, computed with
 *  the power method on the Google matrix (damping 0.85). */
public class PageRank {
    static final String[] PAGES = {"Village portal", "Homestay Melati",
        "Durian orchard tour", "Warung Bu Tini", "Batik workshop",
        "Waterfall guide", "Ojek wisata", "Travel blog"};
    static final int[][] LINKS = {{1, 2, 3, 4, 5}, {0, 2, 3, 6}, {0, 1, 3}, {0, 2},
                                  {0, 1}, {}, {1, 2, 5}, {2, 1, 5}};  // 5: no links
    static final double DAMPING = 0.85;
    static int iterations;

    static double[] pagerank(int[][] links, double d, double tol) {
        int n = links.length;
        double[] r = new double[n];
        Arrays.fill(r, 1.0 / n);
        for (int k = 1; k < 1000; k++) {
            double[] next = new double[n];
            Arrays.fill(next, (1 - d) / n);
            double dangling = 0;
            for (int j = 0; j < n; j++) {
                if (links[j].length == 0) {
                    dangling += r[j];
                }
            }
            for (int j = 0; j < n; j++) {
                for (int i : links[j]) {           // j shares its rank among links
                    next[i] += d * r[j] / links[j].length;
                }
            }
            for (int i = 0; i < n; i++) {          // a page without links: to all
                next[i] += d * dangling / n;
            }
            double change = 0;
            for (int i = 0; i < n; i++) {
                change += Math.abs(next[i] - r[i]);
            }
            r = next;
            iterations = k;
            if (change < tol) {
                break;
            }
        }
```

Listing 12.6 (Continued) *PageRank.java: the same in Java; the ranking is sorted with a comparator.*

```
        return r;
    }

    public static void main(String[] args) {
        double[] r = pagerank(LINKS, DAMPING, 1e-10);
        double sum = 0;
        for (double v : r) {
            sum += v;
        }
        System.out.println(String.format(Locale.ROOT,
            "converged after %d iterations, ranks sum to %.12f", iterations, sum));
        Integer[] order = new Integer[r.length];
        for (int i = 0; i < r.length; i++) {
            order[i] = i;
        }
        Arrays.sort(order, Comparator.comparingDouble(i -> -r[i]));
        for (int place = 0; place < order.length; place++) {
            int i = order[place];
            int inbound = 0;
            for (int[] out : LINKS) {
                for (int t : out) {
                    inbound += t == i ? 1 : 0;
                }
            }
            System.out.println(String.format(Locale.ROOT,
                "%d. %-20s %.4f   (linked from %d)",
                place + 1, PAGES[i], r[i], inbound));
        }
    }
}
```

Listing 12.6 and the Python program print:

```
converged after 25 iterations, ranks sum to 1.000000000000
1. Village portal       0.2168   (linked from 4)
2. Durian orchard tour  0.1969   (linked from 5)
3. Homestay Melati      0.1757   (linked from 5)
4. Warung Bu Tini       0.1585   (linked from 3)
5. Waterfall guide      0.0922   (linked from 3)
6. Ojek wisata          0.0659   (linked from 1)
7. Batik workshop       0.0654   (linked from 1)
8. Travel blog          0.0285   (linked from 0)
```

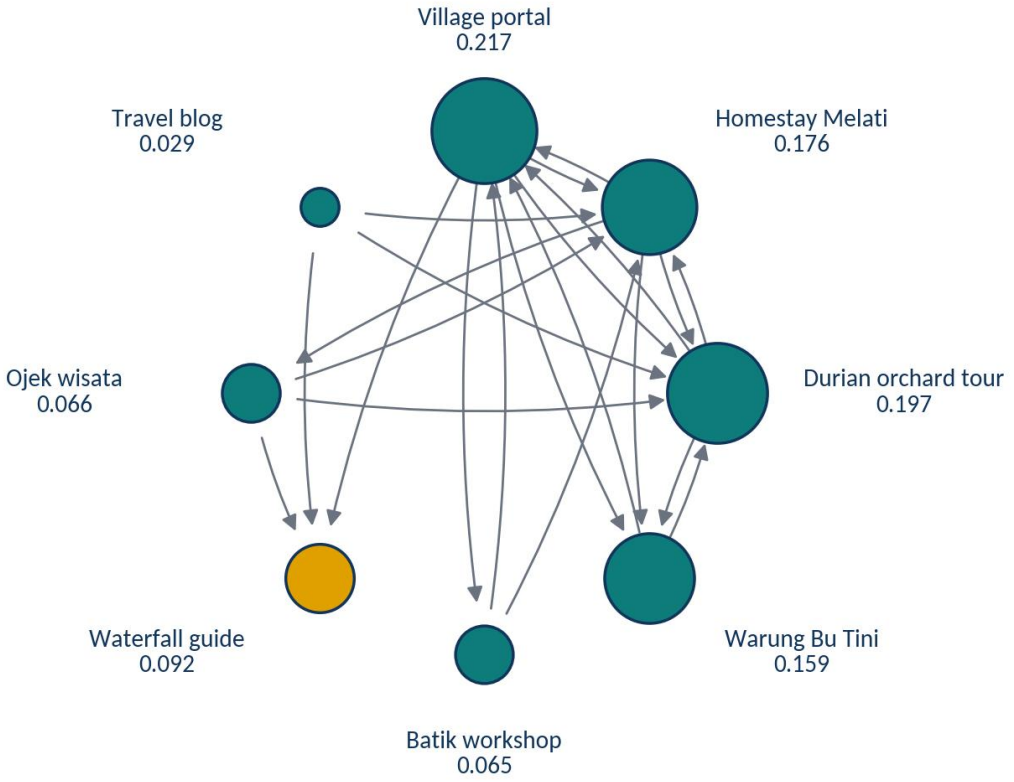


Figure 12.3 The village-tourism network: arrows are links, circle areas are proportional to PageRank, and the waterfall guide (amber) is a dangling page with no links out.

It is worth doing the first step by hand for one page, to see Equation (12.7) at work. All pages start with $r_j = 1/8 = 0.125$. The village portal is linked from the homestay (which has 4 links), the orchard tour (3 links), the warung (2 links) and the batik workshop (2 links), and the waterfall guide is the dangling page. So after one step the portal has $0.15/8 + 0.85 \times 0.125 \times \left(\frac{1}{4} + \frac{1}{3} + \frac{1}{2} + \frac{1}{2}\right) + 0.85 \times 0.125/8 = 0.01875 + 0.16823 + 0.01328 = 0.20026$. Already after one step it is close to its final 0.2168, because most of its inheritance comes from pages whose own ranks are near the average. The program repeats exactly this bookkeeping for all eight pages, 25 times.

Why 0.85? The damping factor balances two wishes. A small d gives the random jump much weight, so the ranks become nearly equal and say little about the links: with $d = 0.5$ the eight ranks lie between 0.07 and 0.17. A d close to 1 lets the links speak almost alone, but the second eigenvalue of G approaches 1, so the power method slows down on large webs, and the ranks become sensitive to small changes in the links: with $d = 0.99$ the travel blog falls to 0.011 and the portal rises to 0.238, and the same program needs 30 iterations instead of 25 here and many more on a real web. Brin and Page chose 0.85 as a compromise [7], and it has remained the customary value [9].

Three lessons stand out. The village portal ranks first although the durian orchard tour and the homestay have more incoming links (five each, against four): the portal is linked from the important pages, the homestay, the orchard and the warung, which pass on a large share of their rank. The waterfall guide, linked from three pages, ranks well above the batik workshop and the ojek service, each linked from one, although it links to nobody; its rank does not vanish, because the dangling rule spreads it back over the network. And the travel blog, linked from no one, gets

only the minimum that the random jump gives every page, $(1-d)/n = 0.15/8 = 0.019$, plus its share of the dangling page's rank. The iteration converged in 25 steps, much faster than the worst case of 0.85^k, because this small network mixes well (Section 12.7 shows its second eigenvalue is 0.38).

A useful variant changes where the bored reader jumps. In *personalized* PageRank the random jump always lands on one chosen page, or a chosen set, instead of anywhere. Jumping always to the durian orchard tour answers the question "starting from the orchard, which pages will a visitor end up reading?". Changing the jump term of Equation (12.7) accordingly (and sending the dangling page's rank to the orchard as well) gives the orchard 0.341, the portal 0.217, the warung 0.167 and the homestay 0.159, while the travel blog drops to 0, since no page links to it and the jump never lands there. The orchard's owner learns that his visitors flow on mostly to the portal, the warung and the homestay, which are the natural partners for a joint promotion. Search engines have used personalized variants of this kind for topic-specific ranking [9].

Early search engines stopped at this. Modern search engines combine many signals of which link structure is one, and the AI answer engines that increasingly reply to questions directly choose which sources to quote with still other signals; but the idea that importance is inherited through links, and that it is computed as an eigenvector, remains one of the central ideas of the web [9].

12.7 Checking Against the Library

NumPy's `numpy.linalg` computes all eigenvalues of a dense matrix at once: `eig` for a general matrix and `eigh` or `eigvalsh` for a symmetric one, using the LAPACK routines based on the QR algorithm, which is the method of choice for small and medium dense matrices [2], [10]. The current release is NumPy 2.5.3 of September 6, 2026 [11]. For very large sparse matrices, `scipy.sparse.linalg.eigs` and `eigsh` find a few eigenvalues with Krylov-subspace methods, sophisticated descendants of the power method that use not only the latest vector but all previous ones. **Listing 12.7** checks the three problems of this chapter.

Listing 12.7 `compare_numpy.py`: the market, the shophouse and PageRank with `numpy.linalg`.

```
"""Check this chapter's eigenvalues with NumPy."""
import numpy as np
import scipy
from market import P
from power import A
from pagerank import LINKS, DAMPING

print(f"NumPy {np.__version__}, SciPy {scipy.__version__}")
w, v = np.linalg.eig(np.array(P))
i = int(np.argmax(w.real))
print("market eigenvalues:", np.round(np.sort(w.real)[::-1], 6))
print("stationary shares: ", np.round(v[:, i].real / v[:, i].real.sum(), 6))
w = np.linalg.eigvalsh(np.array(A))                  # symmetric: eigvalsh
print("building eigenvalues:", np.round(w, 3))
print("frequencies (Hz):    ", np.round(np.sqrt(w) / (2 * np.pi), 2))
n = len(LINKS)
S = np.zeros((n, n))
for j, out in LINKS.items():
    if out:
        S[out, j] = 1 / len(out)
    else:
        S[:, j] = 1 / n                              # dangling page
G = DAMPING * S + (1 - DAMPING) / n                  # the Google matrix
w, v = np.linalg.eig(G)
i = int(np.argmax(w.real))
r = v[:, i].real / v[:, i].real.sum()
print("PageRank:", np.round(r, 4))
print(f"second eigenvalue of G: {sorted(abs(w))[-2]:.4f}")
```

```
NumPy 2.4.4, SciPy 1.17.1
market eigenvalues: [1.   0.65 0.5 ]
stationary shares:  [0.457143 0.342857 0.2     ]
building eigenvalues: [ 396.125 3109.916 6493.959]
frequencies (Hz):     [ 3.17  8.88 12.83]
PageRank: [0.2168 0.1757 0.1969 0.1585 0.0654 0.0922 0.0659 0.0285]
second eigenvalue of G: 0.3847
```

Everything agrees: the market's eigenvalues 1, 0.65 and 0.5 and its stationary shares; the building's three eigenvalues and frequencies; and the PageRank vector, here computed by building the full 8×8 Google matrix and asking for all its eigenvalues. The second eigenvalue of that matrix, 0.3847, explains the 25 iterations: $0.3847^{25} \approx 4 \times 10^{-11}$. Note two practical details in the listing: `eig` returns eigenvectors of length 1 with an arbitrary sign, so the stationary vector must be rescaled to sum to 1; and for a symmetric matrix `eigvalsh` is both faster and more accurate than `eig`, and guarantees real results.

For the Java programs there is no standard eigenvalue library; they are checked against the Python programs, which print the same characters, and by the tests in `EigenTest.java` and `test_eigen.py`: the stationary shares are unchanged by one multiplication, the power method finds 3 for diag(3,1), the LU solver solves a known system, and the PageRank vector sums to 1.

12.8 Numerical Methods in Practice: Profitina

Profitina measures how ready a brand's website is to be cited by AI answer engines, and one family of its signals concerns *technical standing*: how well a site is built and how it is connected. Standing on the web has always been inherited through links, which is the idea of Section 12.6; and within a single website, the same arithmetic describes which pages the site's own internal links make central. When Profitina's scanner crawls a site, it builds the adjacency structure of its pages, and questions such as "which pages would a crawler, following internal links, reach most often?" or "is the page the owner cares about most buried three clicks deep with few links

pointing to it?" are questions about the dominant eigenvector of that link graph. A site whose important product page receives few internal links is telling every crawler, human and machine, that the page is unimportant. Profitina's recommendations about internal linking follow from this class of analysis; the exact signals and their weights are its own, but the eigenvector idea is this chapter's.

Markov chains appear in the community side of the platform too. Readers move between channels of the real-time community much as customers moved between laundries, and the long-run share of attention each channel receives is the stationary distribution of such movements. Knowing that the answer is an eigenvector, and that the speed at which a new channel reaches its long-run share is set by the second eigenvalue, turns vague impressions ("the new channel is catching on") into numbers that can be checked. As always, such numbers advise; the people who run the brand or the channel decide.

12.9 Using It for Real

- **Build the switching table from real counts.** For a shop, ask a sample of customers where they bought last month and this month; count, and divide each column by its total. With fewer than a few dozen customers per shop the fractions are rough; say so when you report the result.
- **Check that the columns add up to 1.** A column that adds up to 0.97 means lost customers (they stopped using any laundry); add a fourth state, "no laundry", instead of hiding the loss.
- **Look at λ_2, not only at the eigenvector.** The stationary shares say where the market goes; $_2$ says how fast. With $\lambda_2 = 0.65$ half of the remaining gap closes every $\ln 0.5 / \ln 0.65 \approx 1.6$ months; with $\lambda_2 = 0.95$ it would take thirteen.
- **Measure how much the survey can be trusted.** Two hundred households give fractions that could easily be off by a few percentage points. A simple way to see what that means for the answer is to resample: draw two hundred households at random, with replacement, from the survey answers, rebuild the switching matrix, recompute the stationary shares, and repeat a hundred times. The spread of Bersih's long-run share over the hundred runs is an honest error bar for the 45.7%, and it costs a few lines of code around `market.py`.
- **Update the matrix as new months come in.** A switching matrix is a snapshot of behavior. Repeat the survey every few months, and watch whether the stationary shares and λ_2 move; a sudden change often means a new competitor or a new price, which the numbers show before anyone says so.
- **Try what-if questions.** Change one entry (Wangi's loyalty from 0.60 to 0.75, rebalancing the column) and rerun; the new eigenvector is the market after the change. This is the same "change one thing, replay" discipline as in earlier chapters.
- **For your own website, compute PageRank on your internal links.** List each page and the pages it links to, as in `LINKS`, run `pagerank.py`, and compare the ranking with the pages you actually want visitors and crawlers to find. Pages that matter to you but rank low need more internal links from pages that rank high.
- **For structures, leave design to engineers.** The shophouse model teaches why buildings have preferred frequencies; it is not a safety assessment. Cracks after an earthquake, sagging floors or new vibrations are reasons to call a structural engineer.

12.10 Common Mistakes, and How to Avoid Them

- **Mixing row and column conventions.** Some books write Markov matrices with rows adding to 1 and multiply a row vector from the left, $x_{k+1}^{\mathsf{T}} = x_k^{\mathsf{T}} P$. Ours has columns adding to 1 and multiplies from the right. Both are fine; mixing them silently transposes the matrix and gives wrong shares. Check the sums.
- **Forgetting to normalize.** Without normalization $A^k x$ overflows for $|\lambda_1| > 1$ and underflows for $|\lambda_1| < 1$. (For a stochastic matrix the sum is preserved, which is why `market.py` could skip it.)
- **Stopping on a small change in the vector when the eigenvalues are close.** If $|\lambda_2/\lambda_1| = 0.999$, the vector changes little per step long before it is accurate. Estimate the ratio from successive changes, as `market.py` does, and use a shift when it is close to 1.
- **Expecting the power method to find a complex or tied eigenvalue.** If λ_1 and λ_2 have equal absolute values (for example 1 and -1, or a complex pair), the plain power method does not converge. Use `numpy.linalg.eig`.
- **Inverting the matrix in inverse iteration.** Factor $A - \sigma I$ once and solve; never form the inverse.
- **Shifting exactly onto an eigenvalue.** Then $A - \sigma I$ is singular. In practice a shift very close to an eigenvalue is fine, even ideal: the nearly singular system is solved with a large but accurate result in the right direction, and normalization takes care of the size.
- **Ignoring dangling pages.** If pages without links are left out of Equation (12.7), rank leaks away at every step and the vector no longer sums to 1.

12.11 Chapter Summary

- An eigenvector of A is a direction that A only stretches; the stretch factor is the eigenvalue (Equation (12.1)). Eigenvalues describe what a system does when left to itself.
- Customers switching between laundries form a Markov chain (Equation (12.2)). Its stationary shares are the eigenvector for the eigenvalue 1, and the distance to them shrinks like $|\lambda_2|^k$ (Equation (12.3)). For the kampung: Bersih 45.7%, Kilat 34.3%, Wangi 20.0%, approached at the rate 0.65 per month.
- The power method repeats $x \leftarrow Ax/\|Ax\|$ and converges to the dominant eigenvector at the rate $\_2/\_1$; for symmetric matrices the Rayleigh quotient (Equation (12.4)) doubles the rate for the eigenvalue.
- Inverse and shifted inverse iteration (Equation (12.5)) find the eigenvalue nearest a chosen shift, using one LU factorization and cheap solves per step. They gave the shophouse's three natural frequencies (Equation (12.6)), 3.17, 8.88 and 12.83 Hz, in 7, 6 and 18 steps.
- When the two largest eigenvalues have equal absolute value, as for a periodic chain or a rotation, the plain power method does not converge; a shift or the QR algorithm of the libraries does.
- A heavier roof turns the shophouse problem into a generalized eigenvalue problem $Kv = \omega^2 Mv$, solved directly by `scipy.linalg.eigh(K, M)`; the fundamental frequency fell from 3.17 to 2.80 Hz.
- PageRank (Equation (12.7)) is the stationary distribution of a random reader with damping; the power method finds it, and the damping guarantees a unique answer and convergence at least at the rate d.
- `numpy.linalg.eig` and `eigvalsh` compute all eigenvalues of small dense matrices; `scipy.sparse.linalg.eigs` finds a few for large sparse ones.

12.12 Exercises

1. By hand, compute the laundry shares after the first and second month from 40%, 35% and 25%, and check against the program. Then verify that $s = (16/35, 12/35, 7/35)$ satisfies $Ps = s$.
2. Using only the trace and the determinant of P and the known eigenvalue 1, find the other two eigenvalues. Predict in which month the total change first drops below 10^{-6}, given that it was 6.91×10^{-5} in month 15.
3. Hotelling deflation: once the dominant eigenpair (λ_1, v_1) of a symmetric matrix is known, with $\| v_1 \| = 1$, the matrix $B = A - \lambda_1 v_1 v_1^{\mathsf{T}}$ has the same eigenvectors, with λ_1 replaced by 0. Write a program in Python and Java that finds the shophouse's highest mode with the power method, deflates it, and finds the second highest with the power method again.
4. The batik workshop persuades the travel blog to add a link to it, so the blog links to the orchard tour, the homestay, the waterfall guide and the batik workshop. Predict qualitatively how much the workshop's rank will rise, then change `LINKS` and run the program.
5. Gershgorin's theorem says that every eigenvalue of A lies in at least one of the discs centered at a_{ii} with radius \${j i} a{ij}\$. Apply it to the shophouse matrix $A = K/m$. What bounds does it give for the frequencies?
6. From the eigenvalues of the shophouse matrix, predict how many iterations the power method with the Rayleigh quotient needs to reach a relative accuracy of 10^{-10}, and compare with the 18 of the program. Do the same for inverse iteration.

12.13 Solutions to the Exercises

1. Month 1: Bersih $0.80 \times 0.40 + 0.15 \times 0.35 + 0.20 \times 0.25 = 0.32 + 0.0525 + 0.05 = 0.4225$; Kilat $0.10 \times 0.40 + 0.75 \times 0.35 + 0.20 \times 0.25 = 0.04 + 0.2625 + 0.05 = 0.3525$; Wangi $0.10 \times 0.40 + 0.10 \times 0.35 + 0.60 \times 0.25 = 0.04 + 0.035 + 0.15 = 0.225$. Month 2: Bersih $0.80 \times 0.4225 + 0.15 \times 0.3525 + 0.20 \times 0.225 = 0.338 + 0.052875 + 0.045 = 0.435875$; Kilat $0.04225 + 0.264375 + 0.045 = 0.351625$; Wangi $0.04225 + 0.03525 + 0.135 = 0.2125$. Rounded, these are the program's 0.4359, 0.3516 and 0.2125. For s: the first component of Ps is $(0.80 \times 16 + 0.15 \times 12 + 0.20 \times 7)/35 = (12.8 + 1.8 + 1.4)/35 = 16/35$; the second is $(1.6 + 9 + 1.4)/35 = 12/35$; the third is $(1.6 + 1.2 + 4.2)/35 = 7/35$. So $Ps = s$.

2. As in Section 12.3.3, the trace is 2.15 and $\det P = 0.8(0.75 \times 0.6 - 0.2 \times 0.1) - 0.15(0.1 \times 0.6 - 0.2 \times 0.1) + 0.2(0.1 \times 0.1 - 0.75 \times 0.1) = 0.344 - 0.006 - 0.013 = 0.325$. The other eigenvalues solve $\lambda^2 - 1.15\lambda + 0.325 = 0$, so $\lambda = \left(1.15 \pm \sqrt{1.3225 - 1.3}\right)/2 = (1.15 \pm 0.15)/2$, that is 0.65 and 0.5. The change shrinks by about 0.65 per month, so from 6.91×10^{-5} it needs k more months with $6.91 \times 10^{-5} \times 0.65^k < 10^{-6}$, that is $k > \ln(0.01447)/\ln(0.65) = 9.8$: ten more months, month 25.

3. **Listing 12.8** and **Listing 12.9** reuse `iterate` and `report` from the shophouse program.

Listing 12.8 `deflation.py`: Hotelling deflation to find the second highest mode.

```
"""Exercise 12.3: after the highest mode, remove it (Hotelling deflation)
and run the power method again to find the second highest."""
from power import A, mat_vec, iterate, report


def main():
    start = [1.0, 1.0, 1.0]
    lam1, v1, k1 = iterate(lambda v: mat_vec(A, v), A, start)
    report("highest", lam1, v1, k1)
    b = [[A[i][j] - lam1 * v1[i] * v1[j] for j in range(3)] for i in range(3)]
    lam2, v2, k2 = iterate(lambda v: mat_vec(b, v), b, start)
    report("second", lam2, v2, k2)


if __name__ == "__main__":
    main()
```

Listing 12.9 `Deflation.java`: the same in Java.

```
/** Exercise 12.3: after the highest mode, remove it (Hotelling deflation)
 *  and run the power method again to find the second highest. */
public class Deflation {
    public static void main(String[] args) {
        double[] start = {1.0, 1.0, 1.0};
        Power.Eigen e1 = Power.iterate(v -> Power.matVec(Power.A, v), Power.A,
                                       start);
        Power.report("highest", e1);
        double[][] b = new double[3][3];
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                b[i][j] = Power.A[i][j] - e1.lam() * e1.x()[i] * e1.x()[j];
            }
        }
        Power.report("second", Power.iterate(v -> Power.matVec(b, v), b, start));
    }
}
```

Both print:

```
highest lambda   6493.959  f 12.83 Hz  iterations  18  shape  1.802  -2.247   1.000
second  lambda   3109.916  f  8.88 Hz  iterations   8  shape -1.247  -0.555   1.000
```

The second run finds the middle mode, 3109.916 and 8.88 Hz, in 8 iterations, with the same shape as shifted inverse iteration found. B has the eigenvalues 0, 3110 and 396, so the power method on B converges at the rate $(396/3110)^2$. Deflation is simple, but errors in λ_1 and v_1 are carried into B, so each further deflation is less accurate; for more than a few eigenvalues, use shifted inverse iteration or a library.

4. The blog itself has the lowest rank in the network, about 0.029, and it would divide its vote among four links instead of three. So the workshop gains at most $0.85 \times 0.029/4 \approx 0.006$ directly, a little more through the pages it links on to. With `LINKS[7] = [2, 1, 5, 4]` the program gives the workshop 0.0714 instead of 0.0654, a gain of 0.006, and moves it just above the ojek service. A link from the village portal, with rank 0.22, would be worth far more than a link from the blog: in PageRank it matters who links to you, not only how many.

5. The rows of $A = 2000\begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{pmatrix}$ give the discs with center 4000 and radius 2000, center 4000 and radius 4000, and center 2000 and radius 2000. Their union is the interval [0,8000] (the eigenvalues of a symmetric matrix are real, so only the real line matters). Hence every $\lambda = \omega^2$ lies between 0 and 8000, and every frequency below $\sqrt{8000}/(2\pi) = 14.2$ Hz. The actual highest

frequency is 12.83 Hz. Gershgorin's bound costs nothing to compute and is often good enough to choose a shift or to check a program's answer for plausibility.

6. For the power method the Rayleigh quotient's error shrinks by $(\lambda_2/\lambda_1)^2 = (3109.9/6494.0)^2 = 0.229$ per step. From an initial relative error of order 1, reaching 10^{-10} needs $k = \ln(10^{-10})/\ln(0.229) = 15.6$ steps. The program took 18: the first step has no previous value to compare with, and the stopping test compares successive estimates rather than the true error. For inverse iteration the rate is $(396.1/3109.9)^2 = 0.0162$, so $k = \ln(10^{-10})/\ln(0.0162) = 5.6$ steps, against 7 in the program, for the same reasons.

References

[1] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical Analysis*, 10th ed. Boston, MA, USA: Cengage Learning, 2016.

[2] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed. Baltimore, MD, USA: Johns Hopkins University Press, 2013.

[3] S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers*, 8th ed. New York, NY, USA: McGraw-Hill Education, 2021.

[4] R. von Mises and H. Pollaczek-Geiringer, "Praktische Verfahren der Gleichungsauflösung," *Zeitschrift für Angewandte Mathematik und Mechanik*, vol. 9, pp. 58–77 and 152–164, 1929.

[5] Badan Standardisasi Nasional, *SNI 1726:2019 Tata Cara Perencanaan Ketahanan Gempa untuk Struktur Bangunan Gedung dan Nongedung*. Jakarta, Indonesia: BSN, 2019.

[6] A. K. Chopra, *Dynamics of Structures: Theory and Applications to Earthquake Engineering*, 5th ed. Hoboken, NJ, USA: Pearson, 2017.

[7] S. Brin and L. Page, "The anatomy of a large-scale hypertextual Web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1–7, pp. 107–117, 1998.

[8] T. H. Haveliwala and S. D. Kamvar, "The second eigenvalue of the Google matrix," Stanford University, Technical Report 2003-20, 2003.

[9] A. N. Langville and C. D. Meyer, *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton, NJ, USA: Princeton University Press, 2006.

[10] NumPy Developers, "Linear algebra (numpy.linalg)," NumPy v2.5 Manual, accessed Sep. 2026. [Online]. Available: https://numpy.org/doc/stable/reference/routines.linalg.html

[11] NumPy Developers, "numpy 2.5.3," Python Package Index, Sep. 6, 2026. [Online]. Available: https://pypi.org/project/numpy/

Chapter 13

Optimization: Golden Section, Newton and Gradient Descent

Every small business asks the same question in many forms: which setting of the knobs I control gives the best result? A tea stall chooses its price, a laundry chooses how many kilograms to wash per load, an orchard chooses when to switch the pump on. Chapters 2 and 3 taught the computer to find where a function is *zero*; this chapter teaches it to find where a function is *largest* or *smallest*. By the end of the chapter you will be able to state an optimization problem in plain words and in symbols; find the best value of one variable with golden-section search, which needs nothing but function values, and explain why it shrinks the interval by the same factor 0.618 at every step; use Newton's method for optimization, derive its quadratic convergence, and see exactly where and why it runs away; move to several variables with gradient descent, compute the largest safe step from the Hessian matrix, and predict the number of steps from its condition number; choose steps automatically with a backtracking line search; write all of it in Java and Python with identical output; and check the answers against SciPy's optimizers.

13.1 Mbak Rina's Price

Mbak Rina sells *es teh jumbo*, a big plastic cup of sweet iced tea, from a small stall in front of a vocational school in Sidoarjo. The tea, the sugar, the ice, the cup and the straw cost her about Rp 1,500 a cup. For two years she has charged Rp 3,000, because that is what the stall next to hers charged when she started. Last month the neighbor raised his price to Rp 5,000 and, to her surprise, still sells plenty. Should she follow him?

She is afraid of two opposite mistakes. If the price is too low, she sells many cups but earns almost nothing on each; at Rp 1,500 she would be working for free. If the price is too high, every cup earns a lot but few students buy; at Rp 10,000 she would sell a handful of cups a day. Somewhere in between lies a price at which the *daily profit*, the margin per cup times the number of cups, is as large as it can be. Finding that price is an optimization problem, and it is the same problem that a laundry faces with its price per kilogram, that an *ojek* driver faces with the number of hours to drive, and that a farmer faces with the amount of fertilizer: more is better up to a point, and then it is worse.

Mbak Rina did what careful traders do. For four weeks she tried different prices on different days and wrote down the number of cups sold. Chapter 8 showed how to fit a curve through such notes; her notes are described well by an exponential demand curve,

$$D(p) = A\,e^{-p/s}, \qquad A = 600,\ s = 2.5, \tag{13.1}$$

where p is the price in thousands of rupiah, $D(p)$ the number of cups sold in a day, A the number of cups she would give away if the tea were free, and s the *price scale*: every time the price rises by s thousand rupiah, sales fall to $1/e \approx 37\%$ of what they were. At Rp 3,000, Equation (13.1) gives $600\,e^{-1.2} \approx 181$ cups, which matches what she sells now; at Rp 5,000 it gives $600\,e^{-2} \approx 81$ cups.

Her daily profit, with a cost of $c = 1.5$ thousand rupiah per cup, is

$$\pi(p) = (p - c)\, D(p) = (p - c)\, A\, e^{-p/s}. \tag{13.2}$$

At Rp 3,000 that is $1.5 \times 181 = 271$ thousand rupiah a day; at Rp 5,000 it is $3.5 \times 81 = 284$ thousand. The neighbor's price is indeed better than hers. But is it the best? That is what this chapter computes, and on the way it builds the three tools that almost every optimizer in science and industry is made of.

Later in the chapter she adds a second product, *gorengan* (fried tofu, tempeh and banana fritters), whose sales depend on both prices, because students who buy tea often buy a snack too. Two prices at once cannot be found by trying a few values by hand, and that is where gradient descent and Newton's method in several variables earn their place.

13.2 The Idea in Plain Words

13.2.1 From finding zeros to finding peaks

Picture the profit curve $\pi(p)$ as a hill. On the left slope, walking to the right (raising the price) goes uphill; on the right slope it goes downhill; at the top the ground is flat. "Flat" means the slope, the derivative $\pi'(p)$, is zero. So one way to find the top is to find the zero of π', and Chapters 2 and 3 already know how to find zeros. That idea is sound and we shall use it, but it needs the derivative, which is not always available, and it cannot tell a hilltop from a valley bottom, where the ground is also flat.

> **DEFINITION.** An *optimization problem* asks for a point x^* in a set of allowed points at which an *objective function* f is smallest (a *minimization* problem) or largest (a *maximization* problem). A *local minimum* is a point where f is not larger than at any nearby point; a *global minimum* is a point where f is not larger than anywhere in the allowed set. Maximizing f is the same as minimizing $-f$, so methods are usually written for minimization only.

A function of one variable is *unimodal* on an interval $[a, b]$ when it has exactly one peak there: it rises, then falls. Mbak Rina's profit is unimodal on [2,10], and for a unimodal function there is a way to find the peak that needs no derivative at all.

13.2.2 Squeezing the peak between two probes

Suppose we know the peak lies somewhere between $a = 2$ and $b = 10$ (Rp 2,000 and Rp 10,000). Taste the curve at two points inside, $x_1 < x_2$. If $\pi(x_1) > \pi(x_2)$, the peak cannot lie to the right of x_2: if it did, the curve would have to rise from x_1 to x_2 on its way to the peak, but it fell. So we may throw away the piece $(x_2, b]$. If instead $\pi(x_1) \le \pi(x_2)$, by the same argument we throw away $[a, x_1)$. Either way, the interval that must contain the peak becomes shorter, and we repeat.

Where should the two probes go? Two wishes decide it. First, we do not know in advance which piece we will throw away, so the two possible remaining intervals should have the same length; that makes the probes symmetric, at distance $r(b - a)$ from the opposite ends, for some fraction r. Second, each step should cost only *one* new evaluation of π: the probe that survives inside the new interval should be exactly where one of the new interval's probes must be. Section 13.3.1 shows that these two wishes force $r = \left(\sqrt{5} - 1\right)/2 \approx 0.618$, the golden ratio's reciprocal, and give the method its name.

Let us do the first step by hand. With $a = 2$, $b = 10$ and $b - a = 8$:

- $x_1 = b - 0.618034 \times 8 = 10 - 4.944272 = 5.055728$ and $x_2 = a + 0.618034 \times 8 = 6.944272$.
- $\pi(x_1) = 3.555728 \times 600\, e^{-2.022291} = 3.555728 \times 79.40 = 282.36$.
- $\pi(x_2) = 5.444272 \times 600\, e^{-2.777709} = 5.444272 \times 37.31 = 203.12$.
- $\pi(x_1) > \pi(x_2)$, so the peak is not in (6.944272,10]. The new interval is [2,6.944272], of length $4.944272 = 0.618 \times 8$.
- The old $x_1 = 5.055728$ is now the *right* probe of the new interval, because $2 + 0.618034 \times 4.944272 = 5.055728$. Only the new left probe $x_1 = 6.944272 - 0.618034 \times 4.944272 = 3.888544$ needs a new evaluation: $\pi(3.888544) = 302.53$.

Compare with the first row printed by the program in Section 13.5: after step 1 the interval is [2,6.94427] with probes 3.88854 and 5.05573 and profits 302.5347 and 282.3644. Every step removes 38.2% of the interval, and after n steps its length is 8×0.618^n.

13.2.3 Following the slope: Newton and gradient descent

When the derivative is available, we can do much better than squeezing. Newton's method from Chapter 3, applied to the equation $\pi'(p) = 0$, replaces the curve near the current guess by the parabola that has the same value, slope and curvature, and jumps to the top of that parabola. Near the answer the number of correct digits roughly doubles at each step.

With two prices, the hill is a surface over a map of (p_1, p_2), and the slope is a vector, the *gradient*, that points in the direction of steepest ascent. Gradient descent, written for minimization, takes a small step in the opposite direction of the gradient of the loss, again and again, like a walker in fog who always steps straight downhill. It needs only the gradient, not the curvature, so each step is cheap; but the size of the step matters enormously, and Section 13.3.4 shows how to compute the largest step that is safe.

> **KEY INSIGHT:** Every optimizer in this chapter is a trade between information and effort per step. Golden section uses only values and gains a constant 0.618 per evaluation. Gradient descent uses slopes and gains a constant factor per step that depends on the shape of the hill. Newton uses slopes and curvatures and, close to the answer, squares the error at every step.

13.3 The Methods, Derived

13.3.1 Golden-section search

Let the current interval be $[a, b]$ with length $L = b - a$, and place the probes at

$$x_1 = b - rL, \qquad x_2 = a + rL, \qquad \frac{1}{2} < r < 1. \tag{13.3}$$

With the probes of Equation (13.3), suppose $f(x_1) > f(x_2)$ (for maximization), so the new interval is $[a, x_2]$ with length rL. Inside it the surviving probe x_1 sits at distance $x_1 - a = L - rL = (1 - r)L$ from the left end. The new right probe must sit at $a + r(rL) = a + r^2 L$. For the old probe to serve as the new one we need

$$r^2 = 1 - r \quad \Leftrightarrow \quad r^2 + r - 1 = 0 \quad \Leftrightarrow \quad r = \frac{\sqrt{5} - 1}{2} \approx 0.6180340. \tag{13.4}$$

The other root is negative and useless. The same r works in the mirror case by symmetry. So each step costs one evaluation and multiplies the length by r. After n steps,

$$L_n = r^n L_0, \qquad n \geq \frac{\ln(L_0/\varepsilon)}{\ln(1/r)} \approx 2.078 \ln \frac{L_0}{\varepsilon} \tag{13.5}$$

steps guarantee a final length below a tolerance ε. For $L_0 = 8$ and $\varepsilon = 10^{-4}$, Equation (13.5) gives $n \geq \ln(80000)/0.48121 = 11.290/0.48121 = 23.46$, so 24 steps and $24 + 2 = 26$ evaluations, which is exactly what the program reports. This is *linear* convergence with rate 0.618, a little slower than bisection's 0.5 per evaluation; but bisection needs a sign, and a peak has no sign change, only values.

Two facts make golden section a reliable workhorse [1], [2]. It never fails on a unimodal function, because every step keeps the peak inside; and its cost is known before it starts. Kiefer proved in 1953 that the Fibonacci search, of which golden section is the limit, is the best possible method of this kind in the worst case [3]. Its weaknesses are just as clear: it finds *a* peak, not *the* highest peak, when the function has several; and its accuracy is limited near the top, where the curve is flat. Near a maximum x^*, Taylor's theorem gives $f(x) \approx f(x^*) + \frac{1}{2} f''(x^*)(x - x^*)^2$, so a change δ in x changes f by only about $1/2\,|f''|\delta^2$. When that change is smaller than the rounding error in f, about $\epsilon_{\text{mach}}|f|$ with the machine epsilon of Chapter 1, the comparison $f(x_1) > f(x_2)$ becomes a coin toss. Setting the two equal,

$$\delta_{\min} \approx \sqrt{\epsilon_{\text{mach}}} \sqrt{\frac{2|f(x^*)|}{|f''(x^*)|}}, \tag{13.6}$$

which is about $\sqrt{\epsilon_{\text{mach}}} \approx 1.5 \times 10^{-8}$ times a scale factor. A tolerance below that is wasted work; this is why good codes, such as the one in SciPy, stop at about 10^{-8} relative [4].

Pseudocode 13.1 summarizes the method in the form that the programs follow.

Pseudocode 13.1 Golden-section search for the maximum of a unimodal function on $[a, b]$.

```
GOLDEN-MAX(f, a, b, tol)
 1  r = (sqrt(5) - 1) / 2
 2  x1 = b - r(b - a);  x2 = a + r(b - a)
 3  f1 = f(x1);  f2 = f(x2)
 4  while b - a > tol
 5      if f1 > f2                      // peak is not in (x2, b]
 6          b = x2;  x2 = x1;  f2 = f1
 7          x1 = b - r(b - a);  f1 = f(x1)
 8      else                            // peak is not in [a, x1)
 9          a = x1;  x1 = x2;  f1 = f2
10          x2 = a + r(b - a);  f2 = f(x2)
11  return (a + b) / 2
```

13.3.2 Newton's method for a maximum

Expand f around the current guess x_k to second order:

$$f(x_k + h) \approx f(x_k) + f'(x_k)\,h + \frac{1}{2} f''(x_k)\,h^2.$$

The right side is a parabola in h. Its derivative $f'(x_k) + f''(x_k)h$ is zero at $h = -f'(x_k)/f''(x_k)$, which gives the Newton step for optimization,

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}. \tag{13.7}$$

Equation (13.7) is Newton's root-finding iteration of Chapter 3 applied to f', so it inherits quadratic convergence near a point where $f' = 0$ and $f'' \neq 0$ [1], [2]. It also inherits a blindness: the parabola's stationary point is its top when $f''(x_k) < 0$ and its *bottom* when $f''(x_k) > 0$. Newton's method goes wherever the parabola is flat, uphill or downhill.

Mbak Rina's curve lets us see all of this exactly. From Equation (13.2), with $m = p - c$ the margin,

$$\pi'(p) = Ae^{-p/s}\left(1 - \frac{m}{s}\right), \qquad \pi''(p) = Ae^{-p/s}\left(\frac{m}{s^2} - \frac{2}{s}\right). \tag{13.8}$$

The slope is zero only at $m = s$, that is $p^* = c + s = 1.5 + 2.5 = 4.0$: **the best price is Rp 4,000**, where the margin equals the price scale. The curvature is negative for $m < 2s$ and positive for $m > 2s$; the curve has an inflection point at $p = c + 2s = 6.5$.

In the Newton step the factor $Ae^{-p/s}$ cancels, and with $e = m - s$ the error in the margin (which equals the error in the price), a line of algebra gives

$$e_{k+1} = -\frac{e_k^2}{s - e_k}. \tag{13.9}$$

Three lessons are packed into Equation (13.9). Near the answer, $e_{k+1} \approx -e_k^2/s$: the error is squared at every step, quadratic convergence with constant $1/s = 0.4$. Starting at Rp 3,000, $e_0 = -1$, so $e_1 = -1/3.5 = -0.2857$, $e_2 = -0.0816/2.7857 = -0.0293$, $e_3 = -3.4 \times 10^{-4}$, $e_4 = -4.6 \times 10^{-8}$: the digits double as promised. Far away, when e_k approaches s (the inflection point), the denominator vanishes and the step explodes; and beyond it, $e_k > s$, the new error is *positive and larger*: starting at Rp 8,000, $e_0 = 4$ and $e_1 = -16/(2.5 - 4) = 10.67$, a jump to Rp 14,667, and the iteration marches off to ever higher prices where the curve is flat and the tea unsold.

The cure is the same one Chapter 3 used for root finding: keep a bracket and fall back to a safe step. We keep an interval $[a, b]$ in which π' changes sign (positive at a, negative at b), update it with the sign of π' at every new point, and accept the Newton point only when $\pi'' < 0$ and the point lies inside the bracket; otherwise we bisect. The result converges whenever bisection does and as fast as Newton once it is close. Numerical libraries combine the same two ingredients, a safe method and a fast one, in Brent's method [4], [5].

13.3.3 Several variables: gradient and Hessian

Now the loss f depends on a vector $\mathbf{x} = (x_1, \ldots, x_n)$. Its *gradient* collects the first partial derivatives and its *Hessian* the second ones:

$$\nabla f(\mathbf{x}) = \begin{pmatrix} \partial f / \partial x_1 \\ \vdots \\ \partial f / \partial x_n \end{pmatrix}, \qquad H(\mathbf{x}) = \left[\frac{\partial^2 f}{\partial x_i \, \partial x_j}\right]_{i,j=1}^{n}. \tag{13.10}$$

Near a point $\mathbf{x}$, Taylor's theorem in several variables, written with the quantities of Equation (13.10), says

$$f(\mathbf{x}+\mathbf{h}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x})^{\mathsf{T}}\mathbf{h} + \frac{1}{2}\,\mathbf{h}^{\mathsf{T}} H(\mathbf{x})\,\mathbf{h}. \tag{13.11}$$

At a local minimum the gradient is the zero vector (otherwise a small step against it would decrease f), and the Hessian is *positive semidefinite*: $\mathbf{h}^{\mathsf{T}}H\mathbf{h} \geq 0$ for every direction $\mathbf{h}$. If H is positive definite (all eigenvalues positive, Chapter 12), a point with zero gradient is a strict local minimum. For a symmetric 2×2 Hessian this is easy to test: the diagonal entries and the determinant must be positive.

Mbak Rina's two products are tea at price p_1 and gorengan at price p_2, with costs $c_1 = 1.5$ and $c_2 = 0.6$ thousand rupiah. Her notes give demand curves in which each price also lowers the other product's sales a little, because the two are bought together:

$$T = 600\,e^{-p_1/2.5 \,-\, 0.10\,p_2}, \qquad G = 900\,e^{-p_2 \,-\, 0.08\,p_1}, \tag{13.12}$$

and the loss to minimize is minus the profit, $f(p_1,p_2) = -[(p_1-c_1)T + (p_2-c_2)G]$. Differentiating with the product rule, with $m_1 = p_1 - c_1$ and $m_2 = p_2 - c_2$, the profit's partial derivatives are

$$\frac{\partial \pi}{\partial p_1} = T - \frac{m_1 T}{2.5} - 0.08\,m_2 G, \qquad \frac{\partial \pi}{\partial p_2} = G - 0.10\,m_1 T - m_2 G,$$

and the gradient of the loss is their negative. At the starting guess (3,1): $T = 600\,e^{-1.3} = 163.52$ cups, $G = 900\,e^{-1.24} = 260.45$ snacks, $m_1 = 1.5$, $m_2 = 0.4$, so $\partial\pi/\partial p_1 = 163.52 - 98.11 - 8.33 = 57.07$ and $\partial\pi/\partial p_2 = 260.45 - 24.53 - 104.18 = 131.74$. Both are positive: raising either price would increase the profit, the snack price more urgently.

A formula for a gradient is easy to get wrong by a sign or a factor. The cheap insurance is the central difference of Chapter 9: $\partial f/\partial x_1 \approx [f(x_1+h,x_2) - f(x_1-h,x_2)]/(2h)$, with an error of order h^2. The program prints both at (3,1) and they agree to all six printed decimals. Every serious optimization code does this *gradient check* before trusting a hand-written gradient.

13.3.4 Gradient descent and its step size

Gradient descent repeats

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha\,\nabla f(\mathbf{x}_k), \tag{13.13}$$

where the *step size* $\alpha > 0$ (in machine learning, the *learning rate*) is chosen by us. By Equation (13.11) with $\mathbf{h} = -\alpha\nabla f$, the loss changes by $-\alpha \parallel \nabla f \parallel^2 + O(\alpha^2)$, so a small enough step always goes downhill. How small is small enough?

Near the minimum $\mathbf{x}^*$ the gradient is approximately $H(\mathbf{x}-\mathbf{x}^*)$, with H the Hessian at the minimum. Writing $\mathbf{e}_k = \mathbf{x}_k - \mathbf{x}^*$, Equation (13.13) becomes $\mathbf{e}_{k+1} = (I - \alpha H)\,\mathbf{e}_k$. Decompose $\mathbf{e}_k$ along the eigenvectors of H (Chapter 12): along the eigenvector with eigenvalue λ_i the component is multiplied by $1 - \alpha\lambda_i$ at every step. All components shrink exactly when every factor is smaller than one in size:

$$|1 - \alpha\lambda_i| < 1 \text{ for all } i \quad \Leftrightarrow \quad 0 < \alpha < \frac{2}{\lambda_{\max}}. \tag{13.14}$$

The slowest component decides the speed. The worst factor, $\max_i |1 - \alpha\lambda_i|$, is smallest when the factors for $\lambda_{\min}$ and $\lambda_{\max}$ are equal and opposite, which happens at $\alpha = 2/(\lambda_{\min} + \lambda_{\max})$, and then

$$\rho = \frac{\lambda_{\max} - \lambda_{\min}}{\lambda_{\max} + \lambda_{\min}} = \frac{\kappa - 1}{\kappa + 1}, \qquad \kappa = \frac{\lambda_{\max}}{\lambda_{\min}}. \tag{13.15}$$

The number κ is the *condition number* of the Hessian: the ratio between the steepest and the flattest curvature of the valley. A round bowl has $\kappa = 1$ and gradient descent reaches the bottom in one step; a long narrow valley has a large κ, the rate ρ is close to 1, and the walker zigzags from wall to wall while creeping along the valley floor [6]. To reduce the error by a factor 10^{-d} takes about $d\ln 10/\ln(1/\rho)$ steps, which for large κ is roughly $1.15\, d\, \kappa$ steps.

For Mbak Rina's two prices the program computes the Hessian at the optimum: eigenvalues 49.34 and 182.06, so $\kappa = 3.69$, the stability limit is $\alpha < 2/182.06 = 0.01099$, the best fixed step is $2/231.39 = 0.00864$, and Equation (13.15) gives $\rho = 2.69/4.69 = 0.574$. The gradient starts at length $\sqrt{57.07^2 + 131.74^2} = 143.6$ and must fall below 10^{-6}, a factor of 1.4×10^8, so we expect $\ln(1.4 \times 10^8)/\ln(1/0.574) = 18.8/0.556 \approx 34$ steps. The program, with $\alpha = 0.008$, needs 35. With $\alpha = 0.002$ the rate is $1 - 0.002 \times 49.34 = 0.901$ and it needs 166 steps. With $\alpha = 0.012$ the factor along the steep direction is $1 - 0.012 \times 182.06 = -1.18$: the error along that direction grows and flips sign at each step, and the iteration never settles.

> **EXAMPLE:** One gradient step by hand. At the start (3,1) the profit is 349.46 thousand rupiah and the gradient of the loss is $(-57.07, -131.74)$, the negative of the profit's slopes computed in Section 13.3.3. With $\alpha = 0.008$, Equation (13.13) gives $p_1 = 3 - 0.008 \times (-57.07) = 3.4566$ and $p_2 = 1 - 0.008 \times (-131.74) = 2.0539$. The profit there is 367.13, an improvement of almost 18 thousand rupiah a day. But the new gradient of the loss is $(-16.47, 63.72)$: its second component has changed sign, which means the snack price has gone past its best value (the optimum is 1.435) and the next step will pull it back. The tea price, in the flatter direction, still wants to rise. This is the first zigzag of **Figure 13.2**: along the steep direction the step α is large compared with $1/\lambda_{\max}$, so the iterate overshoots; along the flat direction it is small compared with $1/\lambda_{\min}$, so the iterate creeps. With $\kappa = 3.7$ the zigzag dies out in a few steps; with $\kappa = 100$ it would dominate the whole walk.

13.3.5 Backtracking: letting the method choose the step

Computing the Hessian's eigenvalues just to pick a step defeats the purpose of a cheap method. A *line search* chooses the step at every iteration from the function itself. The simplest reliable one is *backtracking*: try $\alpha = 1$; while the loss does not drop by at least a small fraction of what the slope promises, halve α. The test is the *Armijo condition* [7]:

$$f(\mathbf{x}_k - \alpha \nabla f_k) \le f(\mathbf{x}_k) - c\,\alpha \,\| \nabla f_k \|^2, \qquad c = 10^{-4}. \tag{13.16}$$

Because f decreases at the rate $\| \nabla f \|^2$ for tiny steps, a step that satisfies Equation (13.16) always exists, and halving finds it in a few tries. Backtracking removes the danger of divergence entirely, at the cost of extra function evaluations: on Mbak Rina's problem it takes the same 35 steps as the best fixed step but 294 trial points, because each step starts at $\alpha = 1$ and halves about eight times to reach $1/128 \approx 0.0078$. A simple improvement, starting each search from twice the previous step, is Exercise 13.4. Nocedal and Wright give the full theory, including the *Wolfe conditions* that also bound the step from below [6].

13.3.6 Newton's method in several variables

Minimizing the quadratic model Equation (13.11) over $\mathbf{h}$ gives the condition $\nabla f + H\mathbf{h} = \mathbf{0}$, a linear system. Newton's step is its solution:

$$H(\mathbf{x}_k)\,\mathbf{s}_k = \nabla f(\mathbf{x}_k), \qquad \mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{s}_k. \tag{13.17}$$

For two variables the system is solved by Cramer's rule, as in the programs; for many variables by the Gaussian elimination of Chapter 4 or, since a Hessian at a minimum is symmetric positive definite, by the Cholesky factorization of Chapter 6. Newton's method sees the whole shape of the bowl, so it is not slowed by a large condition number: on a quadratic it lands at the minimum in one step, whatever κ is. Near a minimum with positive definite Hessian it converges quadratically [2], [6]. Its costs are the second derivatives, $n(n+1)/2$ of them, and a linear solve of order n^3 per step, which is why problems with millions of variables use gradient methods or *quasi-Newton* methods such as BFGS, which build an approximate Hessian from successive gradients [6].

Table 13.1 summarizes the four methods.

Table 13.1 The optimizers of this chapter: what they need, what a step costs, and how fast they converge.

Method	Needs	Cost per step	Convergence	Main danger
Golden section	values, a bracket	1 evaluation	linear, rate 0.618	several peaks
Newton (1 variable)	f', f''	2 derivatives	quadratic near x^*	wrong side of an inflection
Gradient descent	∇f	1 gradient	linear, $(\kappa-1)/(\kappa+1)$	step above $2/\lambda_{\max}$
Newton (n variables)	∇f, H	gradient, Hessian, solve	quadratic near $\mathbf{x}^*$	H not positive definite

13.4 The Programs

The chapter's code lives in `Chapter13/python` and `Chapter13/java`; every program prints exactly the same text in both languages, so either can be read alongside the output. None needs a library except the comparison with SciPy in Section 13.6.

13.4.1 Golden section for one price

Listing 13.1 implements **Pseudocode 13.1** for Mbak Rina's profit. The function `golden_max` keeps the four numbers x_1, x_2, f_1, f_2 and updates them exactly as the pseudocode does; it prints the first six steps and the last one. After the search, `main` compares with the exact optimum $c + s$ and with a brute-force grid of every Rp 50 between Rp 2,000 and Rp 10,000.

Listing 13.1 `golden.py`: golden-section search for the best tea price.

```
"""Golden-section search for the price that maximizes Mbak Rina's daily profit.
Usage: python golden.py"""
import math

A, S, C = 600.0, 2.5, 1.5    # cups at price 0, price scale, cost (thousand Rp)
R = (math.sqrt(5.0) - 1.0) / 2.0   # 0.618..., the golden-section ratio


def cups(p):
    """Cups of tea sold in a day at price p (thousand rupiah)."""
    return A * math.exp(-p / S)


def profit(p):
    """Daily profit in thousand rupiah."""
    return (p - C) * cups(p)


def golden_max(f, a, b, tol, trace=False):
    """Maximize a unimodal f on [a, b]; return (x, iterations, evaluations)."""
    x1, x2 = b - R * (b - a), a + R * (b - a)
    f1, f2 = f(x1), f(x2)
    k, evals = 0, 2
    while b - a > tol:
        k += 1
        if f1 > f2:              # the peak cannot lie in (x2, b]
            b, x2, f2 = x2, x1, f1
            x1 = b - R * (b - a)
            f1 = f(x1)
        else:                    # the peak cannot lie in [a, x1)
            a, x1, f1 = x1, x2, f2
            x2 = a + R * (b - a)
            f2 = f(x2)
        evals += 1
        if trace and (k <= 6 or b - a <= tol):
            print(f"{k:3d} {a:9.5f} {b:9.5f} {b - a:9.5f} {x1:9.5f}"
                  f" {x2:9.5f} {f1:10.4f} {f2:10.4f}")
    return (a + b) / 2.0, k, evals
```

Listing 13.1 (Continued) *golden.py: golden-section search for the best tea price.*

```
def main():
    print("  k         a         b     b - a        x1        x2"
          "     pi(x1)     pi(x2)")
    p, k, evals = golden_max(profit, 2.0, 10.0, 1e-4, trace=True)
    print(f"golden section: p = {p:.5f} after {k} steps, {evals} evaluations")
    print(f"exact optimum : p = {C + S:.5f} (from pi'(p) = 0)")
    print(f"cups {cups(p):.1f}, profit {profit(p):.3f} thousand rupiah a day")
    grid = [2.0 + 0.05 * i for i in range(161)]     # every Rp 50
    best = max(grid, key=profit)
    print(f"grid of {len(grid)} prices: best p = {best:.2f},"
          f" profit {profit(best):.3f}")


if __name__ == "__main__":
    main()
```

Listing 13.2 is the same program in Java. The function to maximize is passed as a `DoubleUnaryOperator`, Java's type for "a function from `double` to `double`", and the method reference `Golden::profit` supplies it. All formatting goes through `String.format` with `Locale.ROOT`, so that the decimal point is a point on every computer, including those set to Indonesian, where the default would print a comma.

Listing 13.2 `Golden.java`: the same search in Java.

```
// Golden-section search for the price that maximizes Mbak Rina's daily profit.
// Usage: java Golden
import java.util.Locale;
import java.util.function.DoubleUnaryOperator;

public class Golden {
    static final double A = 600.0, S = 2.5, C = 1.5;
    static final double R = (Math.sqrt(5.0) - 1.0) / 2.0;

    static double cups(double p) { return A * Math.exp(-p / S); }

    static double profit(double p) { return (p - C) * cups(p); }

    static String f(String fmt, Object... xs) {
        return String.format(Locale.ROOT, fmt, xs);
    }

    /** Maximize a unimodal f on [a, b]; returns {x, iterations, evaluations}. */
    static double[] goldenMax(DoubleUnaryOperator fn, double a, double b,
                              double tol, boolean trace) {
        double x1 = b - R * (b - a), x2 = a + R * (b - a);
        double f1 = fn.applyAsDouble(x1), f2 = fn.applyAsDouble(x2);
        int k = 0, evals = 2;
        while (b - a > tol) {
            k++;
            if (f1 > f2) {                  // the peak cannot lie in (x2, b]
                b = x2; x2 = x1; f2 = f1;
                x1 = b - R * (b - a);
                f1 = fn.applyAsDouble(x1);
            } else {                        // the peak cannot lie in [a, x1)
                a = x1; x1 = x2; f1 = f2;
                x2 = a + R * (b - a);
                f2 = fn.applyAsDouble(x2);
            }
            evals++;
            if (trace && (k <= 6 || b - a <= tol))
                System.out.println(f("%3d %9.5f %9.5f %9.5f %9.5f %9.5f"
                        + " %10.4f %10.4f", k, a, b, b - a, x1, x2, f1, f2));
        }
        return new double[] {(a + b) / 2.0, k, evals};
    }
```

Listing 13.2 (Continued) *Golden.java: the same search in Java.*

```
    public static void main(String[] args) {
        System.out.println("  k         a         b     b - a        x1        x2"
                + "      pi(x1)     pi(x2)");
        double[] r = goldenMax(Golden::profit, 2.0, 10.0, 1e-4, true);
        double p = r[0];
        System.out.println(f("golden section: p = %.5f after %d steps,"
                + " %d evaluations", p, (int) r[1], (int) r[2]));
        System.out.println(f("exact optimum : p = %.5f (from pi'(p) = 0)", C + S));
        System.out.println(f("cups %.1f, profit %.3f thousand rupiah a day",
                cups(p), profit(p)));
        double best = 2.0;
        for (int i = 0; i <= 160; i++) {           // every Rp 50
            double q = 2.0 + 0.05 * i;
            if (profit(q) > profit(best)) best = q;
        }
        System.out.println(f("grid of 161 prices: best p = %.2f, profit %.3f",
                best, profit(best)));
    }
}
```

13.4.2 Newton's method with and without a safeguard

Listing 13.3 codes the derivatives of Equation (13.8) and runs plain Newton twice, from Rp 3,000 and from Rp 8,000, then the safeguarded version on the bracket [2,9]. In `safeguarded`, the sign of π' updates the bracket first; the Newton point is accepted only if the curvature is negative and the point lies strictly inside the bracket. When the curvature is not negative, the expression `a - 1.0` is a deliberately impossible point that forces the bisection branch.

Listing 13.3 `newton1d.py`: Newton's method for the best price, plain and safeguarded.

```
"""Newton's method for the best tea price, with and without a safeguard.
Usage: python newton1d.py"""
import math

A, S, C = 600.0, 2.5, 1.5


def d1(p):
    """pi'(p): the slope of the profit curve."""
    return A * math.exp(-p / S) * (1.0 - (p - C) / S)


def d2(p):
    """pi''(p): the curvature of the profit curve."""
    return A * math.exp(-p / S) * ((p - C) / (S * S) - 2.0 / S)


def newton(p, steps):
    print(f"plain Newton from p0 = {p:.2f}")
    for k in range(1, steps + 1):
        p = p - d1(p) / d2(p)
        print(f"  {k:2d}  p = {p:12.8f}  error = {abs(p - (C + S)):.3e}")


def safeguarded(a, b, tol):
    """Newton on pi' = 0 inside [a, b]; bisect when Newton would leave the
    bracket or when the curvature says 'minimum' (pi'' >= 0)."""
    p, k = (a + b) / 2.0, 0
    while b - a > tol and k < 60:
        k += 1
        g, h = d1(p), d2(p)
        if g == 0.0:              # landed exactly on the peak
            break
        if g > 0.0:               # still climbing: the peak is to the right
            a = p
        else:
            b = p
        q = p - g / h if h < 0.0 else a - 1.0
        how = "newton"
        if not (a < q < b):       # unsafe step: halve the bracket instead
            q, how = (a + b) / 2.0, "bisect"
        if abs(q - p) < tol:
            p = q
            break
        p = q
        print(f"  {k:2d}  {how}  p = {p:12.8f}  bracket [{a:.5f}, {b:.5f}]")
```

Listing 13.3 (Continued) *newton1d.py: Newton's method for the best price, plain and safeguarded.*

```
    return p, k


def main():
    newton(3.0, 5)
    newton(8.0, 5)
    print("safeguarded Newton on [2, 9]")
    p, k = safeguarded(2.0, 9.0, 1e-10)
    print(f"p = {p:.10f} after {k} steps")


if __name__ == "__main__":
    main()
```

Listing 13.4 translates it line by line into Java.

Listing 13.4 `Newton1D.java`: the same in Java.

```
// Newton's method for the best tea price, with and without a safeguard.
// Usage: java Newton1D
import java.util.Locale;

public class Newton1D {
    static final double A = 600.0, S = 2.5, C = 1.5;

    /** pi'(p): the slope of the profit curve. */
    static double d1(double p) {
        return A * Math.exp(-p / S) * (1.0 - (p - C) / S);
    }

    /** pi''(p): the curvature of the profit curve. */
    static double d2(double p) {
        return A * Math.exp(-p / S) * ((p - C) / (S * S) - 2.0 / S);
    }

    static void out(String fmt, Object... xs) {
        System.out.println(String.format(Locale.ROOT, fmt, xs));
    }

    static void newton(double p, int steps) {
        out("plain Newton from p0 = %.2f", p);
        for (int k = 1; k <= steps; k++) {
            p = p - d1(p) / d2(p);
            out("  %2d  p = %12.8f  error = %.3e", k, p, Math.abs(p - (C + S)));
        }
    }

    /** Newton on pi' = 0 inside [a, b]; bisect when Newton would leave the
     *  bracket or when the curvature says 'minimum' (pi'' >= 0). */
    static double[] safeguarded(double a, double b, double tol) {
        double p = (a + b) / 2.0;
        int k = 0;
        while (b - a > tol && k < 60) {
            k++;
            double g = d1(p), h = d2(p);
            if (g == 0.0) break;              // landed exactly on the peak
            if (g > 0.0) a = p; else b = p;   // climbing: the peak is to the right
            double q = h < 0.0 ? p - g / h : a - 1.0;
            String how = "newton";
            if (!(a < q && q < b)) {          // unsafe step: halve the bracket
                q = (a + b) / 2.0;
                how = "bisect";
            }
            if (Math.abs(q - p) < tol) { p = q; break; }
```

Listing 13.4 (Continued) *Newton1D.java: the same in Java.*

```
            p = q;
            out("  %2d  %s  p = %12.8f  bracket [%.5f, %.5f]", k, how, p, a, b);
        }
        return new double[] {p, k};
    }

    public static void main(String[] args) {
        newton(3.0, 5);
        newton(8.0, 5);
        System.out.println("safeguarded Newton on [2, 9]");
        double[] r = safeguarded(2.0, 9.0, 1e-10);
        out("p = %.10f after %d steps", r[0], (int) r[1]);
    }
}
```

13.4.3 Two prices: descent and Newton

Listing 13.5 is the heart of the chapter. `demand`, `f`, `grad` and `hess` code Equation (13.12) and its derivatives; `descent` is Equation (13.13) with a fixed step and a guard that stops when the numbers blow up; `backtracking` adds the Armijo test Equation (13.16); `newton` solves the 2×2 system of Equation (13.17) by Cramer's rule. At the end the program computes the eigenvalues of the Hessian at the optimum from the trace and determinant (for a symmetric 2×2 matrix they are $\frac{t}{2} \pm \sqrt{t^2/4 - d}$), the condition number and the step limits of Equation (13.14) and Equation (13.15).

Listing 13.5 `twoprices.py`: gradient descent with fixed and backtracking steps, and Newton's method, for two prices.

```
"""Two prices at once: tea (p1) and fried snacks (p2).
Gradient descent with a fixed step, with backtracking, and Newton's method.
Usage: python twoprices.py"""
import math

C1, C2 = 1.5, 0.6          # cost of a cup of tea and of a snack (thousand Rp)


def demand(p1, p2):
    """Cups of tea and snacks sold in a day at prices p1 and p2."""
    tea = 600.0 * math.exp(-p1 / 2.5 - 0.10 * p2)
    snack = 900.0 * math.exp(-p2 / 1.0 - 0.08 * p1)
    return tea, snack


def f(p):
    """The loss to minimize: minus the daily profit."""
    tea, snack = demand(p[0], p[1])
    return -((p[0] - C1) * tea + (p[1] - C2) * snack)


def grad(p):
    tea, snack = demand(p[0], p[1])
    m1, m2 = p[0] - C1, p[1] - C2
    g1 = tea - m1 * tea / 2.5 - 0.08 * m2 * snack
    g2 = snack - 0.10 * m1 * tea - m2 * snack
    return [-g1, -g2]


def hess(p):
    tea, snack = demand(p[0], p[1])
    m1, m2 = p[0] - C1, p[1] - C2
    h11 = -2.0 * tea / 2.5 + m1 * tea / 6.25 + 0.0064 * m2 * snack
    h12 = -0.10 * tea + 0.04 * m1 * tea - 0.08 * snack + 0.08 * m2 * snack
    h22 = -2.0 * snack + 0.01 * m1 * tea + m2 * snack
    return [[-h11, -h12], [-h12, -h22]]


def norm(v):
    return math.sqrt(v[0] * v[0] + v[1] * v[1])
```

Listing 13.5 (Continued) *twoprices.py: gradient descent with fixed and backtracking steps, and Newton's method, for two prices.*

```
def descent(p, alpha, tol=1e-6, kmax=2000):
    """Fixed-step gradient descent; returns (point, steps) or None."""
    for k in range(1, kmax + 1):
        g = grad(p)
        p = [p[0] - alpha * g[0], p[1] - alpha * g[1]]
        if not all(math.isfinite(x) for x in p) or norm(p) > 1e6:
            return None, k
        if norm(g) < tol:
            return p, k
    return p, kmax


def backtracking(p, tol=1e-6, kmax=2000):
    """Gradient descent where each step is halved until f drops enough."""
    tries = 0
    for k in range(1, kmax + 1):
        g = grad(p)
        gg = g[0] * g[0] + g[1] * g[1]
        if math.sqrt(gg) < tol:
            return p, k - 1, tries
        alpha, fp = 1.0, f(p)
        while True:                          # Armijo condition, c = 1e-4
            tries += 1
            q = [p[0] - alpha * g[0], p[1] - alpha * g[1]]
            if f(q) <= fp - 1e-4 * alpha * gg:
                break
            alpha /= 2.0
        p = q
    return p, kmax, tries


def newton(p, tol=1e-10, kmax=50, show=True):
    for k in range(1, kmax + 1):
        g, h = grad(p), hess(p)
        det = h[0][0] * h[1][1] - h[0][1] * h[1][0]
        s1 = (h[1][1] * g[0] - h[0][1] * g[1]) / det    # solve H s = g
        s2 = (h[0][0] * g[1] - h[1][0] * g[0]) / det
        p = [p[0] - s1, p[1] - s2]
        if show:
            print(f"  {k:2d}  p1 = {p[0]:.10f}  p2 = {p[1]:.10f}"
                  f"  |grad| = {norm(grad(p)):.3e}")
        if norm(grad(p)) < tol:
            return p, k
    return p, kmax
```

Listing 13.5 (Continued) *twoprices.py: gradient descent with fixed and backtracking steps, and Newton's method, for two prices.*

```
def main():
    p0 = [3.0, 1.0]
    g, h = 1e-5, grad(p0)
    num = [(f([3.0 + g, 1.0]) - f([3.0 - g, 1.0])) / (2 * g),
           (f([3.0, 1.0 + g]) - f([3.0, 1.0 - g])) / (2 * g)]
    print(f"gradient at (3, 1), formula           ({h[0]:.6f}, {h[1]:.6f})")
    print(f"gradient at (3, 1), central difference"
          f" ({num[0]:.6f}, {num[1]:.6f})")
    for alpha in (0.002, 0.008, 0.012, 0.020):
        p, k = descent(p0, alpha)
        if p is None:
            print(f"fixed step {alpha:.3f}: diverged after {k} steps")
        elif k == 2000:
            print(f"fixed step {alpha:.3f}: no convergence in {k} steps,"
                  f" |grad| = {norm(grad(p)):.3f}")
        else:
            print(f"fixed step {alpha:.3f}: p = ({p[0]:.6f}, {p[1]:.6f})"
                  f" after {k} steps")
    p, k, tries = backtracking(p0)
    print(f"backtracking  : p = ({p[0]:.6f}, {p[1]:.6f}) after {k} steps,"
          f" {tries} trial points")
    print("Newton from (3, 1):")
    p, k = newton(p0)
    tea, snack = demand(p[0], p[1])
    print(f"best prices: tea Rp {1000 * p[0]:.0f}, snack Rp {1000 * p[1]:.0f}")
    print(f"sales: {tea:.1f} cups, {snack:.1f} snacks;"
          f" profit {-f(p):.3f} thousand rupiah a day")
    h = hess(p)
    tr, det = h[0][0] + h[1][1], h[0][0] * h[1][1] - h[0][1] * h[1][0]
    lo = tr / 2 - math.sqrt(tr * tr / 4 - det)
    hi = tr / 2 + math.sqrt(tr * tr / 4 - det)
    print(f"Hessian eigenvalues {lo:.4f} and {hi:.4f}:"
          f" condition number {hi / lo:.3f}")
    print(f"step limit 2/lambda_max = {2 / hi:.5f},"
          f" best fixed step 2/(lo + hi) = {2 / (lo + hi):.5f}")


if __name__ == "__main__":
    main()
```

Listing 13.6 is the Java version. Java has no tuples, so `descent` and `backtracking` return a small `double` array whose last entries carry the step count and a status.

Listing 13.6 `TwoPrices.java`: the same optimizers in Java.

```java
// Two prices at once: tea (p1) and fried snacks (p2).
// Gradient descent with a fixed step, with backtracking, and Newton's method.
// Usage: java TwoPrices
import java.util.Locale;

public class TwoPrices {
    static final double C1 = 1.5, C2 = 0.6;   // cost of a tea and of a snack

    static void out(String fmt, Object... xs) {
        System.out.println(String.format(Locale.ROOT, fmt, xs));
    }

    /** Cups of tea and snacks sold in a day at prices p1 and p2. */
    static double[] demand(double p1, double p2) {
        return new double[] {600.0 * Math.exp(-p1 / 2.5 - 0.10 * p2),
                             900.0 * Math.exp(-p2 / 1.0 - 0.08 * p1)};
    }

    /** The loss to minimize: minus the daily profit. */
    static double f(double[] p) {
        double[] d = demand(p[0], p[1]);
        return -((p[0] - C1) * d[0] + (p[1] - C2) * d[1]);
    }

    static double[] grad(double[] p) {
        double[] d = demand(p[0], p[1]);
        double tea = d[0], snack = d[1], m1 = p[0] - C1, m2 = p[1] - C2;
        double g1 = tea - m1 * tea / 2.5 - 0.08 * m2 * snack;
        double g2 = snack - 0.10 * m1 * tea - m2 * snack;
        return new double[] {-g1, -g2};
    }

    static double[][] hess(double[] p) {
        double[] d = demand(p[0], p[1]);
        double tea = d[0], snack = d[1], m1 = p[0] - C1, m2 = p[1] - C2;
        double h11 = -2.0 * tea / 2.5 + m1 * tea / 6.25 + 0.0064 * m2 * snack;
        double h12 = -0.10 * tea + 0.04 * m1 * tea - 0.08 * snack
                + 0.08 * m2 * snack;
        double h22 = -2.0 * snack + 0.01 * m1 * tea + m2 * snack;
        return new double[][] {{-h11, -h12}, {-h12, -h22}};
    }

    static double norm(double[] v) { return Math.sqrt(v[0] * v[0] + v[1] * v[1]); }
```

Listing 13.6 (Continued) *TwoPrices.java: the same optimizers in Java.*

```
    /** Fixed-step gradient descent; returns {p1, p2, steps, status}. */
    static double[] descent(double[] p, double alpha, double tol, int kmax) {
        for (int k = 1; k <= kmax; k++) {
            double[] g = grad(p);
            p = new double[] {p[0] - alpha * g[0], p[1] - alpha * g[1]};
            if (!Double.isFinite(p[0]) || !Double.isFinite(p[1])
                    || norm(p) > 1e6)
                return new double[] {p[0], p[1], k, -1};
            if (norm(g) < tol) return new double[] {p[0], p[1], k, 1};
        }
        return new double[] {p[0], p[1], kmax, 0};
    }

    /** Gradient descent where each step is halved until f drops enough. */
    static double[] backtracking(double[] p, double tol, int kmax) {
        int tries = 0;
        for (int k = 1; k <= kmax; k++) {
            double[] g = grad(p);
            double gg = g[0] * g[0] + g[1] * g[1];
            if (Math.sqrt(gg) < tol)
                return new double[] {p[0], p[1], k - 1, tries};
            double alpha = 1.0, fp = f(p);
            double[] q;
            while (true) {                          // Armijo condition, c = 1e-4
                tries++;
                q = new double[] {p[0] - alpha * g[0], p[1] - alpha * g[1]};
                if (f(q) <= fp - 1e-4 * alpha * gg) break;
                alpha /= 2.0;
            }
            p = q;
        }
        return new double[] {p[0], p[1], kmax, tries};
    }

    static double[] newton(double[] p, double tol, int kmax) {
        for (int k = 1; k <= kmax; k++) {
            double[] g = grad(p);
            double[][] h = hess(p);
            double det = h[0][0] * h[1][1] - h[0][1] * h[1][0];
            double s1 = (h[1][1] * g[0] - h[0][1] * g[1]) / det;   // solve H s = g
            double s2 = (h[0][0] * g[1] - h[1][0] * g[0]) / det;
            p = new double[] {p[0] - s1, p[1] - s2};
            out("  %2d  p1 = %.10f  p2 = %.10f  |grad| = %.3e",
                    k, p[0], p[1], norm(grad(p)));
            if (norm(grad(p)) < tol) return p;
        }
```

Listing 13.6 (Continued) *TwoPrices.java: the same optimizers in Java.*

```
        return p;
    }

    public static void main(String[] args) {
        double[] p0 = {3.0, 1.0};
        double e = 1e-5;
        double[] h = grad(p0);
        double n1 = (f(new double[] {3.0 + e, 1.0})
                - f(new double[] {3.0 - e, 1.0})) / (2 * e);
        double n2 = (f(new double[] {3.0, 1.0 + e})
                - f(new double[] {3.0, 1.0 - e})) / (2 * e);
        out("gradient at (3, 1), formula           (%.6f, %.6f)", h[0], h[1]);
        out("gradient at (3, 1), central difference (%.6f, %.6f)", n1, n2);
        for (double alpha : new double[] {0.002, 0.008, 0.012, 0.020}) {
            double[] r = descent(p0, alpha, 1e-6, 2000);
            if (r[3] < 0)
                out("fixed step %.3f: diverged after %d steps", alpha, (int) r[2]);
            else if (r[2] == 2000)
                out("fixed step %.3f: no convergence in %d steps, |grad| = %.3f",
                        alpha, (int) r[2], norm(grad(new double[] {r[0], r[1]})));
            else
                out("fixed step %.3f: p = (%.6f, %.6f) after %d steps",
                        alpha, r[0], r[1], (int) r[2]);
        }
        double[] b = backtracking(p0, 1e-6, 2000);
        out("backtracking  : p = (%.6f, %.6f) after %d steps, %d trial points",
                b[0], b[1], (int) b[2], (int) b[3]);
        System.out.println("Newton from (3, 1):");
        double[] p = newton(p0, 1e-10, 50);
        double[] d = demand(p[0], p[1]);
        out("best prices: tea Rp %.0f, snack Rp %.0f", 1000 * p[0], 1000 * p[1]);
        out("sales: %.1f cups, %.1f snacks; profit %.3f thousand rupiah a day",
                d[0], d[1], -f(p));
        double[][] hs = hess(p);
        double tr = hs[0][0] + hs[1][1];
        double det = hs[0][0] * hs[1][1] - hs[0][1] * hs[1][0];
        double lo = tr / 2 - Math.sqrt(tr * tr / 4 - det);
        double hi = tr / 2 + Math.sqrt(tr * tr / 4 - det);
        out("Hessian eigenvalues %.4f and %.4f: condition number %.3f",
                lo, hi, hi / lo);
        out("step limit 2/lambda_max = %.5f, best fixed step 2/(lo + hi) = %.5f",
                2 / hi, 2 / (lo + hi));
    }
}
```

13.5 Running the Programs

Compile and run from the chapter folder (on Windows type `python` instead of `python3`):

```
cd Chapter13/python
python3 golden.py
python3 newton1d.py
python3 twoprices.py
cd ../java
javac -d out *.java
java -cp out Golden
java -cp out Newton1D
java -cp out TwoPrices
```

The golden-section search prints:

```
  k         a         b     b - a        x1        x2     pi(x1)     pi(x2)
  1   2.00000   6.94427   4.94427   3.88854   5.05573   302.5347   282.3644
  2   2.00000   5.05573   3.05573   3.16718   3.88854   281.7984   302.5347
  3   3.16718   5.05573   1.88854   3.88854   4.33437   302.5347   300.3659
  4   3.16718   4.33437   1.16718   3.61301   3.88854   298.8193   302.5347
  5   3.61301   4.33437   0.72136   3.88854   4.05883   302.5347   302.7622
  6   3.88854   4.33437   0.44582   4.05883   4.16408   302.7622   302.2204
 24   3.99998   4.00006   0.00008   4.00001   4.00003   302.8448   302.8448
golden section: p = 4.00002 after 24 steps, 26 evaluations
exact optimum : p = 4.00000 (from pi'(p) = 0)
cups 121.1, profit 302.845 thousand rupiah a day
grid of 161 prices: best p = 4.00, profit 302.845
```

Figure 13.1 draws the profit curve and the first seven intervals. The interval shrinks by the same factor at every step, so its length falls from 8 to $8 \times 0.618^{24} = 7.7 \times 10^{-5}$ after 24 steps, below the tolerance 10^{-4}. The answer, Rp 4,000, earns 302.8 thousand rupiah a day with 121 cups: about Rp 32,000 a day more than her present price and Rp 19,000 more than her neighbor's. The grid of 161 prices finds the same answer to the nearest Rp 50 but needs six times as many evaluations, and to reach the golden section's accuracy it would need 80,000.

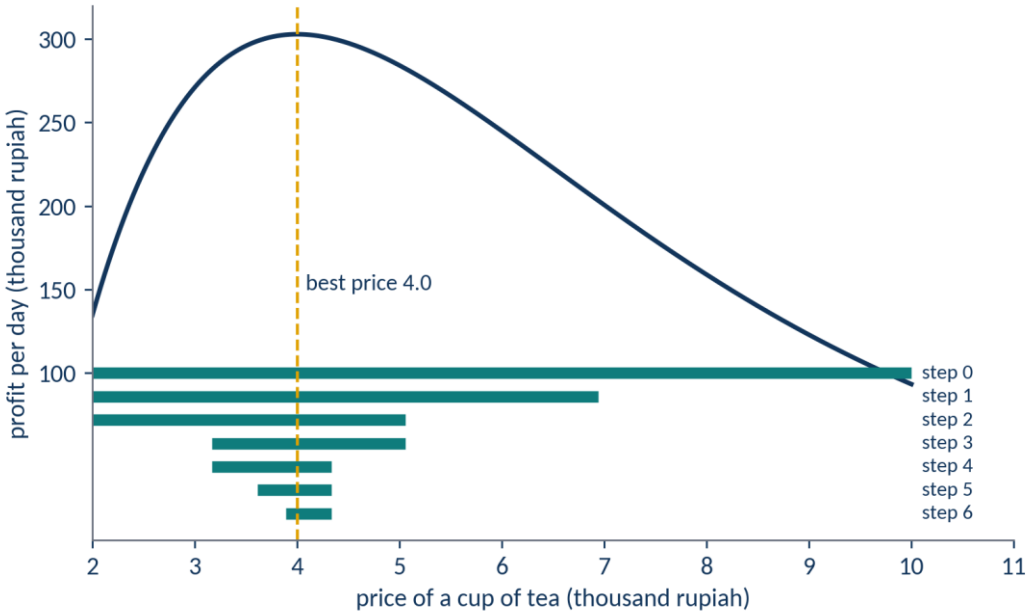


Figure 13.1 Mbak Rina's daily profit and the golden-section intervals of steps 0 to 6: each bar is 0.618 times as long as the one above it, and all of them contain the best price.

Notice in the output how at step 5 the interval [3.61,4.33] still contains the peak even though the probe values 302.53 and 302.76 differ by less than a tenth of a percent. Close to the top the curve is flat, exactly as Equation (13.6) warned: at the end, both probes show 302.8448 to four decimals. The method decides from the digits that remain, which is why a tolerance far below 10^{-8} would be meaningless.

Newton's method prints:

```
plain Newton from p0 = 3.00
   1  p =   3.71428571  error = 2.857e-01
   2  p =   3.97069597  error = 2.930e-02
   3  p =   3.99966049  error = 3.395e-04
   4  p =   3.99999995  error = 4.610e-08
   5  p =   4.00000000  error = 8.882e-16
plain Newton from p0 = 8.00
   1  p =  14.66666667  error = 1.067e+01
   2  p =  17.93197279  error = 1.393e+01
   3  p =  20.97868508  error = 1.698e+01
   4  p =  23.91035411  error = 1.991e+01
   5  p =  26.76933590  error = 2.277e+01
safeguarded Newton on [2, 9]
   1  bisect  p =   3.75000000  bracket [2.00000, 5.50000]
   2  newton  p =   3.97727273  bracket [3.75000, 5.50000]
   3  newton  p =   3.99979525  bracket [3.97727, 5.50000]
   4  newton  p =   3.99999998  bracket [3.99980, 5.50000]
   5  newton  p =   4.00000000  bracket [4.00000, 5.50000]
p = 4.0000000000 after 6 steps
```

The first run is Equation (13.9) in action: the errors 2.9×10^{-1}, 2.9×10^{-2}, 3.4×10^{-4}, 4.6×10^{-8} each roughly square the one before, and the fifth step reaches the rounding level. The second run starts beyond the inflection point at 6.5 and walks away forever, each step adding about three thousand rupiah to a price at which nobody buys tea. The safeguarded run starts from the midpoint 5.5, where the curvature is negative but the Newton point would fall outside the bracket, so it bisects once to 3.75; from there every Newton step is accepted and the answer is found in six steps.

The two-price program prints:

```
gradient at (3, 1), formula           (-57.073365, -131.739616)
gradient at (3, 1), central difference (-57.073365, -131.739616)
fixed step 0.002: p = (3.769928, 1.435197) after 166 steps
fixed step 0.008: p = (3.769928, 1.435197) after 35 steps
fixed step 0.012: no convergence in 2000 steps, |grad| = 43.533
fixed step 0.020: no convergence in 2000 steps, |grad| = 212.849
backtracking  : p = (3.769928, 1.435197) after 35 steps, 294 trial points
Newton from (3, 1):
   1  p1 = 3.5666834402  p2 = 1.2919630099  |grad| = 3.321e+01
   2  p1 = 3.7512263928  p2 = 1.4160375865  |grad| = 3.750e+00
   3  p1 = 3.7697253414  p2 = 1.4348269754  |grad| = 6.884e-02
   4  p1 = 3.7699275219  p2 = 1.4351964841  |grad| = 2.473e-05
   5  p1 = 3.7699275577  p2 = 1.4351966188  |grad| = 3.188e-12
best prices: tea Rp 3770, snack Rp 1435
sales: 115.1 cups, 158.5 snacks; profit 393.533 thousand rupiah a day
Hessian eigenvalues 49.3370 and 182.0565: condition number 3.690
step limit 2/lambda_max = 0.01099, best fixed step 2/(lo + hi) = 0.00864
```

The gradient check agrees; the four fixed steps behave as Equation (13.14) and Equation (13.15) predicted, 166 steps for the timid step 0.002, 35 for 0.008, and no convergence for 0.012 and 0.020, both above the limit 0.01099. Backtracking converges in 35 steps without being told any limit. Newton reaches the answer in five steps, and its gradient lengths 33, 3.75, 0.069, 2.5×10^{-5}, 3.2×10^{-12} show quadratic convergence in two variables.

Figure 13.2 shows the paths on a map of the profit. Newton's path cuts across the contour lines almost straight to the top. Gradient descent with step 0.008 first overshoots in the steep direction, the snack price, then follows the valley in a few zigzags. With step 0.012 the iteration bounces up and down across the valley in the steep direction forever: the linear analysis predicts growing oscillations, and the curve's nonlinearity turns them into a cycle that neither grows nor dies.

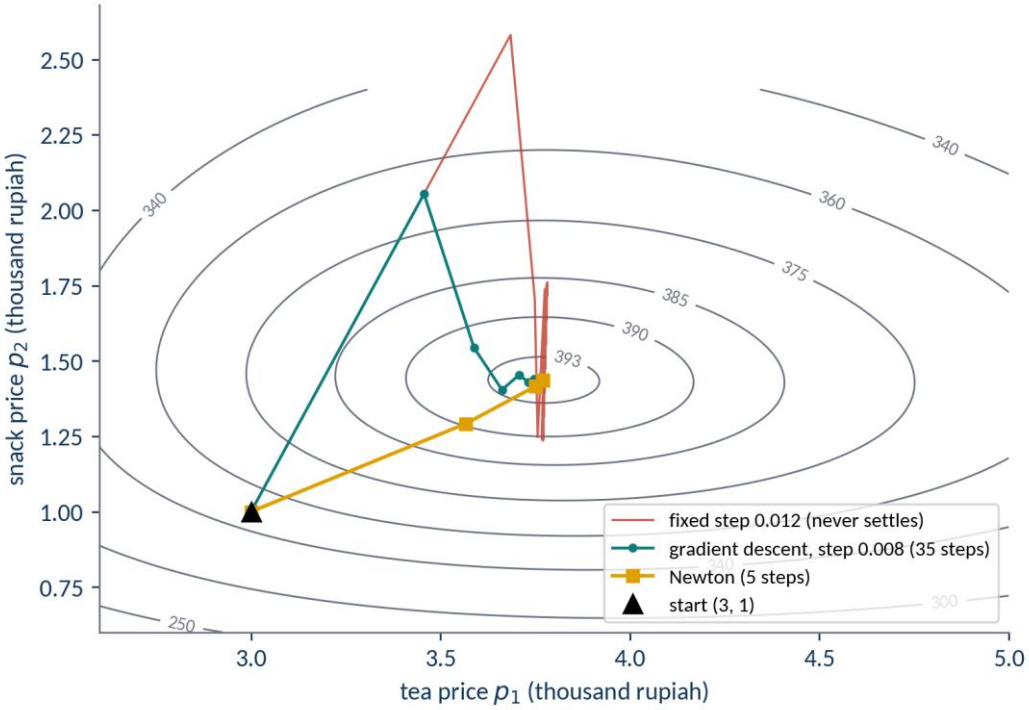


Figure 13.2 Contours of the daily profit (thousand rupiah) over the two prices, with the paths of gradient descent (step 0.008), a fixed step above the stability limit (0.012), and Newton's method, all starting from (3, 1).

The answer for Mbak Rina: tea at Rp 3,770 and gorengan at Rp 1,435, selling 115 cups and 158 snacks for 393.5 thousand rupiah a day. In practice she will round to Rp 3,800 and Rp 1,500, and the flatness of the top tells her that such rounding costs almost nothing: Equation (13.11) with the Hessian's eigenvalues shows that moving the prices by Rp 50 changes the profit by well under a thousand rupiah a day.

13.6 Comparing with SciPy

SciPy's `scipy.optimize` module [4] provides the same families of methods. **Listing 13.7** calls them on the same functions, imported from our own files, so that only the optimizer differs. It was run with Python 3.13, NumPy 2.5.3 and SciPy 1.18.1, the current releases in September 2026 [8], [9]; `requirements.txt` in the chapter folder lists them.

Listing 13.7 `check_scipy.py`: the same problems solved by SciPy's optimizers.

```
"""Compare the chapter's optimizers with SciPy's.
Usage: python check_scipy.py   (needs numpy and scipy)"""
import numpy as np
from scipy.optimize import minimize, minimize_scalar
from golden import profit
from twoprices import f, grad, hess

r = minimize_scalar(lambda p: -profit(p), bracket=(2.0, 3.0, 10.0),
                    method="golden", tol=1e-8)
print(f"SciPy golden : p = {r.x:.8f}, {r.nfev} evaluations")
r = minimize_scalar(lambda p: -profit(p), bounds=(2.0, 10.0),
                    method="bounded", options={"xatol": 1e-8})
print(f"SciPy Brent  : p = {r.x:.8f}, {r.nfev} evaluations")
x0 = np.array([3.0, 1.0])
for m in ("BFGS", "Newton-CG", "Nelder-Mead"):
    kw = {"jac": grad} if m == "BFGS" else {}
    if m == "Newton-CG":
        kw = {"jac": grad, "hess": hess}
    r = minimize(f, x0, method=m, **kw)
    print(f"SciPy {m:11s}: p = ({r.x[0]:.6f}, {r.x[1]:.6f}),"
          f" {r.nit} iterations, {r.nfev} evaluations")
```

```
SciPy golden : p = 3.99999998, 45 evaluations
SciPy Brent  : p = 4.00000000, 11 evaluations
SciPy BFGS       : p = (3.769928, 1.435197), 9 iterations, 11 evaluations
SciPy Newton-CG  : p = (3.769928, 1.435197), 5 iterations, 6 evaluations
SciPy Nelder-Mead: p = (3.769919, 1.435177), 35 iterations, 67 evaluations
```

SciPy's golden-section search agrees with ours; it uses more evaluations because we asked it for a tolerance of 10^{-8} instead of 10^{-4}. The `bounded` method, which is Brent's combination of golden section with parabolic interpolation [5], needs only 11 evaluations: once the peak is bracketed, a parabola through three points lands very close to it, much as Newton's step does, without needing derivatives. BFGS, which learns the Hessian from gradients, needs 9 iterations; Newton-CG, given our Hessian, needs 5, like our Newton. Nelder–Mead, a derivative-free method that moves a triangle of three points downhill [10], needs 67 evaluations and is accurate only to about four decimals with its default tolerances. The lesson carries over to Java, where no standard library optimizer exists: a short, well-tested golden section, a safeguarded Newton and a gradient method with backtracking cover a great many practical problems.

13.7 Making It Better

Three improvements turn the programs into tools that behave well on problems you have not seen.

- **Bracket first, then search.** Golden section needs an interval that contains the peak. When none is known, start at a reasonable guess, step to the right, and keep doubling the step while the function keeps rising; the last three points bracket the peak. SciPy's `bracket` function does exactly this.
- **Scale the variables.** The condition number in Equation (13.15) depends on the units. Had Mbak Rina written her prices in rupiah instead of thousands, and her snack price in hundreds, the Hessian's entries would differ by factors of a hundred and κ would grow accordingly. Choosing units in which all variables have the same size is free and often speeds gradient descent by orders of magnitude; Chapter 14 does it systematically for machine learning.
- **Respect the limits of the real problem.** A price cannot fall below cost and cannot exceed what the market bears. The simplest way to keep an iteration inside a box $l \le x \le u$ is *projection*: after each step, clip every variable back into its range. For golden section the box is the starting

bracket; for gradient descent, clip after each step. Problems with more complicated constraints, such as a limited budget shared among products, need the methods of constrained optimization, Lagrange multipliers and their descendants [6], [11].

13.8 Using It for Real

Mbak Rina's method works for any seller who can change a price and count what sells. Here is how to use it on your own stall, shop or service.

- **Keep price notes for a few weeks.** Try three to five prices, each on several comparable days (the same weekdays, no holidays, similar weather), and write down the number sold. Changing the price only on rainy days, or only at the end of the month, mixes the effect of the price with the effect of the day, and the curve you fit will be wrong. Chapter 8 fits the exponential curve Equation (13.1) as a straight line: $\ln D = \ln A - p/s$, so a least-squares line through the points $(p, \ln D)$ gives s from its slope.
- **Count only the costs that grow with sales.** In Equation (13.2), c is the cost of one more unit: ingredients, packaging, the commission of a delivery app. Rent, the stall permit and your own wage for the day are the same whatever you sell; subtracting them lowers the whole profit curve but does not move its peak, so they do not change the best price. They do decide whether the business is worth running at all.
- **Look at the curve before you search.** Plot the fitted profit over the range of prices you would consider, or evaluate it on a coarse grid. If it has one peak, bracket it and run the golden section of **Pseudocode 13.1**; if it has two (it can, when demand drops sharply at a "psychological" price such as Rp 5,000), search each bracket separately and compare.
- **Round, then test.** Round the optimum to a price that is easy to pay in cash. The flat top means that rounding costs little, and the few hundred rupiah of difference is worth the convenience. Then use the new price for two weeks, keep counting, and refit: people react to a new price for a while before they settle, and the first days are not the last word.

A second example shows how little changes for a different business. Bu Wati runs a *laundry kiloan* (washing priced per kilogram). Detergent, water, electricity and plastic bags cost her about Rp 3,500 per kilogram. Her notes over six weeks, at prices from Rp 6,000 to Rp 9,000, fit the exponential curve with a price scale of $s = 4.0$ thousand rupiah: every Rp 4,000 more per kilogram would lose her about 63% of her customers. The analysis of Section 13.3.2 gives the best price at once, without any search: $p^* = c + s = 3.5 + 4.0 = 7.5$ thousand, Rp 7,500 per kilogram. If she also offers an express service, whose customers react less to price (a larger s) but which costs more electricity per kilogram (a larger c), the two services form a two-price problem like Mbak Rina's tea and gorengan, and the programs of Section 13.4.3 apply with her numbers in `demand`.

The model advises; the owner decides. It knows nothing about a new competitor across the street, a regular customer who would be offended by a price rise, or the goodwill of a round number. Those are the owner's knowledge, and the right use of the computation is to show her what each price is likely to earn, so that she can weigh the rest herself.

13.9 Numerical Methods in Practice: Profitina

Optimization appears in Profitina's AI Visibility platform wherever a setting trades one good thing against another. The clearest example is the automatic re-scan. The platform re-scans each brand's website at a cadence that depends on the subscription tier and keeps a weekly score

trend for the historical dashboard [12]. Scanning more often gives fresher scores and quicker alerts; it also costs server time and puts load on the customer's website. As a function of the interval between scans, the total "cost" (staleness plus load) has the shape of Mbak Rina's profit turned upside down: too short an interval wastes work, too long an interval serves stale results, and somewhere between lies a minimum. A one-variable search of the golden-section kind is the natural tool for choosing such a setting from a cost model, and the flatness of the minimum is good news, because a round number near it (daily, weekly) loses almost nothing.

The Content Engine's quality gate is an optimization of a different kind. A draft written by the local Qwen2.5 model is scored against a built-in template, and the better of the two is served; after the September 2026 revision the AI draft wins in 5 of 6 benchmark cases [12]. Choosing the best of a small set of candidates by a score is optimization over a discrete set, where no gradient exists and the method is simply to evaluate and compare. And behind the PROFIT utility coin sits a decaying emission curve with burning; tuning the parameters of such a curve so that supply and demand stay in a sensible range is, in the end, the minimization of a loss over a few continuous parameters, the problem of Section 13.3.4.

13.10 Common Mistakes, and How to Avoid Them

- **Maximizing when the code minimizes.** Most libraries, including SciPy, minimize. Pass $-\pi$, not π, and remember to negate the reported value back.
- **Golden section on a function with several peaks.** The method keeps a peak inside the bracket, not the highest. Plot the function or evaluate it on a coarse grid first, and bracket the peak you want.
- **Asking for too much accuracy.** Near an optimum the function is flat, and Equation (13.6) limits the achievable accuracy of the location to about $\sqrt{\epsilon_{\text{mach}}}$ relative. A tolerance of 10^{-12} makes the method compare rounding noise.
- **Trusting Newton far from the answer.** Newton's step goes to the stationary point of the local parabola, which can be a minimum when you want a maximum, or far away beyond an inflection point. Check the sign of the curvature and keep a bracket.
- **A step size copied from another problem.** The safe step depends on the largest curvature, Equation (13.14); a step that works for one data set diverges on another. Use backtracking, or estimate $\lambda_{\max}$ with the power method of Chapter 12.
- **An unchecked gradient.** A wrong sign in one partial derivative sends gradient descent uphill and Newton in circles. Compare with central differences at one or two points before any long run.
- **Stopping on a small step instead of a small gradient.** With a tiny α the steps are small even far from the answer. Stop on the gradient's length (relative to its starting length) or on the change of the loss, and always cap the number of steps.

13.11 Chapter Summary

- An optimization problem asks for the point where an objective is smallest or largest; maximizing f is minimizing $-f$. At a smooth interior optimum the derivative or gradient is zero, and the second derivative or Hessian tells a minimum from a maximum.
- Golden-section search brackets the peak of a unimodal function and shrinks the bracket by $r = (\sqrt{5}-1)/2$ per evaluation (Equation (13.4)); its number of steps is known in advance (Equation (13.5)), and its accuracy is limited by the flatness of the peak (Equation (13.6)).

- Newton's method for optimization, $x_{k+1} = x_k - f'(x_k)/f''(x_k)$, converges quadratically near the optimum; for Mbak Rina's profit the error obeys $e_{k+1} = -e_k^2/(s - e_k)$ exactly, which also shows the runaway beyond the inflection point. A bracket with bisection fallback makes it safe.
- Gradient descent converges when $\alpha < 2/\lambda_{\max}$ and, with the best fixed step, at the rate $(\kappa - 1)/(\kappa + 1)$ set by the Hessian's condition number; backtracking with the Armijo condition chooses safe steps automatically.
- Newton's method in several variables solves $H\mathbf{s} = \nabla f$ at every step and converges quadratically, whatever the condition number, at the cost of second derivatives and a linear solve.
- Mbak Rina's best single price is Rp 4,000 (302.8 thousand rupiah profit a day); with gorengan the best pair is Rp 3,770 and Rp 1,435 (393.5 thousand a day). Java and Python print identical results, and SciPy's optimizers agree.

13.12 Exercises

1. Do two more golden-section steps by hand after the one in Section 13.2.2, and check your intervals against the program's rows 2 and 3.
2. Mbak Rina's supplier raises the cost of a cup to Rp 2,000. Without running any program, what is the new best price, and what is the daily profit there? Explain why the best price rises by exactly the cost increase.
3. Starting plain Newton from $p_0 = 6$, compute the first two iterates with Equation (13.9). Then find the range of starting prices from which plain Newton converges to Rp 4,000.
4. Change `backtracking` so that each line search starts from twice the step accepted in the previous iteration (and from 1 the first time). How many trial points does it need now? Explain why this works.
5. Show that for a quadratic loss $f(\mathbf{x}) = 1/2\,\mathbf{x}^{\mathsf{T}}Q\mathbf{x} - \mathbf{b}^{\mathsf{T}}\mathbf{x}$ with Q symmetric positive definite, Newton's method reaches the minimum in one step from any starting point, and that the minimum solves the linear system $Q\mathbf{x} = \mathbf{b}$.
6. Suppose Mbak Rina measured the snack price in hundreds of rupiah instead of thousands, so that $q_2 = 10\,p_2$. How do the Hessian's entries change, what happens to the condition number, and how many steps would gradient descent with the best fixed step need?

13.13 Solutions to the Exercises

1. After step 1 the interval is [2,6.944272] with $x_1 = 3.888544$ ($\pi = 302.5347$) and $x_2 = 5.055728$ ($\pi = 282.3644$). Step 2: $\pi(x_1) > \pi(x_2)$, so $b = 5.055728$, the new x_2 is the old $x_1 = 3.888544$, and the new $x_1 = 5.055728 - 0.618034 \times 3.055728 = 3.167184$ with $\pi(3.167184) = 1.667184 \times 600\,e^{-1.266874} = 1.667184 \times 169.03 = 281.80$. The interval is [2,5.055728], as in row 2. Step 3: now $\pi(x_1) = 281.80 < \pi(x_2) = 302.53$, so $a = 3.167184$, the new x_1 is 3.888544, and the new $x_2 = 3.167184 + 0.618034 \times 1.888544 = 4.334369$ with $\pi = 2.834369 \times 600\,e^{-1.733748} = 2.834369 \times 105.97 = 300.37$. The interval [3.16718,5.05573] matches row 3.

2. By Equation (13.8) the slope is zero where the margin equals the price scale, $p - c = s$, whatever c is. So $p^* = 2.0 + 2.5 = 4.5$, Rp 4,500. The profit there is $s \cdot Ae^{-(c+s)/s} = 2.5 \times 600\,e^{-1.8} = 1500 \times 0.16530 = 247.9$ thousand rupiah a day, lower than before because every cup costs more. The best margin, s, depends only on how quickly buyers leave when the price rises, not on the cost; a higher cost shifts the whole curve to the right and passes the full increase to the price. (With other demand curves only part of a cost increase is passed on; the exponential curve is special.)

3. At $p_0 = 6$: $m = 4.5$, $e_0 = 2$. Then $e_1 = -4/(2.5 - 2) = -8$, so $p_1 = 4 - 8 = -4$, a negative price! Next $e_2 = -64/(2.5 + 8) = -6.095$, $p_2 = -2.10$; the errors then shrink ($e_3 = -37.15/8.595 = -4.32$, and so on) and the iteration crawls back to 4, but only after passing through prices that make no sense. From Equation (13.9), $|e_{k+1}| < |e_k|$ requires $|e_k| < |s - e_k|$. For $e_k < 0$ this always holds, since $s - e_k = s + |e_k|$; for $0 < e_k$ it holds when $e_k < s - e_k$, that is $e_k < s/2 = 1.25$. So every start with $p_0 < 5.25$ (margin below $1.25 + 2.5 = 3.75$) converges monotonically after the first step; starts between 5.25 and 6.5 converge after a wild detour through negative prices; starts at or above 6.5 diverge. A safeguard is not optional.

4. Keep a variable `last` (initially 0.5) outside the loop, start each search at `alpha = 2.0 * last`, and set `last = alpha` after acceptance. In the Java version the same change needs two lines. The number of trial points falls from 294 to 62, and the number of steps from 35 to 28: the accepted step on this problem is usually about 1/128, so starting from twice the previous one needs one halving, and now and then the doubled step is accepted outright, which shortens the walk. The idea is that successive steps of the same problem have similar good sizes, so the last one is the best first guess; doubling first lets the step grow again when the valley widens.

5. The gradient is $\nabla f = Q\mathbf{x} - \mathbf{b}$ and the Hessian is Q, constant. The Newton step from any $\mathbf{x}_0$ solves $Q\mathbf{s} = Q\mathbf{x}_0 - \mathbf{b}$, so $\mathbf{s} = \mathbf{x}_0 - Q^{-1}\mathbf{b}$ and $\mathbf{x}_1 = \mathbf{x}_0 - \mathbf{s} = Q^{-1}\mathbf{b}$, independent of $\mathbf{x}_0$. At $\mathbf{x}_1$ the gradient is $QQ^{-1}\mathbf{b} - \mathbf{b} = \mathbf{0}$ and Q is positive definite, so it is the minimum, and it solves $Q\mathbf{x} = \mathbf{b}$. This is why minimizing a quadratic and solving a symmetric positive definite linear system are the same problem, and why the iterative solvers of Chapter 5 can be read as optimizers.

6. With $p_2 = q_2/10$, the chain rule gives $\partial f / \partial q_2 = \frac{1}{10}\,\partial f / \partial p_2$ and $\partial^2 f / \partial q_2^2 = \frac{1}{100}\,\partial^2 f / \partial p_2^2$, and the mixed entry is divided by 10. The Hessian at the optimum becomes approximately diag(1,0.1) H diag(1,0.1). Its snack-direction curvature drops from about 180 to about 1.8 while the tea-direction curvature stays near 50, so the eigenvalues become roughly 1.8 and 50, and $\kappa \approx 27$ instead of 3.7. The best fixed-step rate Equation (13.15) becomes $26/28 \approx 0.93$, and the number of steps to gain a factor 10^8 grows from about 34 to about $18.8/0.074 \approx 250$. Same problem, same answer, seven times the work, only because of the units: the reason for scaling in Section 13.7.

References

[1] S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers*, 8th ed. New York, NY, USA: McGraw-Hill Education, 2021.

[2] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical Analysis*, 10th ed. Boston, MA, USA: Cengage Learning, 2016.

[3] J. Kiefer, "Sequential minimax search for a maximum," *Proceedings of the American Mathematical Society*, vol. 4, no. 3, pp. 502–506, 1953.

[4] The SciPy community, "Optimization and root finding (scipy.optimize)," SciPy 1.18.1 documentation, 2026. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/optimize.html

[5] R. P. Brent, *Algorithms for Minimization without Derivatives*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1973; reprinted Mineola, NY, USA: Dover, 2002.

[6] J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd ed. New York, NY, USA: Springer, 2006.

[7] L. Armijo, "Minimization of functions having Lipschitz continuous first partial derivatives," *Pacific Journal of Mathematics*, vol. 16, no. 1, pp. 1–3, 1966.

[8] NumPy Developers, "numpy 2.5.3," Python Package Index, Sep. 6, 2026. [Online]. Available: https://pypi.org/project/numpy/

[9] SciPy Developers, “scipy 1.18.1,” Python Package Index, Aug. 21, 2026. [Online]. Available: https://pypi.org/project/scipy/

[10] J. A. Nelder and R. Mead, “A simplex method for function minimization,” *The Computer Journal*, vol. 7, no. 4, pp. 308–313, 1965.

[11] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge, U.K.: Cambridge University Press, 2004.

[12] Profitina, “Profitina — AI Visibility (GEO) platform,” 2026. [Online]. Available: https://profitina.com

Chapter 14

Numerical Methods for Machine Learning

"Machine learning" sounds like a new kind of computing, but under the hood it is the numerical analysis of the earlier chapters put to work on data: a least-squares fit (Chapter 8) solved with a matrix factorization (Chapter 6), a loss minimized by gradient descent or Newton's method (Chapter 13), eigenvalues that decide how fast the minimization goes (Chapter 12), and floating-point arithmetic (Chapter 1) that decides whether the answer can be trusted at all. This chapter follows an online motorbike-taxi driver who wants two predictions from his own trip log, and uses them to teach the numerical side of learning. By the end of the chapter you will be able to fit a linear model by the normal equations and by QR, and explain with the condition number why the second is safer; train the same model by gradient descent, compute the safe learning rate with the power method, and show why standardizing the inputs speeds training by orders of magnitude; explain what mini-batches, momentum and Adam change; derive logistic regression, its gradient and Hessian, and solve it by Newton's method (IRLS); write the log-loss so that it never overflows; check a gradient with finite differences; recognize when single precision is not enough; and write all of it in Java and Python with identical output, checked against NumPy and SciPy.

14.1 Mas Dimas's Two Questions

Mas Dimas drives a motorbike taxi in Yogyakarta, taking orders through an app from six in the morning until the evening. He has kept every trip of the last two months in a spreadsheet: the trip length in kilometers, whether it was rush hour, whether it rained, how far he had to ride to pick the customer up, how many minutes the trip took, and whether the customer cancelled before he arrived. There are 400 rows.

He has two questions, and both are about money. The first: *how long will this trip take?* The app shows a fare and an estimated time, but he knows the estimate is often wrong in the rain and at rush hour, and a driver who can judge the time can plan his day, his breaks and his fuel. The second: *will this customer cancel?* A cancellation after he has ridden three kilometers to the pickup point costs him fuel and a quarter of an hour for nothing. If he knew which orders were likely to be cancelled, he could accept the others first.

The first question asks for a number and is called *regression*; the second asks for a probability of yes or no and is called *classification*. Machine learning answers both the same way: choose a formula with adjustable numbers, called *weights*, and choose the weights that make the formula agree best with the log. That second step is where all the numerical difficulty lives. In his data the trip length runs from 0.9 to 15 km, rush hour and rain are 0 or 1, and pickup distances run from 0.1 to 4 km; 110 of the 400 orders were cancelled. Those plain facts, the sizes of the numbers and how they relate, will decide whether the computation takes 12 steps or thousands, and whether it gives six correct digits or none.

14.2 The Idea in Plain Words

A *model* is a formula that turns the facts about a trip, its *features*, into a prediction. The simplest model for the trip time is a weighted sum:

$$\hat{y} = w_0 + w_1\,\text{km} + w_2\,\text{rush} + w_3\,\text{rain} = \mathbf{x}^{\mathsf{T}}\mathbf{w}, \tag{14.1}$$

where $\mathbf{x} = (1,\text{km},\text{rush},\text{rain})$ collects the features of one trip (the leading 1 multiplies the constant w_0), and $\mathbf{w} = (w_0, w_1, w_2, w_3)$ the weights. In words: a fixed time for starting and parking, plus so many minutes per kilometer, plus extra minutes for rush hour and for rain.

Learning the weights means choosing them so that the predictions $\hat{y}_i$ are close to the recorded times y_i for all $N = 400$ trips. "Close" is measured by a *loss*; for times the usual choice is half the mean squared error,

$$L(\mathbf{w}) = \frac{1}{2N}\sum_{i=1}^{N}(\mathbf{x}_i^{\mathsf{T}}\mathbf{w} - y_i)^2 = \frac{1}{2N}\,\|\,X\mathbf{w} - \mathbf{y}\,\|^2, \tag{14.2}$$

where the *design matrix* X has one row $\mathbf{x}_i^{\mathsf{T}}$ per trip. This is exactly the least-squares problem of Chapter 8 [1], and learning is exactly the optimization of Chapter 13. What is new in machine learning is scale (millions of rows, millions of weights) and, because of scale, the choice of methods: formulas that are fine for 400 rows break down, and methods that look slow on paper win in practice.

Let us learn a tiny model by hand. Take three trips: 2 km in 10 minutes, 5 km in 17, 8 km in 26, and the model $\hat{y} = w_0 + w_1\,\text{km}$. Setting the derivatives of Equation (14.2) to zero gives two equations (Section 14.3.1 derives them in general):

- $3w_0 + 15w_1 = 10 + 17 + 26 = 53$ (the sums $\sum 1 = 3$ and $\sum\text{km} = 15$),
- $15w_0 + 93w_1 = 2\cdot 10 + 5\cdot 17 + 8\cdot 26 = 313$ (with $\sum\text{km}^2 = 4 + 25 + 64 = 93$).

The determinant is $3\cdot 93 - 15\cdot 15 = 54$, so $w_1 = (3\cdot 313 - 15\cdot 53)/54 = 144/54 = 2.667$ and $w_0 = (93\cdot 53 - 15\cdot 313)/54 = 234/54 = 4.333$. The learned rule: about 4.3 minutes plus 2.7 minutes per kilometer. On Mas Dimas's full log, with rush hour and rain as well, the programs of this chapter find 4.0 minutes plus 2.6 per kilometer, plus 5.7 at rush hour and 5.5 in the rain.

The 2-by-2 matrix $\begin{pmatrix} 3 & 15 \\ 15 & 93 \end{pmatrix}$ on the left already shows the first numerical danger. Its eigenvalues are $48 \pm \sqrt{48^2 - 54} = 95.43$ and 0.566, so its condition number is $95.43/0.566 = 169$. Measure the distance in meters instead of kilometers and the entry 93 becomes 93,000,000 and the condition number about 10^8. The data have not changed; only their units. Center and scale the distance instead, replacing 2, 5, 8 by $-1.22, 0, 1.22$, and the matrix becomes $\begin{pmatrix} 3 & 0 \\ 0 & 3 \end{pmatrix}$ with condition number 1. Units and scaling, invisible in the mathematics, dominate the numerics.

> **KEY INSIGHT:** Three numerical questions decide whether learning works: *How ill-conditioned is the problem?* (it decides the digits you lose and the steps you need); *how are the features scaled?* (it decides the condition number of the loss); and *can any intermediate quantity overflow or cancel?* (it decides whether you get a number at all). The rest of the chapter answers each for Mas Dimas's two questions.

14.3 The Methods, Derived

14.3.1 Least squares: normal equations or QR

The gradient of Equation (14.2) is

$$\nabla L(\mathbf{w}) = \frac{1}{N} X^{\mathsf{T}}(X\mathbf{w} - \mathbf{y}), \qquad \nabla^2 L = \frac{1}{N} X^{\mathsf{T}}X, \tag{14.3}$$

and setting it to zero gives the *normal equations*

$$X^{\mathsf{T}}X\,\mathbf{w} = X^{\mathsf{T}}\mathbf{y}. \tag{14.4}$$

The matrix $X^{\mathsf{T}}X$ is symmetric and, when the columns of X are independent, positive definite, so the Cholesky factorization of Chapter 6 solves Equation (14.4) in about $n^3/3$ operations for n weights, after about Nn^2 operations to form the matrix. This is fast and simple, and it is how the hand example was solved.

The trouble is accuracy. For a rectangular matrix, the condition number $\kappa(X)$ is the ratio of its largest to its smallest *singular value*, the square roots of the eigenvalues of $X^{\mathsf{T}}X$. Hence

$$\kappa(X^{\mathsf{T}}X) = \kappa(X)^2. \tag{14.5}$$

Forming $X^{\mathsf{T}}X$ squares the condition number, and the standard perturbation theory of least squares [2], [3] says that a backward-stable method loses about $\log_{10}\kappa$ decimal digits of the ≈ 16 available in double precision. The normal equations therefore lose about $2\log_{10}\kappa(X)$ digits, while the QR factorization of Chapter 6, which works on X directly, loses about $\log_{10}\kappa(X)$ (plus a term proportional to κ^2 times the relative size of the residual, which is small when the model fits well):

$$\frac{\|\,\hat{\mathbf{w}} - \mathbf{w}\,\|}{\|\,\mathbf{w}\,\|} \lesssim \kappa(X)^2\,\epsilon_{\text{mach}} \text{ (normal equations)}, \qquad \lesssim \kappa(X)\,\epsilon_{\text{mach}} \text{ (QR)}. \tag{14.6}$$

QR by Householder reflections writes $X = QR$ with Q orthogonal and R upper triangular. Orthogonal matrices do not change lengths, so $\|\,X\mathbf{w} - \mathbf{y}\,\| = \|\,R\mathbf{w} - Q^{\mathsf{T}}\mathbf{y}\,\|$, which is minimized by solving the triangular system formed by the top n rows, $R_{1:n}\mathbf{w} = (Q^{\mathsf{T}}\mathbf{y})_{1:n}$. It costs about twice as much as the normal equations and never squares κ [4].

For Mas Dimas's four-feature model $\kappa(X^{\mathsf{T}}X) = 546$, and both methods agree to every printed digit. To see the difference we give the model something harder to swallow: the trip time as a polynomial of high degree in the distance, with columns $1, d, d^2, \ldots, d^{12}$. Powers of numbers between 1 and 15 grow at very different rates, the columns become nearly parallel, and at degree 12 $\kappa(X) \approx 8 \times 10^{16}$. By Equation (14.5) $\kappa(X^{\mathsf{T}}X)$ is then far beyond $1/\epsilon_{\text{mach}} \approx 4.5 \times 10^{15}$: in floating point, $X^{\mathsf{T}}X$ is indistinguishable from a singular matrix, and Cholesky meets a negative number under a square root. Section 14.5 shows exactly that. (A degree-12 polynomial is a bad model of trip time anyway; we use it only because it provokes the numerical failure that, in real work, near-duplicate features provoke silently.)

14.3.2 Gradient descent and the scale of the features

For millions of rows and features, even QR is too expensive, and learning is done by gradient descent as in Section 13.3.4, with the gradient Equation (14.3). Because the Hessian of the least-squares loss is the constant matrix $X^{\mathsf{T}}X/N$, the analysis of Section 13.3.4 applies exactly: the

iteration converges for a learning rate $\alpha < 2/\lambda_{\max}$, and at best by the factor $(\kappa - 1)/(\kappa + 1)$ per step, where now $\lambda_{\max}$ and κ belong to $X^{\mathsf{T}}X/N$.

The largest eigenvalue is easy to get without any library: the power method of Chapter 12 applied to $X^{\mathsf{T}}X/N$. For the raw features it gives $\lambda_{\max} = 77.4$, dominated by the kilometers column (the average of km^2 is about 77); the smallest eigenvalue is about 0.14, so $\kappa \approx 546$. Even at the best learning rate the error shrinks by only $545/547 = 0.9963$ per step, and gaining six digits takes about $6\ln 10/0.0037 \approx 3{,}800$ steps. With the practical rate $\alpha = 0.012$, about $1/\lambda_{\max}$, it is worse: the flattest direction shrinks by $1 - 0.012 \times 0.14 = 0.9983$ per step.

The cure is to *standardize* each feature: subtract its mean μ_j and divide by its standard deviation σ_j,

$$u_{ij} = \frac{x_{ij} - \mu_j}{\sigma_j}. \tag{14.7}$$

After standardization every feature has mean 0 and variance 1, the constant column is orthogonal to the others, and $X^{\mathsf{T}}X/N$ becomes the *correlation matrix* of the features with a 1 in the corner. For Mas Dimas its eigenvalues lie between 0.90 and 1.09: $\kappa = 1.21$, and gradient descent with $\alpha = 0.5$ reduces the error by a factor of about 0.55 per step. The model is the same (the weights for standardized inputs convert back exactly, as the program shows); only the geometry of the loss changed, from a long thin valley to an almost round bowl. This is the single most effective numerical trick in machine learning, and the reason every serious library standardizes or normalizes its inputs [5], [6].

> **EXAMPLE:** The first epoch by hand. Start from $\mathbf{w} = \mathbf{0}$ with standardized inputs. Every prediction is 0, so the prediction errors are $-y_i$ and by Equation (14.3) the gradient is $-\frac{1}{N}X^{\mathsf{T}}\mathbf{y}$. Its first component is minus the mean trip time, -27.50 minutes; the others are minus the average of each standardized feature times the trip time, which is minus the feature's covariance with the time divided by its standard deviation: -10.38 for the distance, -2.69 for rush hour and -2.69 for rain. One step with $\alpha = 0.5$ gives $\mathbf{w} = (13.75, 5.19, 1.35, 1.34)$, and the loss falls from 442.3 to 113.0. The constant has gone exactly halfway to the mean time 27.5, because its direction is uncoupled from the others (Exercise 14.2) and its curvature is 1; each later epoch halves the remaining distance again. The feature weights move toward their final values 10.42 minutes per standard deviation of distance, 2.73 for rush hour and 2.34 for rain at a similar pace, because after standardization all directions have curvatures near 1. With raw inputs the same first step would be dominated by the kilometers column, with its curvature of 77, and the step would have to be forty times smaller.

14.3.3 Mini-batches, momentum and Adam

The gradient Equation (14.3) is an average over all N rows. When N is in the millions, computing it for every single step is wasteful: an average over a random handful of rows points in nearly the same direction at a tiny fraction of the cost. *Stochastic gradient descent* (SGD) with *mini-batches* of size B shuffles the rows at the start of each pass through the data (an *epoch*), cuts them into batches $\mathcal{B}$, and takes one step per batch:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{1}{B} \sum_{i \in \mathcal{B}} (\mathbf{x}_i^{\mathsf{T}}\mathbf{w} - y_i)\, \mathbf{x}_i. \tag{14.8}$$

Because the batch is random, its gradient is an *unbiased* estimate of the full gradient: right on average, wrong in each draw. Far from the minimum the error in each draw hardly matters. Near the minimum the true gradient becomes small but the noise does not, so with a fixed α the weights stop converging and wander in a small cloud around the minimum; the size of the cloud grows with α and with the variance of the per-row gradients, and shrinks with B [6]. To converge fully, SGD must decrease α over time, for example in proportion to $1/t$.

Momentum (the "heavy ball" of Polyak [7]) keeps a running sum of past gradients and steps along it,

$$\mathbf{m} \leftarrow \beta\,\mathbf{m} + \mathbf{g}, \qquad \mathbf{w} \leftarrow \mathbf{w} - \alpha\,\mathbf{m}, \qquad \beta = 0.9. \tag{14.9}$$

Components of the gradient that keep their sign accumulate to about $1/(1-\beta) = 10$ times their size, while components that flip sign from step to step, the zigzag across a narrow valley, cancel. On an ill-conditioned quadratic, well-tuned momentum improves the rate from $(\kappa-1)/(\kappa+1)$ to $\left(\sqrt{\kappa}-1\right)/\left(\sqrt{\kappa}+1\right)$ [5]. Because momentum multiplies the effective step by about 10, its α must be about ten times smaller, which is why the program uses 0.005.

Adam [8] keeps two running averages per weight, of the gradient and of its square, corrects them for starting at zero, and divides one by the square root of the other:

$$\mathbf{m} \leftarrow \beta_1\mathbf{m} + (1-\beta_1)\,\mathbf{g}, \quad \mathbf{v} \leftarrow \beta_2\mathbf{v} + (1-\beta_2)\,\mathbf{g}^2, \quad \mathbf{w} \leftarrow \mathbf{w} - \alpha\,\frac{\mathbf{m}/(1-\beta_1^t)}{\sqrt{\mathbf{v}/(1-\beta_2^t)} + 10^{-8}}, \tag{14.10}$$

with squares and roots taken element by element, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and t the step number. The division gives every weight a step of about α in size regardless of the scale of its gradient, a built-in per-weight rescaling that makes Adam robust to badly scaled features and the default optimizer for neural networks. The price is that its step never shrinks by itself: like SGD, it ends in a cloud around the minimum unless α decreases.

14.3.4 Logistic regression and Newton's method

For the second question, whether an order will be cancelled, a weighted sum $z = \mathbf{x}^{\mathsf{T}}\mathbf{w}$ can be any number, but a probability must lie between 0 and 1. The *logistic* or *sigmoid* function squeezes one into the other:

$$p = \sigma(z) = \frac{1}{1+e^{-z}}, \qquad \sigma'(z) = \sigma(z)\,(1-\sigma(z)). \tag{14.11}$$

It gives 0.5 at $z = 0$, approaches 1 for large z and 0 for very negative z. Here $\mathbf{x} =$ (1,pickup km,rain,rush).

Which weights fit best? The principle of *maximum likelihood* says: those that make the recorded outcomes most probable. If trip i was cancelled ($t_i = 1$) the model gave it probability p_i, and if not ($t_i = 0$) probability $1 - p_i$. Assuming the trips are independent, the probability of the whole log is the product of these numbers, and maximizing a product is the same as minimizing minus its logarithm, averaged over trips: the *log-loss* or *cross-entropy*,

$$L(\mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N}\left[\, t_i \ln p_i + (1-t_i)\ln(1-p_i)\,\right], \qquad p_i = \sigma(\mathbf{x}_i^{\mathsf{T}}\mathbf{w}). \tag{14.12}$$

Using $\ln\sigma(z) = -\ln(1+e^{-z})$ and $\ln(1-\sigma(z)) = -\ln(1+e^{z})$, and writing $\mathrm{softplus}(z) = \ln(1+e^{z})$, a few lines of algebra give the equivalent form

$$L(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}\left[\, \mathrm{softplus}(z_i) - t_i z_i \,\right], \qquad z_i = \mathbf{x}_i^{\mathsf{T}}\mathbf{w}. \tag{14.13}$$

Differentiating with $\mathrm{softplus}'(z) = \sigma(z)$ and $\sigma' = \sigma(1-\sigma)$ gives the gradient and the Hessian:

$$\nabla L = \frac{1}{N}X^{\mathsf{T}}(\mathbf{p}-\mathbf{t}), \qquad H = \frac{1}{N}X^{\mathsf{T}}WX, \qquad W = \mathrm{diag}(p_i(1-p_i)). \tag{14.14}$$

The gradient has the same shape as Equation (14.3): features times prediction errors. The Hessian is $X^{\mathsf{T}}X$ with each row weighted by $p_i(1-p_i) > 0$, so it is positive definite and the log-loss is *convex*: it has one minimum and no traps. Newton's method of Section 13.3.6 applies directly:

$$\frac{1}{N}X^{\mathsf{T}}WX\,\mathbf{s} = \frac{1}{N}X^{\mathsf{T}}(\mathbf{p}-\mathbf{t}), \qquad \mathbf{w} \leftarrow \mathbf{w} - \mathbf{s}. \tag{14.15}$$

Each step is a weighted least-squares problem, solved here by Cholesky, which is why statisticians call the method *iteratively reweighted least squares* (IRLS) [9]. With four weights each step is trivial, and quadratic convergence gives the minimum to machine precision in about six steps. Gradient descent also converges, because $p(1-p) \le 1/4$ gives $H \preccurlyeq X^{\mathsf{T}}X/(4N)$, so any $\alpha < 8/\lambda_{\max}(X^{\mathsf{T}}X/N)$ is safe; for the standardized features that bound is $8/1.226 = 6.5$, and the program uses a cautious 2.0.

Let us predict by hand with the weights the program finds, $\mathbf{w} = (-4.8074, 1.2994, 1.0072, 1.1131)$. An order with a 2 km pickup in the rain outside rush hour has $z = -4.8074 + 2\times 1.2994 + 1.0072 = -1.2014$, and $p = 1/(1+e^{1.2014}) = 1/(1+3.325) = 0.231$: about one chance in four that the ride to the pickup is wasted. Each weight also has a plain meaning: e^{w_j} is the factor by which the *odds* $p/(1-p)$ change when feature j grows by one. With $e^{1.2994} = 3.67$, each extra kilometer of pickup multiplies the odds of a cancellation by 3.7.

14.3.5 Formulas that do not overflow

The textbook log-loss Equation (14.12) hides two traps for floating-point arithmetic (Chapter 1). If $z = -800$ for a trip that was cancelled, $e^{-z} = e^{800}$ overflows (the largest double is about $1.8\times 10^{308} \approx e^{709.8}$), p becomes 0, and $\ln 0 = -\infty$. And if $z = 40$ for a trip that was not cancelled, $e^{-40} \approx 4\times 10^{-18}$ is smaller than half the machine epsilon, $1 + e^{-40}$ rounds to exactly 1, $p = 1$, and $\ln(1-p) = \ln 0$ again. Both are *wrong answers to a well-posed question*: the true losses are 800 and 40.

The form Equation (14.13) avoids both, if softplus itself is computed carefully:

$$\mathrm{softplus}(z) = \max(z, 0) + \mathrm{log1p}(e^{-|z|}), \tag{14.16}$$

where $\mathrm{log1p}(u) = \ln(1+u)$ computed accurately for small u, a function both Java and Python provide. The exponent $-|z|$ is never positive, so $e^{-|z|}$ cannot overflow. The sigmoid

Equation (14.11) is treated the same way: for $z \geq 0$ use $1/(1+e^{-z})$, and for $z < 0$ the algebraically equal $e^{z}/(1+e^{z})$. The same idea, subtract the largest exponent before exponentiating, gives the *log-sum-exp* trick behind every softmax layer of a neural network [10].

14.3.6 Checking gradients, and the width of a float

A gradient written by hand, like Equation (14.14), should be checked against central differences (Chapter 9) before it is trusted: $\partial L/\,\partial w_j \approx [L(\mathbf{w}+h\mathbf{e}_j) - L(\mathbf{w}-h\mathbf{e}_j)]/(2h)$. The truncation error is of order h^2 and the rounding error of order $\epsilon_{\text{mach}}|L|/h$; with $h = 10^{-6}$ both are near 10^{-10}, which is the size of the difference the program reports. A wrong sign or a missing factor would show up as a difference of order 0.01 to 1.

Machine learning often computes in *single precision* (float32), which stores 24 significant bits instead of double precision's 53, or even in 16-bit formats, because narrower numbers move through memory and through graphics processors faster [11]. The unit roundoffs are

$$\epsilon_{32} = 2^{-24} \approx 6.0 \times 10^{-8}, \qquad \epsilon_{64} = 2^{-53} \approx 1.1 \times 10^{-16}, \tag{14.17}$$

about 7 against 16 decimal digits. Weights and gradients tolerate that well, because learning is itself noisy. Sums over many rows, and formulas that subtract nearly equal numbers, do not: in float32 the integers stop being exact at $2^{24} = 16{,}777{,}216$, and the variance formula $\overline{x^2} - \bar{x}^2$ applied to times near 4.3×10^4 minutes subtracts two numbers near 1.9×10^9 whose spacing in float32 is 128. Section 14.6 shows the result. Library code therefore accumulates sums in higher precision, or in pairs (Chapter 15), even when the data are stored in float32 [3].

14.4 The Programs

The programs live in `Chapter14/python` and `Chapter14/java`, and read `Chapter14/data/trips.csv`, which `data/make_trips.py` generates reproducibly. The numerical building blocks are gathered in one module per language, the three programs use them, and every Java program prints exactly what its Python twin prints.

14.4.1 The building blocks

Listing 14.1 contains the CSV reader, Cholesky solving (Chapter 6), the normal equations, Householder QR least squares, mean and standard deviation, RMSE, a small random-number generator, and the stable sigmoid and softplus of Equation (14.16). The generator is SplitMix64, a tiny, well-tested generator that produces the same stream of 64-bit numbers in any language with 64-bit integer arithmetic; Python needs the mask `0xFFFF...` to imitate 64-bit wraparound, Java gets it free. Python's own `random` and Java's `java.util.Random` produce different streams, so neither could be used if the two programs are to shuffle the trips identically. For the same reason the module adds numbers with its own function `add`, a plain left-to-right loop: since Python 3.12 the built-in `sum` of floats compensates for rounding errors (Chapter 15 explains how), which is more accurate but no longer the same arithmetic as Java's loop.

Listing 14.1 `mlcore.py`: data, least squares by normal equations and by QR, a portable random-number generator, and stable logistic functions.

```python
"""Numerical building blocks for Chapter 14: data, least squares, random numbers,
and numerically stable logistic functions. Plain Python, no libraries."""
import math
import os

DATA = os.path.join(os.path.dirname(os.path.abspath(__file__)), "..", "data")


def add(values):
    """Left-to-right sum, the same additions as a Java loop. (Since Python
    3.12 the built-in sum() compensates rounding errors, so it can differ.)"""
    s = 0.0
    for v in values:
        s += v
    return s


def load_trips():
    """Rows of trips.csv as lists of floats (header skipped)."""
    with open(os.path.join(DATA, "trips.csv"), encoding="utf-8") as f:
        next(f)
        return [[float(x) for x in line.split(",")] for line in f if line.strip()]


def cholesky_solve(a, b):
    """Solve A x = b for symmetric positive definite A (Chapter 6)."""
    n = len(a)
    L = [[0.0] * n for _ in range(n)]
    for j in range(n):
        s = a[j][j] - add(L[j][k] * L[j][k] for k in range(j))
        if s <= 0.0:
            raise ArithmeticError(f"not positive definite at column {j}")
        L[j][j] = math.sqrt(s)
        for i in range(j + 1, n):
            L[i][j] = (a[i][j] - add(L[i][k] * L[j][k] for k in range(j))) / L[j][j]
    z = [0.0] * n
    for i in range(n):                                  # L z = b
        z[i] = (b[i] - add(L[i][k] * z[k] for k in range(i))) / L[i][i]
    x = [0.0] * n
    for i in range(n - 1, -1, -1):                      # L^T x = z
        x[i] = (z[i] - add(L[k][i] * x[k] for k in range(i + 1, n))) / L[i][i]
    return x
```

Listing 14.1 (Continued) *mlcore.py: data, least squares by normal equations and by QR, a portable random-number generator, and stable logistic functions.*

```
def normal_equations(X, y):
    """Least squares by forming X^T X w = X^T y and using Cholesky."""
    n = len(X[0])
    a = [[add(r[i] * r[j] for r in X) for j in range(n)] for i in range(n)]
    b = [add(r[i] * t for r, t in zip(X, y)) for i in range(n)]
    return cholesky_solve(a, b)


def qr_solve(X, y):
    """Least squares by Householder QR: never forms X^T X (Chapter 6)."""
    A = [row[:] for row in X]
    c = y[:]
    m, n = len(A), len(A[0])
    for k in range(n):
        norm = math.sqrt(add(A[i][k] * A[i][k] for i in range(k, m)))
        alpha = -norm if A[k][k] >= 0 else norm
        v = [0.0] * m
        v[k] = A[k][k] - alpha
        for i in range(k + 1, m):
            v[i] = A[i][k]
        vv = add(v[i] * v[i] for i in range(k, m))
        if vv == 0.0:
            continue
        for j in range(k, n):                              # A = (I - 2vv^T/v^Tv) A
            s = 2.0 * add(v[i] * A[i][j] for i in range(k, m)) / vv
            for i in range(k, m):
                A[i][j] -= s * v[i]
        s = 2.0 * add(v[i] * c[i] for i in range(k, m)) / vv
        for i in range(k, m):
            c[i] -= s * v[i]
    w = [0.0] * n
    for i in range(n - 1, -1, -1):                         # R w = (Q^T y)[0:n]
        w[i] = (c[i] - add(A[i][j] * w[j] for j in range(i + 1, n))) / A[i][i]
    return w


def mean_std(values):
    mu = add(values) / len(values)
    var = add((v - mu) * (v - mu) for v in values) / len(values)
    return mu, math.sqrt(var)


def rmse(X, y, w):
    e = [add(a * b for a, b in zip(r, w)) - t for r, t in zip(X, y)]
    return math.sqrt(add(v * v for v in e) / len(e))
```

Listing 14.1 (Continued) *mlcore.py: data, least squares by normal equations and by QR, a portable random-number generator, and stable logistic functions.*

```
class Rng:
    """SplitMix64: the same stream of numbers in Python and in Java."""

    def __init__(self, seed):
        self.s = seed & 0xFFFFFFFFFFFFFFFF

    def next64(self):
        self.s = (self.s + 0x9E3779B97F4A7C15) & 0xFFFFFFFFFFFFFFFF
        z = self.s
        z = ((z ^ (z >> 30)) * 0xBF58476D1CE4E5B9) & 0xFFFFFFFFFFFFFFFF
        z = ((z ^ (z >> 27)) * 0x94D049BB133111EB) & 0xFFFFFFFFFFFFFFFF
        return z ^ (z >> 31)

    def below(self, n):
        return self.next64() % n

    def shuffle(self, items):
        for i in range(len(items) - 1, 0, -1):          # Fisher-Yates
            j = self.below(i + 1)
            items[i], items[j] = items[j], items[i]


def sigmoid(z):
    """1 / (1 + e^-z) without overflow for large |z|."""
    if z >= 0:
        return 1.0 / (1.0 + math.exp(-z))
    e = math.exp(z)
    return e / (1.0 + e)


def softplus(z):
    """log(1 + e^z) without overflow: max(z, 0) + log1p(e^-|z|)."""
    return max(z, 0.0) + math.log1p(math.exp(-abs(z)))
```

Listing 14.2 is the Java module. Every sum is accumulated in the same order as in Python, which is what makes the outputs identical to the last digit even for the ill-conditioned polynomial fits: floating-point addition is not associative (Chapter 1), so two programs that add the same numbers in a different order may disagree in the last bits, and an ill-conditioned problem magnifies those bits.

Listing 14.2 `MlCore.java`: the same building blocks in Java.

```java
// Numerical building blocks for Chapter 14: data, least squares, random numbers,
// and numerically stable logistic functions. Plain Java, no libraries.
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;

public class MlCore {
    static final Path DATA = Path.of("..", "data");

    static String f(String fmt, Object... xs) {
        return String.format(Locale.ROOT, fmt, xs);
    }

    /** Rows of trips.csv as arrays of doubles (header skipped). */
    static double[][] loadTrips() throws IOException {
        List<double[]> rows = new ArrayList<>();
        List<String> lines = Files.readAllLines(DATA.resolve("trips.csv"));
        for (String line : lines.subList(1, lines.size())) {
            if (line.isBlank()) continue;
            String[] p = line.split(",");
            double[] r = new double[p.length];
            for (int i = 0; i < p.length; i++) r[i] = Double.parseDouble(p[i]);
            rows.add(r);
        }
        return rows.toArray(new double[0][]);
    }

    /** Solve A x = b for symmetric positive definite A (Chapter 6). */
    static double[] choleskySolve(double[][] a, double[] b) {
        int n = a.length;
        double[][] L = new double[n][n];
        for (int j = 0; j < n; j++) {
            double s = 0.0;
            for (int k = 0; k < j; k++) s += L[j][k] * L[j][k];
            s = a[j][j] - s;
            if (s <= 0.0)
                throw new ArithmeticException(
                        "not positive definite at column " + j);
            L[j][j] = Math.sqrt(s);
            for (int i = j + 1; i < n; i++) {
                double t = 0.0;
                for (int k = 0; k < j; k++) t += L[i][k] * L[j][k];
                L[i][j] = (a[i][j] - t) / L[j][j];
```

Listing 14.2 (Continued) *MlCore.java: the same building blocks in Java.*

```
            }
        }
        double[] z = new double[n], x = new double[n];
        for (int i = 0; i < n; i++) {                       // L z = b
            double t = 0.0;
            for (int k = 0; k < i; k++) t += L[i][k] * z[k];
            z[i] = (b[i] - t) / L[i][i];
        }
        for (int i = n - 1; i >= 0; i--) {                  // L^T x = z
            double t = 0.0;
            for (int k = i + 1; k < n; k++) t += L[k][i] * x[k];
            x[i] = (z[i] - t) / L[i][i];
        }
        return x;
    }

    /** Least squares by forming X^T X w = X^T y and using Cholesky. */
    static double[] normalEquations(double[][] X, double[] y) {
        int n = X[0].length;
        double[][] a = new double[n][n];
        double[] b = new double[n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                double s = 0.0;
                for (double[] r : X) s += r[i] * r[j];
                a[i][j] = s;
            }
            double s = 0.0;
            for (int k = 0; k < X.length; k++) s += X[k][i] * y[k];
            b[i] = s;
        }
        return choleskySolve(a, b);
    }

    /** Least squares by Householder QR: never forms X^T X (Chapter 6). */
    static double[] qrSolve(double[][] X, double[] y) {
        int m = X.length, n = X[0].length;
        double[][] A = new double[m][];
        for (int i = 0; i < m; i++) A[i] = X[i].clone();
        double[] c = y.clone();
        for (int k = 0; k < n; k++) {
            double norm = 0.0;
            for (int i = k; i < m; i++) norm += A[i][k] * A[i][k];
            norm = Math.sqrt(norm);
            double alpha = A[k][k] >= 0 ? -norm : norm;
            double[] v = new double[m];
```

Listing 14.2 (Continued) *MlCore.java: the same building blocks in Java.*

```
            v[k] = A[k][k] - alpha;
            for (int i = k + 1; i < m; i++) v[i] = A[i][k];
            double vv = 0.0;
            for (int i = k; i < m; i++) vv += v[i] * v[i];
            if (vv == 0.0) continue;
            for (int j = k; j < n; j++) {          // A = (I - 2vv^T/v^Tv) A
                double s = 0.0;
                for (int i = k; i < m; i++) s += v[i] * A[i][j];
                s = 2.0 * s / vv;
                for (int i = k; i < m; i++) A[i][j] -= s * v[i];
            }
            double s = 0.0;
            for (int i = k; i < m; i++) s += v[i] * c[i];
            s = 2.0 * s / vv;
            for (int i = k; i < m; i++) c[i] -= s * v[i];
        }
        double[] w = new double[n];
        for (int i = n - 1; i >= 0; i--) {                  // R w = (Q^T y)[0:n]
            double t = 0.0;
            for (int j = i + 1; j < n; j++) t += A[i][j] * w[j];
            w[i] = (c[i] - t) / A[i][i];
        }
        return w;
    }

    static double[] meanStd(double[] v) {
        double s = 0.0;
        for (double x : v) s += x;
        double mu = s / v.length, q = 0.0;
        for (double x : v) q += (x - mu) * (x - mu);
        return new double[] {mu, Math.sqrt(q / v.length)};
    }

    static double dot(double[] a, double[] b) {
        double s = 0.0;
        for (int i = 0; i < a.length; i++) s += a[i] * b[i];
        return s;
    }
```

Listing 14.2 (Continued) *MlCore.java: the same building blocks in Java.*

```
    static double rmse(double[][] X, double[] y, double[] w) {
        double s = 0.0;
        for (int i = 0; i < X.length; i++) {
            double e = dot(X[i], w) - y[i];
            s += e * e;
        }
        return Math.sqrt(s / X.length);
    }

    /** SplitMix64: the same stream of numbers in Python and in Java. */
    static class Rng {
        long s;

        Rng(long seed) { s = seed; }

        long next64() {
            s += 0x9E3779B97F4A7C15L;
            long z = s;
            z = (z ^ (z >>> 30)) * 0xBF58476D1CE4E5B9L;
            z = (z ^ (z >>> 27)) * 0x94D049BB133111EBL;
            return z ^ (z >>> 31);
        }

        int below(int n) { return (int) Long.remainderUnsigned(next64(), n); }

        void shuffle(int[] items) {
            for (int i = items.length - 1; i > 0; i--) {     // Fisher-Yates
                int j = below(i + 1);
                int t = items[i]; items[i] = items[j]; items[j] = t;
            }
        }
    }

    /** 1 / (1 + e^-z) without overflow for large |z|. */
    static double sigmoid(double z) {
        if (z >= 0) return 1.0 / (1.0 + Math.exp(-z));
        double e = Math.exp(z);
        return e / (1.0 + e);
    }

    /** log(1 + e^z) without overflow: max(z, 0) + log1p(e^-|z|). */
    static double softplus(double z) {
        return Math.max(z, 0.0) + Math.log1p(Math.exp(-Math.abs(z)));
    }
}
```

14.4.2 Fitting the trip time

Listing 14.3 fits Equation (14.1) by both methods, reports the RMSE and R^2 (the fraction of the variance of the times that the model explains), and then fits polynomials of degree 5, 9, 11 and 12 in the raw kilometers, and of degree 12 in the standardized distance. The powers are built by repeated multiplication, not with `**` or `Math.pow`, because the power functions of the two languages' math libraries are not guaranteed to round identically.

Listing 14.3 `linfit.py`: the trip-time model by the normal equations and by QR, and the polynomial experiment.

```
"""Trip time from distance, rush hour and rain: normal equations versus QR.
Usage: python linfit.py"""
from mlcore import load_trips, normal_equations, qr_solve, rmse, mean_std

rows = load_trips()
y = [r[4] for r in rows]
X = [[1.0, r[0], r[1], r[2]] for r in rows]
w_ne, w_qr = normal_equations(X, y), qr_solve(X, y)
print(f"{len(rows)} trips; model minutes = w0 + w1 km + w2 rush + w3 rain")
print("normal equations:", " ".join(f"{v:9.5f}" for v in w_ne))
print("Householder QR  :", " ".join(f"{v:9.5f}" for v in w_qr))
mu, sd = mean_std(y)
e = rmse(X, y, w_qr)
print(f"RMSE {e:.4f} minutes, R^2 {1 - e * e / (sd * sd):.4f}")

print("\npolynomials in distance: RMSE (minutes) by method")
print("degree  normal equations   Householder QR")
d = [r[0] for r in rows]
m, s = mean_std(d)
for deg, scaled in ((5, False), (9, False), (11, False), (12, False), (12, True)):
    t = [(x - m) / s if scaled else x for x in d]
    P = []
    for v in t:                                  # 1, v, v^2, ... by products
        row = [1.0]
        for k in range(deg):
            row.append(row[-1] * v)
        P.append(row)
    try:
        ne = f"{rmse(P, y, normal_equations(P, y)):.6f}"
    except ArithmeticError:
        ne = "Cholesky fails"
    label = f"{deg} std" if scaled else f"{deg} km"
    print(f"{label:6s}  {ne:>16s}   {rmse(P, y, qr_solve(P, y)):14.6f}")
```

Listing 14.4 is the Java twin; it builds the polynomial columns in a plain array and catches Java's `ArithmeticException` where Python catches `ArithmeticError`.

Listing 14.4 `LinFit.java`: the same fits in Java.

```
// Trip time from distance, rush hour and rain: normal equations versus QR.
// Usage: java LinFit
public class LinFit {
    static String row(double[] w) {
        StringBuilder sb = new StringBuilder();
        for (double v : w) sb.append(MlCore.f(" %9.5f", v));
        return sb.toString();
    }

    public static void main(String[] args) throws Exception {
        double[][] rows = MlCore.loadTrips();
        int n = rows.length;
        double[] y = new double[n], d = new double[n];
        double[][] X = new double[n][];
        for (int i = 0; i < n; i++) {
            y[i] = rows[i][4];
            d[i] = rows[i][0];
            X[i] = new double[] {1.0, rows[i][0], rows[i][1], rows[i][2]};
        }
        double[] wNe = MlCore.normalEquations(X, y), wQr = MlCore.qrSolve(X, y);
        System.out.println(n + " trips; model minutes"
                + " = w0 + w1 km + w2 rush + w3 rain");
        System.out.println("normal equations:" + row(wNe));
        System.out.println("Householder QR  :" + row(wQr));
        double[] ms = MlCore.meanStd(y);
        double e = MlCore.rmse(X, y, wQr);
        System.out.println(MlCore.f("RMSE %.4f minutes, R^2 %.4f",
                e, 1 - e * e / (ms[1] * ms[1])));

        System.out.println("\npolynomials in distance: RMSE (minutes) by method");
        System.out.println("degree  normal equations   Householder QR");
        double[] md = MlCore.meanStd(d);
        int[] degs = {5, 9, 11, 12, 12};
        boolean[] scaled = {false, false, false, false, true};
        for (int c = 0; c < degs.length; c++) {
            int deg = degs[c];
            double[][] P = new double[n][deg + 1];
            for (int i = 0; i < n; i++) {        // 1, v, v^2, ... by products
                double v = scaled[c] ? (d[i] - md[0]) / md[1] : d[i];
                P[i][0] = 1.0;
                for (int k = 1; k <= deg; k++) P[i][k] = P[i][k - 1] * v;
            }
            String ne;
            try {
                double[] w = MlCore.normalEquations(P, y);
                ne = MlCore.f("%.6f", MlCore.rmse(P, y, w));
```

Listing 14.4 (Continued) *LinFit.java: the same fits in Java.*

```
            } catch (ArithmeticException ex) {
                ne = "Cholesky fails";
            }
            String label = deg + (scaled[c] ? " std" : " km");
            System.out.println(MlCore.f("%-6s  %16s   %14.6f", label, ne,
                    MlCore.rmse(P, y, MlCore.qrSolve(P, y))));
        }
    }
}
```

14.4.3 Training by gradient methods

Listing 14.5 trains the same model five ways for 200 epochs each: full-batch gradient descent on raw and on standardized inputs, and mini-batch SGD (Equation (14.8)), SGD with momentum, and Adam, all with batches of 20 on standardized inputs. It first prints $\lambda_{\max}$ for both inputs, found by

the power method. For each run it reports the *excess loss*, how far the loss at the end lies above the exact minimum found by QR, and the first epoch at which the excess fell below 10^{-4}. Adam's bias corrections need β_1^t and β_2^t; the program keeps them as running products instead of calling a power function, for the reason just given.

Listing 14.5 `train_gd.py`: full-batch, mini-batch, momentum and Adam training of the trip-time model.

```
"""Training the trip-time model by gradient methods: why scaling matters,
and what mini-batches, momentum and Adam do.   Usage: python train_gd.py"""
import math
from mlcore import load_trips, qr_solve, mean_std, Rng, add

rows = load_trips()
y = [r[4] for r in rows]
RAW = [[1.0, r[0], r[1], r[2]] for r in rows]
stats = [mean_std([r[j] for r in rows]) for j in range(3)]
STD = [[1.0] + [(r[j] - stats[j][0]) / stats[j][1] for j in range(3)] for r in rows]
N = len(rows)


def loss(X, w, idx=None):
    idx = range(N) if idx is None else idx
    total = 0.0
    for i in idx:
        e = add(a * b for a, b in zip(X[i], w)) - y[i]
        total += e * e
    return total / (2 * len(idx))


def grad(X, w, idx):
    g = [0.0] * len(w)
    for i in idx:
        e = add(a * b for a, b in zip(X[i], w)) - y[i]
        for j in range(len(w)):
            g[j] += e * X[i][j]
    return [v / len(idx) for v in g]


def lambda_max(X, iters=100):
    """Largest eigenvalue of X^T X / N by the power method (Chapter 12)."""
    n = len(X[0])
    A = [[add(r[i] * r[j] for r in X) / N for j in range(n)] for i in range(n)]
    v, lam = [1.0] * n, 0.0
    for _ in range(iters):
        u = [add(A[i][j] * v[j] for j in range(n)) for i in range(n)]
        lam = math.sqrt(add(t * t for t in u))
        v = [t / lam for t in u]
    return lam
```

Listing 14.5 (Continued) *train_gd.py: full-batch, mini-batch, momentum and Adam training of the trip-time model.*

```
def run(X, method, lr, epochs, batch=N, seed=7):
    best = loss(X, qr_solve(X, y))
    w, m, v = [0.0] * 4, [0.0] * 4, [0.0] * 4
    order, rng, reached = list(range(N)), Rng(seed), None
    b1t = b2t = 1.0                                  # 0.9^t and 0.999^t
    for ep in range(1, epochs + 1):
        if batch < N:
            rng.shuffle(order)
        for s in range(0, N, batch):
            b1t, b2t = b1t * 0.9, b2t * 0.999
            g = grad(X, w, order[s:s + batch])
            for j in range(4):
                if method == "momentum":             # heavy ball, beta = 0.9
                    m[j] = 0.9 * m[j] + g[j]
                    w[j] -= lr * m[j]
                elif method == "adam":               # beta1 0.9, beta2 0.999
                    m[j] = 0.9 * m[j] + 0.1 * g[j]
                    v[j] = 0.999 * v[j] + 0.001 * g[j] * g[j]
                    mh, vh = m[j] / (1 - b1t), v[j] / (1 - b2t)
                    w[j] -= lr * mh / (math.sqrt(vh) + 1e-8)
                else:
                    w[j] -= lr * g[j]
        if reached is None and loss(X, w) - best < 1e-4:
            reached = ep
    return w, max(loss(X, w) - best, 0.0), reached


def unscale(w):
    """Weights for standardized inputs -> weights for km, rush, rain."""
    out = [w[0]] + [w[j + 1] / stats[j][1] for j in range(3)]
    out[0] -= add(w[j + 1] * stats[j][0] / stats[j][1] for j in range(3))
    return out


def main():
    print(f"lambda_max of X^T X / N: raw {lambda_max(RAW):.3f},"
          f" standardized {lambda_max(STD):.3f}")
    print("method              inputs  lr      epochs  excess loss  reached")
    cases = [("full-batch GD", RAW, "gd", 0.012, N),
             ("full-batch GD", STD, "gd", 0.5, N),
             ("mini-batch SGD", STD, "gd", 0.05, 20),
             ("SGD + momentum", STD, "momentum", 0.005, 20),
             ("Adam", STD, "adam", 0.05, 20)]
    for name, X, method, lr, batch in cases:
        w, excess, reached = run(X, method, lr, 200, batch)
        tag = "raw" if X is RAW else "std"
```

Listing 14.5 (Continued) *train_gd.py: full-batch, mini-batch, momentum and Adam training of the trip-time model.*

```
        print(f"{name:18s}  {tag:6s}  {str(lr):6s}  {200:6d}  {excess:11.2e}"
              f"  {reached if reached else '-':>7}")
        if X is STD and method == "gd" and batch == N:
            w_std = w
    print("GD on standardized inputs, back in km units:",
          " ".join(f"{v:.5f}" for v in unscale(w_std)))


if __name__ == "__main__":
    main()
```

Listing 14.6 spells out the five cases as parallel arrays, because Java has no list of tuples, and keeps the same order of operations everywhere.

Listing 14.6 `TrainGd.java`: the same experiment in Java.

```java
// Training the trip-time model by gradient methods: why scaling matters,
// and what mini-batches, momentum and Adam do.   Usage: java TrainGd
public class TrainGd {
    static double[] y;
    static double[][] RAW, STD, stats = new double[3][];
    static int N;

    static double loss(double[][] X, double[] w) {
        double total = 0.0;
        for (int i = 0; i < N; i++) {
            double e = MlCore.dot(X[i], w) - y[i];
            total += e * e;
        }
        return total / (2 * N);
    }

    static double[] grad(double[][] X, double[] w, int[] idx, int from, int to) {
        double[] g = new double[w.length];
        for (int k = from; k < to; k++) {
            int i = idx[k];
            double e = MlCore.dot(X[i], w) - y[i];
            for (int j = 0; j < w.length; j++) g[j] += e * X[i][j];
        }
        for (int j = 0; j < w.length; j++) g[j] /= (to - from);
        return g;
    }

    /** Largest eigenvalue of X^T X / N by the power method (Chapter 12). */
    static double lambdaMax(double[][] X) {
        int n = X[0].length;
        double[][] A = new double[n][n];
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++) {
                double s = 0.0;
                for (double[] r : X) s += r[i] * r[j];
                A[i][j] = s / N;
            }
        double[] v = {1.0, 1.0, 1.0, 1.0};
        double lam = 0.0;
        for (int it = 0; it < 100; it++) {
            double[] u = new double[n];
            for (int i = 0; i < n; i++) u[i] = MlCore.dot(A[i], v);
            lam = Math.sqrt(MlCore.dot(u, u));
            for (int i = 0; i < n; i++) v[i] = u[i] / lam;
        }
        return lam;
```

Listing 14.6 (Continued) *TrainGd.java: the same experiment in Java.*

```
    }

    /** Returns {w0..w3, excess loss, epoch reached (0 = never)}. */
    static double[] run(double[][] X, String method, double lr, int epochs,
                        int batch) {
        double best = loss(X, MlCore.qrSolve(X, y));
        double[] w = new double[4], m = new double[4], v = new double[4];
        int[] order = new int[N];
        for (int i = 0; i < N; i++) order[i] = i;
        MlCore.Rng rng = new MlCore.Rng(7);
        int reached = 0;
        double b1t = 1.0, b2t = 1.0;                  // 0.9^t and 0.999^t
        for (int ep = 1; ep <= epochs; ep++) {
            if (batch < N) rng.shuffle(order);
            for (int s = 0; s < N; s += batch) {
                b1t *= 0.9;
                b2t *= 0.999;
                double[] g = grad(X, w, order, s, Math.min(s + batch, N));
                for (int j = 0; j < 4; j++) {
                    if (method.equals("momentum")) {      // heavy ball, beta = 0.9
                        m[j] = 0.9 * m[j] + g[j];
                        w[j] -= lr * m[j];
                    } else if (method.equals("adam")) {   // beta1 0.9, beta2 0.999
                        m[j] = 0.9 * m[j] + 0.1 * g[j];
                        v[j] = 0.999 * v[j] + 0.001 * g[j] * g[j];
                        double mh = m[j] / (1 - b1t), vh = v[j] / (1 - b2t);
                        w[j] -= lr * mh / (Math.sqrt(vh) + 1e-8);
                    } else {
                        w[j] -= lr * g[j];
                    }
                }
            }
            if (reached == 0 && loss(X, w) - best < 1e-4) reached = ep;
        }
        return new double[] {w[0], w[1], w[2], w[3],
                Math.max(loss(X, w) - best, 0.0), reached};
    }
```

Listing 14.6 (Continued) *TrainGd.java: the same experiment in Java.*

```
    public static void main(String[] args) throws Exception {
        double[][] rows = MlCore.loadTrips();
        N = rows.length;
        y = new double[N];
        RAW = new double[N][];
        STD = new double[N][];
        for (int j = 0; j < 3; j++) {
            double[] col = new double[N];
            for (int i = 0; i < N; i++) col[i] = rows[i][j];
            stats[j] = MlCore.meanStd(col);
        }
        for (int i = 0; i < N; i++) {
            double[] r = rows[i];
            y[i] = r[4];
            RAW[i] = new double[] {1.0, r[0], r[1], r[2]};
            STD[i] = new double[4];
            STD[i][0] = 1.0;
            for (int j = 0; j < 3; j++)
                STD[i][j + 1] = (r[j] - stats[j][0]) / stats[j][1];
        }
        System.out.println(MlCore.f(
                "lambda_max of X^T X / N: raw %.3f, standardized %.3f",
                lambdaMax(RAW), lambdaMax(STD)));
        System.out.println(
                "method              inputs  lr      epochs  excess loss  reached");
        String[] names = {"full-batch GD", "full-batch GD", "mini-batch SGD",
                          "SGD + momentum", "Adam"};
        double[][][] xs = {RAW, STD, STD, STD, STD};
        String[] methods = {"gd", "gd", "gd", "momentum", "adam"};
        double[] lrs = {0.012, 0.5, 0.05, 0.005, 0.05};
        int[] batches = {N, N, 20, 20, 20};
        double[] wStd = null;
        for (int c = 0; c < names.length; c++) {
            double[] r = run(xs[c], methods[c], lrs[c], 200, batches[c]);
            String tag = xs[c] == RAW ? "raw" : "std";
            String reached = r[5] > 0 ? String.valueOf((int) r[5]) : "-";
            System.out.println(MlCore.f("%-18s  %-6s  %-6s  %6d  %11.2e  %7s",
                    names[c], tag, Double.toString(lrs[c]), 200, r[4], reached));
            if (c == 1) wStd = r;
        }
        double[] out = new double[4];              // back to km, rush, rain units
        double shift = 0.0;
        for (int j = 0; j < 3; j++) {
            out[j + 1] = wStd[j + 1] / stats[j][1];
            shift += wStd[j + 1] * stats[j][0] / stats[j][1];
        }
```

Listing 14.6 (Continued) *TrainGd.java: the same experiment in Java.*

```
        out[0] = wStd[0] - shift;
        StringBuilder sb =
                new StringBuilder("GD on standardized inputs, back in km units:");
        for (double v : out) sb.append(MlCore.f(" %.5f", v));
        System.out.println(sb);
    }
}
```

14.4.4 Predicting cancellations

Listing 14.7 first prints the naive and the stable log-loss for four extreme cases, then checks the gradient Equation (14.14) against central differences, runs Newton's method Equation (14.15) from $\mathbf{w} = \mathbf{0}$, translates the weights into odds factors, trains the same model by gradient descent with the pickup distance standardized, and finally counts how many cancellations a threshold of

0.5 would catch. The naive loss is computed inside `try`, because Python raises an exception on overflow where Java quietly returns infinity; both are turned into the text `inf`.

Listing 14.7 `logistic.py`: logistic regression by Newton's method (IRLS) and by gradient descent, with a gradient check and stable formulas.

```
"""Will the customer cancel? Logistic regression by Newton's method (IRLS) and by
gradient descent, with stable formulas and a gradient check.
Usage: python logistic.py"""
import math
from mlcore import (load_trips, cholesky_solve, mean_std, sigmoid, softplus,
                    add)

rows = load_trips()
y = [r[5] for r in rows]
X = [[1.0, r[3], r[2], r[1]] for r in rows]      # 1, pickup km, rain, rush
N, P = len(rows), 4


def z_of(w, x):
    return add(a * b for a, b in zip(w, x))


def loss(w):
    """Mean log-loss, written with softplus so that it never overflows."""
    return add(softplus(z_of(w, x)) - t * z_of(w, x) for x, t in zip(X, y)) / N


def grad(w):
    g = [0.0] * P
    for x, t in zip(X, y):
        e = sigmoid(z_of(w, x)) - t
        for j in range(P):
            g[j] += e * x[j] / N
    return g


def hessian(w):
    H = [[0.0] * P for _ in range(P)]
    for x in X:
        p = sigmoid(z_of(w, x))
        for i in range(P):
            for j in range(P):
                H[i][j] += p * (1 - p) * x[i] * x[j] / N
    return H
```

Listing 14.7 (Continued) *logistic.py: logistic regression by Newton's method (IRLS) and by gradient descent, with a gradient check and stable formulas.*

```
def naive_loss(z, t):
    """The textbook formula -[t log p + (1-t) log(1-p)], p = 1/(1+e^-z)."""
    try:
        p = 1.0 / (1.0 + math.exp(-z))
        return -math.log(p) if t == 1 else -math.log(1.0 - p)
    except (OverflowError, ValueError):
        return math.inf


def show(v):
    return "inf" if math.isinf(v) else f"{v:.6f}"


print("one trip, stable versus naive log-loss")
for z, t in ((-5.0, 1), (-40.0, 1), (-800.0, 1), (40.0, 0)):
    print(f"  z = {z:7.1f}, cancelled = {t}: naive {show(naive_loss(z, t)):>10s},"
          f" stable {softplus(z) - t * z:.6f}")

w0, h = [-1.0, 0.5, 0.5, 0.5], 1e-6
g = grad(w0)
num = []
for j in range(P):
    wp, wm = w0[:], w0[:]
    wp[j] += h
    wm[j] -= h
    num.append((loss(wp) - loss(wm)) / (2 * h))
print("gradient check, max difference:",
      f"{max(abs(a - b) for a, b in zip(g, num)):.2e}")

print("Newton / IRLS from w = 0")
w = [0.0] * P
for k in range(1, 20):
    step = cholesky_solve(hessian(w), grad(w))        # H s = g
    w = [a - b for a, b in zip(w, step)]
    gn = math.sqrt(add(v * v for v in grad(w)))
    print(f"  {k}  loss {loss(w):.10f}  |grad| {gn:.2e}")
    if gn < 1e-10:
        break
print("weights:", " ".join(f"{v:.5f}" for v in w))
print(f"odds of cancelling x{math.exp(w[1]):.3f} per km of pickup,"
      f" x{math.exp(w[2]):.3f} in rain, x{math.exp(w[3]):.3f} in rush hour")
best = loss(w)
```

Listing 14.7 (Continued) *logistic.py: logistic regression by Newton's method (IRLS) and by gradient descent, with a gradient check and stable formulas.*

```
pk = mean_std([x[1] for x in X])
Xs = [[1.0, (x[1] - pk[0]) / pk[1], x[2], x[3]] for x in X]
X, v, ep = Xs, [0.0] * P, 0
while loss(v) - best > 1e-6 and ep < 100000:
    ep += 1
    v = [a - 2.0 * b for a, b in zip(v, grad(v))]      # fixed step 2.0
print(f"gradient descent, standardized pickup, step 2.0: {ep} epochs")
print("  to come within 1e-6 of Newton's loss")

tp = fp = fn = tn = 0
for x, t in zip(X, y):
    pred = 1 if sigmoid(z_of(v, x)) >= 0.5 else 0
    tp += pred == 1 and t == 1
    fp += pred == 1 and t == 0
    fn += pred == 0 and t == 1
    tn += pred == 0 and t == 0
print(f"threshold 0.5: accuracy {(tp + tn) / N:.3f};"
      f" caught {tp} of {tp + fn} cancellations, {fp} false alarms")
```

Listing 14.8 needs no `try`: in Java `Math.exp(800.0)` is infinity and `Math.log(0.0)` is minus infinity, so the naive loss simply becomes infinity, which `show` prints as `inf`.

Listing 14.8 `Logistic.java`: the same in Java.

```java
// Will the customer cancel? Logistic regression by Newton's method (IRLS) and by
// gradient descent, with stable formulas and a gradient check.
// Usage: java Logistic
public class Logistic {
    static double[][] X;
    static double[] y;
    static int N;
    static final int P = 4;

    /** Mean log-loss, written with softplus so that it never overflows. */
    static double loss(double[] w) {
        double s = 0.0;
        for (int i = 0; i < N; i++) {
            double z = MlCore.dot(w, X[i]);
            s += MlCore.softplus(z) - y[i] * z;
        }
        return s / N;
    }

    static double[] grad(double[] w) {
        double[] g = new double[P];
        for (int i = 0; i < N; i++) {
            double e = MlCore.sigmoid(MlCore.dot(w, X[i])) - y[i];
            for (int j = 0; j < P; j++) g[j] += e * X[i][j] / N;
        }
        return g;
    }

    static double[][] hessian(double[] w) {
        double[][] H = new double[P][P];
        for (double[] x : X) {
            double p = MlCore.sigmoid(MlCore.dot(w, x));
            for (int i = 0; i < P; i++)
                for (int j = 0; j < P; j++)
                    H[i][j] += p * (1 - p) * x[i] * x[j] / N;
        }
        return H;
    }

    /** The textbook formula -[t log p + (1-t) log(1-p)], p = 1/(1+e^-z). */
    static double naiveLoss(double z, int t) {
        double p = 1.0 / (1.0 + Math.exp(-z));
        return t == 1 ? -Math.log(p) : -Math.log(1.0 - p);
    }
```

Listing 14.8 (Continued) *Logistic.java: the same in Java.*

```
    static String show(double v) {
        return Double.isInfinite(v) ? "inf" : MlCore.f("%.6f", v);
    }

    static double norm(double[] v) { return Math.sqrt(MlCore.dot(v, v)); }

    public static void main(String[] args) throws Exception {
        double[][] rows = MlCore.loadTrips();
        N = rows.length;
        X = new double[N][];
        y = new double[N];
        for (int i = 0; i < N; i++) {                  // 1, pickup km, rain, rush
            X[i] = new double[] {1.0, rows[i][3], rows[i][2], rows[i][1]};
            y[i] = rows[i][5];
        }
        System.out.println("one trip, stable versus naive log-loss");
        double[] zs = {-5.0, -40.0, -800.0, 40.0};
        int[] ts = {1, 1, 1, 0};
        for (int c = 0; c < 4; c++)
            System.out.println(MlCore.f(
                    "  z = %7.1f, cancelled = %d: naive %10s, stable %.6f",
                    zs[c], ts[c], show(naiveLoss(zs[c], ts[c])),
                    MlCore.softplus(zs[c]) - ts[c] * zs[c]));

        double[] w0 = {-1.0, 0.5, 0.5, 0.5};
        double h = 1e-6, worst = 0.0;
        double[] g = grad(w0);
        for (int j = 0; j < P; j++) {
            double[] wp = w0.clone(), wm = w0.clone();
            wp[j] += h;
            wm[j] -= h;
            double num = (loss(wp) - loss(wm)) / (2 * h);
            worst = Math.max(worst, Math.abs(g[j] - num));
        }
        System.out.println(MlCore.f("gradient check, max difference: %.2e", worst));
```

Listing 14.8 (Continued) *Logistic.java: the same in Java.*

```
        System.out.println("Newton / IRLS from w = 0");
        double[] w = new double[P];
        for (int k = 1; k < 20; k++) {
            double[] step = MlCore.choleskySolve(hessian(w), grad(w));   // H s = g
            for (int j = 0; j < P; j++) w[j] -= step[j];
            double gn = norm(grad(w));
            System.out.println(MlCore.f("  %d  loss %.10f  |grad| %.2e",
                    k, loss(w), gn));
            if (gn < 1e-10) break;
        }
        System.out.println(MlCore.f("weights: %.5f %.5f %.5f %.5f",
                w[0], w[1], w[2], w[3]));
        System.out.println(MlCore.f(
                "odds of cancelling x%.3f per km of pickup, x%.3f in rain,"
                + " x%.3f in rush hour",
                Math.exp(w[1]), Math.exp(w[2]), Math.exp(w[3])));
        double best = loss(w);

        double[] pick = new double[N];
        for (int i = 0; i < N; i++) pick[i] = X[i][1];
        double[] pk = MlCore.meanStd(pick);
        for (int i = 0; i < N; i++)
            X[i] = new double[] {1.0, (X[i][1] - pk[0]) / pk[1], X[i][2], X[i][3]};
        double[] v = new double[P];
        int ep = 0;
        while (loss(v) - best > 1e-6 && ep < 100000) {
            ep++;
            double[] gv = grad(v);
            for (int j = 0; j < P; j++) v[j] -= 2.0 * gv[j];      // fixed step 2.0
        }
        System.out.println(MlCore.f(
                "gradient descent, standardized pickup, step 2.0: %d epochs", ep));
        System.out.println("  to come within 1e-6 of Newton's loss");

        int tp = 0, fp = 0, fn = 0, tn = 0;
        for (int i = 0; i < N; i++) {
            int pred = MlCore.sigmoid(MlCore.dot(v, X[i])) >= 0.5 ? 1 : 0;
            if (pred == 1 && y[i] == 1) tp++;
            if (pred == 1 && y[i] == 0) fp++;
            if (pred == 0 && y[i] == 1) fn++;
            if (pred == 0 && y[i] == 0) tn++;
        }
        System.out.println(MlCore.f(
                "threshold 0.5: accuracy %.3f; caught %d of %d cancellations,"
                + " %d false alarms", (double) (tp + tn) / N, tp, tp + fn, fp));
    }
```

Listing 14.8 (Continued) *Logistic.java: the same in Java.*

```
}
```

14.5 Running the Programs

```
cd Chapter14/python
python3 linfit.py
python3 train_gd.py
python3 logistic.py
cd ../java
javac -d out *.java
java -cp out LinFit
java -cp out TrainGd
java -cp out Logistic
```

The least-squares program prints:

```
400 trips; model minutes = w0 + w1 km + w2 rush + w3 rain
normal equations:   4.00351   2.59547   5.68935   5.52595
Householder QR  :   4.00351   2.59547   5.68935   5.52595
RMSE 2.5693 minutes, R^2 0.9486

polynomials in distance: RMSE (minutes) by method
degree  normal equations   Householder QR
5 km            4.524659         4.524659
9 km            4.503983         4.503983
11 km           4.502751         4.502704
12 km     Cholesky fails         4.494450
12 std          4.494450         4.494450
```

For the real model both methods give the same weights: 4.0 minutes of fixed time, 2.6 minutes per kilometer, 5.7 extra minutes at rush hour and 5.5 in the rain. The RMSE is 2.57 minutes and $R^2 = 0.949$: the four numbers explain 95% of the variation in Mas Dimas's trip times, and a typical prediction is off by two and a half minutes, which is about the scatter one expects from traffic lights.

The polynomial table is Equation (14.6) made visible. At degree 5 ($\kappa(X) \approx 5.6 \times 10^6$, so $\kappa(X^\mathsf{T}X) \approx 3 \times 10^{13}$) the normal equations still agree with QR to six decimals. At degree 11 ($\kappa(X) \approx 2.5 \times 10^{15}$) they return weights whose RMSE, 4.502751, is *larger* than the true minimum 4.502704 found by QR: the normal equations have not found the least-squares solution at all, only something near it. At degree 12 Cholesky fails outright. With the distance standardized, the same degree-12 polynomial is solved by both methods without trouble, because the powers of numbers between about -1.6 and 2.1 are far less alike than the powers of numbers between 0.9 and 15. Building the powers with `**` instead of by repeated multiplication, which changes only the last bit of some entries, moves the degree-11 RMSE of the normal equations to 4.502767: in an ill-conditioned problem, rounding differences in the last bit of the input change the fifth digit of the result.

The training program prints:

```
lambda_max of X^T X / N: raw 77.409, standardized 1.092
method              inputs  lr      epochs  excess loss  reached
full-batch GD       raw     0.012      200     2.51e+00        -
full-batch GD       std     0.5        200     0.00e+00       12
mini-batch SGD      std     0.05       200     1.88e-03       92
SGD + momentum      std     0.005      200     2.51e-03       24
Adam                std     0.05       200     4.35e-03       88
GD on standardized inputs, back in km units: 4.00351 2.59547 5.68935 5.52595
```

Figure 14.1 plots the excess loss against the epoch for all five runs. Full-batch descent on the raw inputs, limited by $\alpha < 2/77.4$ and slowed by $\kappa = 546$, is still 2.5 above the minimum after 200 epochs. On standardized inputs the same algorithm reaches 10^{-4} after 12 epochs and machine precision soon after, exactly the geometric decrease predicted in Section 13.3.4. The three mini-batch methods drop quickly at first, each epoch being 20 steps instead of one, and then stall in a noise floor between 10^{-3} and 10^{-2}; the column "reached" shows that each of them touched 10^{-4} at some epoch and then drifted away again. That is the behavior Section 14.3.3 predicted for a fixed learning rate. For 400 rows full-batch descent on standardized inputs is clearly the best choice; for 400 million rows, where one full gradient costs as much as a million mini-batch steps, the mini-batch methods win, and their noise floor is then lowered by a decreasing learning rate. The last line confirms that the standardized weights, converted back, are exactly the QR weights.

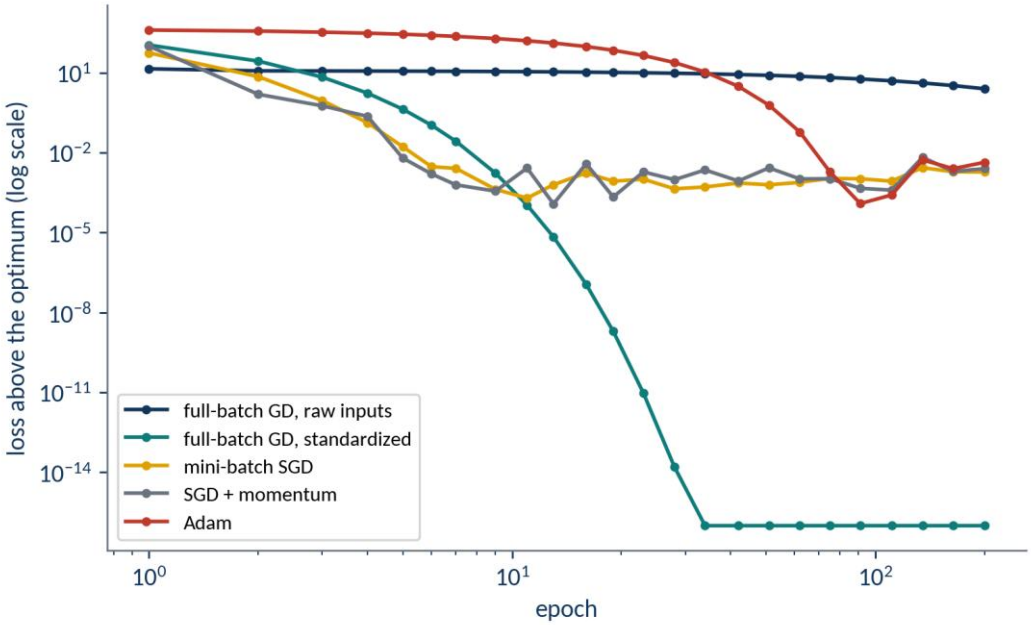


Figure 14.1 Loss above the least-squares optimum during training: full-batch gradient descent on raw and on standardized inputs, and mini-batch SGD, momentum and Adam on standardized inputs (both axes logarithmic).

The logistic program prints:

```
one trip, stable versus naive log-loss
  z =    -5.0, cancelled = 1: naive   5.006715, stable 5.006715
  z =   -40.0, cancelled = 1: naive  40.000000, stable 40.000000
  z =  -800.0, cancelled = 1: naive         inf, stable 800.000000
  z =    40.0, cancelled = 0: naive         inf, stable 40.000000
gradient check, max difference: 2.98e-10
Newton / IRLS from w = 0
  1  loss 0.4596129941  |grad| 5.70e-02
  2  loss 0.4341017682  |grad| 1.40e-02
  3  loss 0.4317674110  |grad| 1.64e-03
  4  loss 0.4317353430  |grad| 2.64e-05
  5  loss 0.4317353354  |grad| 6.52e-09
  6  loss 0.4317353354  |grad| 4.68e-16
weights: -4.80740 1.29936 1.00720 1.11312
odds of cancelling x3.667 per km of pickup, x2.738 in rain, x3.044 in rush hour
gradient descent, standardized pickup, step 2.0: 128 epochs
  to come within 1e-6 of Newton's loss
threshold 0.5: accuracy 0.790; caught 57 of 110 cancellations, 31 false alarms
```

The first block is Equation (14.16) at work: for moderate z the naive and stable formulas agree, and in the two extreme cases the naive formula returns infinity where the true loss is 800 or 40. The gradient check agrees to 3×10^{-10}. Newton's method needs six steps, and the gradient lengths 5.7×10^{-2}, 1.4×10^{-2}, 1.6×10^{-3}, 2.6×10^{-5}, 6.5×10^{-9}, 4.7×10^{-16} show the characteristic doubling of correct digits. Gradient descent needs 128 epochs to come within 10^{-6} of the same loss.

The weights translate into advice Mas Dimas can use: each kilometer of pickup multiplies the odds of a cancellation by 3.7, rain by 2.7, rush hour by 3.0. **Figure 14.2** draws the predicted probability against the pickup distance for the four combinations of rain and rush hour, together with the observed share of cancellations in half-kilometer bins. On a dry afternoon, an order 1 km away is cancelled with probability about 0.03; in the rain at rush hour the same order has 0.20, and one 3 km away has 0.77. The threshold 0.5 gives an accuracy of 0.79, against 0.725 for the

lazy rule “nobody cancels”, and catches 57 of the 110 cancellations at the cost of 31 false alarms. Where he sets his threshold is his choice: a lower one catches more cancellations and turns down more good orders. The model advises; the driver decides.

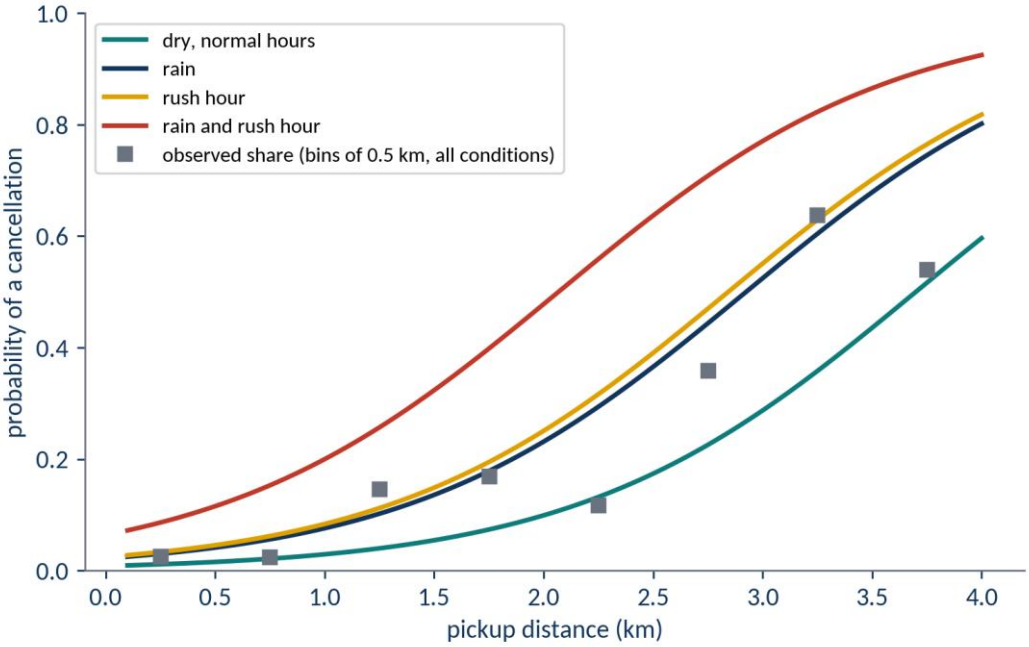


Figure 14.2 Probability that an order is cancelled, as learned by logistic regression, against the pickup distance for four combinations of rain and rush hour; squares show the observed share in half-kilometer bins over all conditions.

14.6 Comparing with NumPy and SciPy

Listing 14.9 checks the answers with NumPy’s `lstsq`, which solves least squares by the singular value decomposition [12], computes the condition numbers quoted in this chapter, fits the logistic model with SciPy’s BFGS using the library’s stable `log_expit` [13], and repeats the variance experiment in float64 and float32. It was run with NumPy 2.5.3 and SciPy 1.18.1 [14], [15].

Listing 14.9 `check_numpy.py`: the chapter’s results checked with NumPy and SciPy, the condition numbers, and a float32 experiment.

```
"""Chapter 14 answers checked with NumPy and SciPy, plus condition numbers and a
float32 experiment.   Usage: python check_numpy.py   (needs numpy and scipy)"""
import numpy as np
from scipy.optimize import minimize
from scipy.special import expit, log_expit
from mlcore import DATA

d = np.loadtxt(f"{DATA}/trips.csv", delimiter=",", skiprows=1)
km, rush, rain, pick, mins, canc = d.T
X = np.column_stack([np.ones(len(d)), km, rush, rain])
w, *_ = np.linalg.lstsq(X, mins, rcond=None)
print("lstsq weights:", " ".join(f"{v:.5f}" for v in w))
F = X[:, 1:]
Z = np.column_stack([np.ones(len(d)), (F - F.mean(0)) / F.std(0)])
print(f"cond(X^T X): raw {np.linalg.cond(X.T @ X):.1f},"
      f" standardized {np.linalg.cond(Z.T @ Z):.3f}")
for deg in (5, 9, 11, 12):
    V = np.vander(km, deg + 1, increasing=True)
    print(f"degree {deg:2d}: cond(X) {np.linalg.cond(V):.2e},"
          f" cond(X^T X) {np.linalg.cond(V.T @ V):.2e}")

A = np.column_stack([np.ones(len(d)), pick, rain, rush])


def f(w):
    z = A @ w
    return -np.mean(canc * log_expit(z) + (1 - canc) * log_expit(-z))


def g(w):
    return A.T @ (expit(A @ w) - canc) / len(d)


r = minimize(f, np.zeros(4), jac=g, method="BFGS", options={"gtol": 1e-10})
print("BFGS logistic weights:", " ".join(f"{v:.5f}" for v in r.x),
      f"loss {r.fun:.10f}")

t = mins + 43200.0          # a clock in minutes, 30 days into the month
for dt in (np.float64, np.float32):
    x = t.astype(dt)
    naive = np.mean(x * x) - np.mean(x) ** 2        # E[x^2] - E[x]^2
    two_pass = np.mean((x - np.mean(x)) ** 2)
    print(f"variance, {dt.__name__:7s}: naive {float(naive):12.4f},"
          f" two-pass {float(two_pass):10.4f}")
```

```
lstsq weights: 4.00351 2.59547 5.68935 5.52595
cond(X^T X): raw 546.1, standardized 1.209
degree  5: cond(X) 5.63e+06, cond(X^T X) 3.17e+13
degree  9: cond(X) 3.15e+12, cond(X^T X) 1.62e+23
degree 11: cond(X) 2.55e+15, cond(X^T X) 9.39e+26
degree 12: cond(X) 7.62e+16, cond(X^T X) 5.73e+28
BFGS logistic weights: -4.80740 1.29936 1.00720 1.11312 loss 0.4317353354
variance, float64: naive     128.4051, two-pass   128.4051
variance, float32: naive     128.0000, two-pass   128.4051
```

The weights of both models agree with ours to every printed digit. The condition numbers confirm Equation (14.5): $3.17 \times 10^{13} \approx (5.63 \times 10^{6})^2$. For degrees 9 and above the computed $\kappa(X^\top X)$ exceeds 10^{16}, and such numbers are themselves unreliable: they only say "numerically singular". Standardization brings $\kappa(X^\top X)$ of the four-feature model from 546 to 1.21.

The last two lines show the float32 trap of Section 14.3.6. The two-pass formula, which subtracts the mean first and then squares small numbers, gives the right variance 128.4051 in both precisions. The one-pass formula gives exactly 128.0000 in float32: the two means it subtracts are near 1.87×10^{9}, where consecutive float32 numbers are 128 apart, so their difference can

only be a multiple of 128. A variance of 128.0 or of 0.0 or of 256.0 are equally possible outcomes; that it came out close to the truth is luck.

14.7 Making It Better

- **Regularize.** Adding $\frac{\lambda}{2} \| \mathbf{w} \|^2$ to the loss (*ridge regression*, for $\lambda > 0$) changes the normal equations to $(X^\top X + N\lambda I)\mathbf{w} = X^\top \mathbf{y}$. Every eigenvalue grows by $N\lambda$, so the condition number drops, Cholesky can no longer fail, and the weights stop chasing noise. It is the numerical and the statistical cure for near-duplicate features at once [5]. (The constant weight w_0 is usually left out of the penalty.)
- **Hold data back.** A model is useful only if it predicts trips it has not seen. Fit on about 80% of the log and measure the RMSE or accuracy on the remaining 20%. A degree-12 polynomial in distance always fits the trips it was fitted to at least as well as a straight line in distance, and usually predicts new trips worse; only the held-back trips reveal it.
- **Schedule the learning rate.** For SGD and Adam, reduce α by a factor after a number of epochs, or in proportion to $1/\sqrt{t}$; the noise floor of **Figure 14.1** falls with it.
- **Keep a float64 accumulator.** When data or weights are float32, accumulate sums and losses in float64 (or use pairwise summation, Chapter 15). It costs almost nothing and removes the worst of the precision problems.

14.8 Using It for Real

Mas Dimas's two models can be built for any small business that keeps a log. The steps are the same whether the log records trips, laundry orders, patients' waiting times at a *puskesmas* queue, or the yield of each row of an orchard.

- **Keep the log in a spreadsheet and export it as CSV.** One row per event, one column per fact, numbers only (0 and 1 for yes and no). Record the facts *before* the outcome is known, because those are the only ones available when a prediction is needed: the pickup distance is known when the order arrives, the trip time only afterwards. A log that contains names or phone numbers is personal data under Indonesia's UU PDP 27/2022; the models need none of it, so leave such columns out from the start.
- **Standardize, fit by QR, and hold a fifth back.** Use Equation (14.7) with the mean and standard deviation of the rows used for fitting, fit on four fifths of the rows, and measure the error on the rest. If the error on the held-back rows is much larger than on the fitted ones, the model is too flexible for the data.
- **Choose the threshold from money, not from 0.5.** For a yes-or-no decision, compare what each mistake costs. For Mas Dimas, accepting an order that is then cancelled costs about Rp 6,000 of fuel and time; refusing an order that would have gone through loses its margin, about Rp 12,000 on a typical trip. Accepting an order with cancellation probability p is worthwhile when the expected gain $(1 - p) \times 12{,}000$ exceeds the expected loss $p \times 6{,}000$, that is when $p < 12{,}000/18{,}000 = 0.67$. The right threshold for him is 0.67, not 0.5; a driver with a more fuel-hungry motorbike would choose a lower one.
- **Refit every month.** Traffic, weather and customers change. A model fitted in the dry season will be too optimistic in the rainy season; refitting on the latest two months keeps it honest, and Newton's method makes refitting instant.

As a second example, take a small *puskesmas* that wants to tell patients how long they will wait. Each visit in its queue log gives the time of arrival, the number of people already waiting, whether

a doctor or a midwife is on duty, and the waiting time. A linear model like Equation (14.1) with those features, fitted by QR, predicts the wait within a few minutes, and the same numerical care applies: standardize the features, keep the log free of names, and check the error on held-back days. What the model must not do is decide who is seen first: the order of treatment is a medical decision, made by the staff on duty, and a patient who feels worse while waiting should tell the nurse at once rather than wait for any predicted time.

14.9 Numerical Methods in Practice: Profitina

Profitina's AI features run on local models only: Qwen2.5 through Ollama, with a 7B model for quality work and an optional very small model (qwen3:1.7b) as a fast lane for short classification tasks [16]. A small model that answers "yes or no" for a short text is doing, at a much larger scale, what Mas Dimas's logistic regression does: turning a weighted sum into a probability with a sigmoid or its many-class cousin, the softmax, whose stable evaluation is the log-sum-exp form of Section 14.3.5.

The platform's feedback pipeline makes the rest of the chapter concrete. Users submit feedback on AI responses; it is curated into a dataset; the model is trained on it and *measured* before a release [16]. Training is the minimization of a loss by a gradient method of Section 14.3.3, and measuring is the held-back evaluation of Section 14.7. The Link Safety feature shows why the threshold of a classifier is a decision and not a formula: its rule floor, which combines several signals, catches 27 of 28 realistic scam texts with no false positives on 30 normal texts in the internal benchmark, and an optional local-AI opinion sits on top [16]. Moving the threshold would trade missed scams against false alarms, exactly as Mas Dimas's threshold trades missed cancellations against refused orders. And the weekly score trends that the dashboard shows are sums over growing tables, where the accumulation advice of Section 14.7 applies.

14.10 Common Mistakes, and How to Avoid Them

- **Solving least squares by inverting $X^{\mathsf{T}}X$.** Forming $X^{\mathsf{T}}X$ squares the condition number (Equation (14.5)), and inverting it adds more error. Use QR, or a library least-squares routine; use the normal equations only when $\kappa(X)$ is known to be small.
- **Unscaled features.** A feature in rupiah next to a feature in kilometers makes κ enormous and gradient descent crawl. Standardize with Equation (14.7), and remember to apply the *training* mean and standard deviation to new data.
- **A learning rate copied from a tutorial.** The safe rate depends on $\lambda_{\max}$ of your data. Estimate it with a few power-method steps, or start small and increase.
- **Coding the log-loss as written in the textbook.** $\ln\sigma(z)$ and $\ln(1-\sigma(z))$ overflow or return $-\infty$ for confident predictions. Use softplus form Equation (14.13) with Equation (14.16), or the library's `log_expit`.
- **Reading SGD's noise floor as convergence.** A mini-batch loss that stops falling may be wandering in a cloud around the minimum. Compare with a full-batch evaluation, and lower the learning rate.
- **Trusting accuracy alone.** When only 27.5% of orders are cancelled, "nobody cancels" is 72.5% accurate. Report how many of the positives are caught and how many false alarms are raised.
- **Summing in float32.** One-pass formulas and long running sums lose everything in single precision; accumulate in float64.

- **Comparing two implementations bit for bit without fixing the order of operations.** Different summation orders, different power functions or different random generators give different last bits; ill-conditioning magnifies them. Fix the order, the functions and the generator, as the programs here do.

14.11 Chapter Summary

- Learning a model is fitting weights by minimizing a loss: least squares Equation (14.2) for numbers, log-loss Equation (14.12) for probabilities. The numerical methods are those of Chapters 6, 8, 12 and 13.
- The normal equations Equation (14.4) square the condition number (Equation (14.5)) and lose twice as many digits as QR (Equation (14.6)); on high-degree polynomial features they gave a wrong minimum at degree 11 and failed at degree 12, while QR did not.
- Gradient descent on least squares converges at a rate set by $\kappa(X^{\mathsf{T}}X)$; the power method gives the safe learning rate, and standardization Equation (14.7) cut κ from 546 to 1.21 and the epochs to 10^{-4} from thousands to 12.
- Mini-batch SGD, momentum Equation (14.9) and Adam Equation (14.10) trade exactness per step for cheap steps; with a fixed learning rate they end in a noise floor.
- Logistic regression has gradient $X^{\mathsf{T}}(\mathbf{p}-\mathbf{t})/N$ and Hessian $X^{\mathsf{T}}WX/N$ (Equation (14.14)); its loss is convex, and Newton's method (IRLS, Equation (14.15)) converged in six steps.
- Stable formulas (Equation (14.16)) keep the log-loss finite; gradient checks catch derivative errors; float32 has about 7 digits (Equation (14.17)) and needs care in sums and subtractions.
- For Mas Dimas: trip time is about 4.0 minutes plus 2.6 per kilometer, plus 5.7 at rush hour and 5.5 in rain; each kilometer of pickup multiplies the odds of a cancellation by 3.7.

14.12 Exercises

1. Fit $\hat{y} = w_0 + w_1$ km by hand to the three trips of Section 14.2 with the distance standardized ($u = -1.2247, 0, 1.2247$), and show that converting back gives the same line $4.333 + 2.667$ km.
2. Show that the Hessian of the least-squares loss Equation (14.2) is $X^{\mathsf{T}}X/N$, and that after standardizing the features and including a constant column, the first row and column of $X^{\mathsf{T}}X/N$ are $(1, 0, \ldots, 0)$.
3. Derive Equation (14.13) from Equation (14.12), and then the gradient in Equation (14.14).
4. Mas Dimas records his pickup distance in meters instead of kilometers. Which weight of the logistic model changes and how? What happens to the condition number of $X^{\mathsf{T}}X/N$ and to the number of gradient-descent epochs, and does Newton's method care?
5. Add ridge regularization to `normal_equations` (a parameter `lam`, adding `N * lam` to the diagonal except the first entry) and rerun the degree-12 polynomial in raw kilometers with $\lambda = 10^{-8}$ and with $\lambda = 10^{-6}$. Does Cholesky still fail? Explain with eigenvalues.
6. Using the learned weights, find the pickup distance at which a rush-hour order in the rain has a 50% chance of being cancelled.

14.13 Solutions to the Exercises

1. With $u = (-1.2247, 0, 1.2247)$: $\sum 1 = 3$, $\sum u = 0$ and $\sum u^2 = 2 \times 1.2247^2 = 3.0$. The normal equations are $3w_0 = 53$ and $3.0\,w_1 = \sum u_i y_i = -1.2247 \times 10 + 0 + 1.2247 \times 26 = 19.596$, so $w_0 = 17.667$ and $w_1 = 6.532$, found without any elimination because the matrix is diagonal.

Converting back with $\mu = 5$ and $\sigma = \sqrt{6} = 2.4495$: the slope per kilometer is $6.532/2.4495 = 2.667$ and the constant is $17.667 - 2.667 \times 5 = 4.333$, the same line as before.

2. $L = \frac{1}{2N}\sum_i (\mathbf{x}_i^\top \mathbf{w} - y_i)^2$, so $\partial L/\partial w_j = \frac{1}{N}\sum_i (\mathbf{x}_i^\top \mathbf{w} - y_i)\, x_{ij}$ and $\partial^2 L/\partial w_j\, \partial w_k = \frac{1}{N}\sum_i x_{ij}\, x_{ik}$, which is entry (j,k) of $X^\top X/N$. With a constant column $x_{i0} = 1$ and standardized features, entry (0,0) is $\frac{1}{N}\sum 1 = 1$ and entry $(0,k)$ is $\frac{1}{N}\sum_i u_{ik}$, the mean of a standardized feature, which is 0. So the constant decouples from the other weights, which is one reason standardization helps.

3. With $p = \sigma(z)$: $\ln p = -\ln(1 + e^{-z})$ and $\ln(1-p) = \ln\frac{e^{-z}}{1+e^{-z}} = -z - \ln(1+e^{-z})$. Then $-[t\ln p + (1-t)\ln(1-p)] = t\ln(1+e^{-z}) + (1-t)(z + \ln(1+e^{-z})) = \ln(1+e^{-z}) + z - tz$. Since $\ln(1+e^{-z}) + z = \ln(e^z(1+e^{-z})) = \ln(1+e^z) = \text{softplus}(z)$, the loss per trip is $\text{softplus}(z) - tz$, which is Equation (14.13). Differentiating with respect to w_j, with $\partial z/\partial w_j = x_j$ and $\text{softplus}'(z) = e^z/(1+e^z) = \sigma(z)$, gives $(\sigma(z) - t)\, x_j$; averaging over trips gives $\frac{1}{N}X^\top(\mathbf{p} - \mathbf{t})$.

4. The pickup feature is multiplied by 1000, so its weight is divided by 1000 (1.2994 becomes 0.0012994), and the predictions do not change. The column of X is 1000 times larger, the corresponding entries of $X^\top X/N$ grow by 10^3 and 10^6, and $\lambda_{\max}$ becomes about 5.6×10^6 (the mean of the squared pickup distance in meters). The safe learning rate falls to about 10^{-6} while the directions of the other weights still need steps of order 1, so gradient descent would need millions of epochs. Newton's method does not care: it is invariant under such rescaling, because $H\mathbf{s} = \mathbf{g}$ changes consistently on both sides, and it still converges in six steps. Standardization makes gradient descent behave the same way.

5. With the ridge term, the matrix is $X^\top X + N\lambda D$, with D the identity with a zero in the first position; adding $N\lambda$ to the diagonal lifts the eigenvalues of the non-constant directions by about that amount. With $\lambda = 10^{-8}$ the lift is $400 \times 10^{-8} = 4 \times 10^{-6}$, and Cholesky *still fails* at column 12: the diagonal entry of that column is $\sum d^{24}$, about 3×10^{29}, and the rounding errors in its last pivot are far larger than 4×10^{-6}. With $\lambda = 10^{-6}$ the lift is 4×10^{-4}, enough to keep the last pivot positive, and the fit succeeds with RMSE 4.500724, a little above QR's unregularized 4.494450 because the penalty pulls the weights toward zero. The lesson is that a ridge parameter has no meaning independent of the units: on raw powers, the right λ is found only by trial, while on standardized features, where every diagonal entry of $X^\top X/N$ is 1, $\lambda = 10^{-3}$ means the same thing in every problem.

6. Probability 0.5 means $z = 0$: $-4.8074 + 1.2994\, d + 1.0072 + 1.1131 = 0$, so $1.2994\, d = 2.6871$ and $d = 2.07$ km. For pickups farther than about 2 km in the rain at rush hour, a cancellation is more likely than not, in agreement with the red curve of **Figure 14.2**.

References

[1] S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers*, 8th ed. New York, NY, USA: McGraw-Hill Education, 2021.

[2] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed. Baltimore, MD, USA: Johns Hopkins University Press, 2013.

[3] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: SIAM, 2002.

[4] L. N. Trefethen and D. Bau, *Numerical Linear Algebra*, twenty-fifth anniversary ed. Philadelphia, PA, USA: SIAM, 2022.

[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[6] L. Bottou, F. E. Curtis, and J. Nocedal, “Optimization methods for large-scale machine learning,” *SIAM Review*, vol. 60, no. 2, pp. 223–311, 2018.

[7] B. T. Polyak, “Some methods of speeding up the convergence of iteration methods,” *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1–17, 1964.

[8] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *Proc. 3rd International Conference on Learning Representations (ICLR)*, San Diego, CA, USA, 2015. [Online]. Available: https://arxiv.org/abs/1412.6980

[9] P. McCullagh and J. A. Nelder, *Generalized Linear Models*, 2nd ed. London, U.K.: Chapman and Hall, 1989.

[10] P. Blanchard, D. J. Higham, and N. J. Higham, “Accurately computing the log-sum-exp and softmax functions,” *IMA Journal of Numerical Analysis*, vol. 41, no. 4, pp. 2311–2330, 2021.

[11] IEEE, *IEEE Standard for Floating-Point Arithmetic*, IEEE Std 754-2019, 2019.

[12] NumPy Developers, “numpy.linalg.lstsq,” NumPy 2.5 documentation, 2026. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.linalg.lstsq.html

[13] The SciPy community, “scipy.special.log_expit,” SciPy 1.18.1 documentation, 2026. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.special.log_expit.html

[14] NumPy Developers, “numpy 2.5.3,” Python Package Index, Sep. 6, 2026. [Online]. Available: https://pypi.org/project/numpy/

[15] SciPy Developers, “scipy 1.18.1,” Python Package Index, Aug. 21, 2026. [Online]. Available: https://pypi.org/project/scipy/

[16] Profitina, “Profitina — AI Visibility (GEO) platform,” 2026. [Online]. Available: https://profitina.com

Chapter 15

Fast and Parallel Numerical Computing

A numerical method that is correct but takes an hour is, for most people, a method they will not use. This chapter takes one computation from Chapter 5, the temperature of a martabak griddle, refines it until it becomes slow, and then makes it fast in four different ways: by removing the interpreter from the inner loop, by compiling, by splitting the work over several processor cores, and, most effectively, by choosing a better algorithm. On the way it meets the one numerical danger that parallel computing adds to all the others: the same sum, added in a different order, gives a different answer. By the end of the chapter you will be able to estimate the cost of a computation in operations and in memory traffic before you run it; measure it honestly; vectorize a loop with NumPy and explain why that helps; split a computation over threads in Java and Python without changing its result; predict the best possible speedup with Amdahl's law and explain why real speedups fall short; recognize which methods parallelize safely (Jacobi) and which need rearranging (Gauss–Seidel, rearranged as red-black ordering); make parallel sums accurate and reproducible with pairwise and Kahan summation; and judge when a better algorithm beats more hardware.

15.1 An Hour for One Picture

In Chapter 5, Pak Joni wanted to know how the heat spreads over the square iron griddle of his martabak cart, forty centimeters on a side, with the four edges held at 200, 60, 80 and 150 degrees Celsius. Jacobi's method, Gauss–Seidel and SOR computed the steady temperature on a coarse grid of points, and the picture answered his question: where the plate is hot enough for martabak manis and where it is only warm enough to keep roti bakar from cooling.

Now a blacksmith in the same market, who makes griddles for a dozen carts, asks for more. He wants to try different burner positions and plate sizes, and he wants the temperature every millimeter, because a cold spot of one centimeter is where a martabak stays pale. Every millimeter on a 40 cm plate means a grid of $400 \times 400 = 160{,}000$ unknown temperatures. The Jacobi program of Chapter 5, written with plain loops, does about 15 million point updates per second in Python on the computer used for this chapter, which sounds fast. But Section 15.3.4 shows that Jacobi needs about 340,000 sweeps over the grid to settle at this resolution, and $160{,}000 \times 340{,}000 = 5.4 \times 10^{10}$ updates at 15 million per second is **an hour** for one picture. The blacksmith wants twenty pictures before lunch.

The same situation arises everywhere numerical methods leave the textbook. A weather model divides Indonesia into millions of cells; an irrigation plan simulates a thousand possible dry seasons (Chapter 16 does exactly that); a machine-learning model, as Chapter 14 showed, passes over millions of rows many times. Making such computations fast is partly engineering, but the decisions that matter most are numerical: how much work the method does, how that work can be divided, and whether dividing it changes the answer.

15.2 The Idea in Plain Words

15.2.1 Where the time goes

A computer spends time on three things. It does *arithmetic*: a modern core performs a few billion additions or multiplications per second, more if it applies one instruction to several numbers at once. It *moves data*: numbers must travel from main memory through several levels of cache to the arithmetic units, and main memory delivers only a few billion numbers per second, shared by all cores. And it does *bookkeeping*: an interpreted language such as Python checks the type of every number, looks up every operation and allocates every result, which costs tens of nanoseconds for each single addition.

For the Jacobi update of one griddle point, $u_{\text{new}} = 1/4\,(u_N + u_S + u_W + u_E)$, the arithmetic is 4 operations (three additions and one multiplication). In plain Python the bookkeeping around those 4 operations costs about 65 nanoseconds (1/15.6 million seconds), roughly a hundred times more than the arithmetic. The first lesson is therefore simple: in Python, move the loop out of the interpreter. NumPy does this: one line of array arithmetic runs a compiled loop over all points, and the program becomes about ten times faster. Java, whose programs are compiled to machine code while they run, is faster again by about ten.

15.2.2 Doing several things at once

A computer with several cores can update several parts of the griddle at once. Split the 400 rows into two strips of 200; one core updates the top strip, the other the bottom. If the work divides perfectly, two cores take half the time.

Whether a method can be split like this depends on what each update reads. In a Jacobi sweep, every new value is computed from the *old* grid only, so the order of updates does not matter and any division into strips gives exactly the same numbers, bit for bit. Gauss–Seidel is different: each update reads neighbors that were *just* updated, so the answer depends on the order, and two cores working on neighboring rows would read each other's half-finished values. The cure, Section 15.3.3 shows, is to color the points like a chessboard and update all red points, then all black points.

15.2.3 The same sum, a different answer

Parallel work often ends with a sum: the mean temperature of the griddle, the total water used in a season, the loss over all rows of a data set. A parallel sum lets each core add up its own chunk and then adds the partial sums. In exact arithmetic the result is the same. In floating point it is not, because floating-point addition is not associative (Chapter 1). A tiny example with double precision: $(10^{16} + 1) - 10^{16} = 0$, because $10^{16} + 1$ rounds back to 10^{16}, while $(10^{16} - 10^{16}) + 1 = 1$. Adding ten million copies of 0.1 one after another gives 999999.99983897, but splitting the same additions into four chunks gives 1000000.00004026. A program whose answer changes with the number of cores it happens to run on is hard to test and hard to trust; Section 15.3.5 shows how to make the answer both accurate and independent of the number of cores.

> **KEY INSIGHT:** There are three ways to make a computation faster, and they should be tried in this order: *do less work* (a better algorithm, often a factor of hundreds), *do the work with less overhead* (vectorized or compiled code, a factor of ten to a hundred), and *do the work at once* (parallel cores, at best a factor equal to the number of cores, usually less).

15.3 The Methods, Derived

15.3.1 A cost model: operations and memory

Let a computation perform W arithmetic operations and move Q bytes between main memory and the processor. With a peak arithmetic rate F (operations per second) and a memory bandwidth B (bytes per second), the time is at least

$$T \geq \max\left(\frac{W}{F}, \frac{Q}{B}\right). \tag{15.1}$$

The ratio $I = W/Q$, operations per byte, is the *arithmetic intensity*. When $I < F/B$ the computation is *memory-bound*: faster arithmetic or more cores do not help once the memory channel is full. This simple bound, drawn as a "roofline" on a log-log chart, is how performance engineers decide what to optimize [1].

A Jacobi sweep on an $n \times n$ grid performs $W = 4n^2$ operations. Each point reads four neighbors, but three of them were already read for nearby points and are still in cache, so in the best case each sweep streams the old grid in and the new grid out: $Q \approx 2 \times 8n^2 = 16n^2$ bytes. The intensity is $I = 4/16 = 0.25$ operations per byte. A typical core computes $F \approx 10^{10}$ operations per second while a typical memory system delivers $B \approx 2 \times 10^{10}$ bytes per second shared by all cores, so $F/B \approx 0.5$ for *one* core and less for several: Jacobi is memory-bound, and the bound on update rate is about $B/16 \approx 1.25 \times 10^9$ updates per second. The Java program measured 1.36 billion updates per second on one core, at this bound (the grid of 400 rows, 1.3 MB per array, partly fits in the large cache of the test machine), and Section 15.5 shows that a second core did not help. That is Equation (15.1) speaking: when memory is the bottleneck, more cores only queue at the same door.

15.3.2 Vectorization

Vectorization means expressing a computation as operations on whole arrays instead of loops over elements. In NumPy, the whole Jacobi sweep is one statement:

```
v[1:-1, 1:-1] = 0.25 * (u[:-2, 1:-1] + u[2:, 1:-1] + u[1:-1, :-2] + u[1:-1, 2:])
```

Each slice such as `u[:-2, 1:-1]` is the grid shifted by one row or column, a *view* of the same memory without copying. The four additions and the multiplication each run as a compiled loop, so the interpreter's bookkeeping is paid five times per sweep instead of five times per point [2]. The price is temporary arrays for intermediate sums, which add memory traffic; this is why NumPy, at about 130 million updates per second, is ten times faster than Python loops but ten times slower than Java, whose compiled loop keeps the four values in registers.

The order of operations inside the expression is exactly that of the loop, $((u_N + u_S) + u_W) + u_E$ times 0.25, and each is rounded the same way, so the vectorized program prints the same digits as the loop program and as Java. Vectorization changes the speed, not the arithmetic. (A library routine that sums an array is another matter: NumPy's `sum` adds pairwise, not left to right, which is why the programs of this chapter print sums with a sensible number of digits and why Section 15.3.5 matters.)

Modern processors can also apply one instruction to 4 or 8 doubles at once (SIMD, single instruction multiple data). Compiled loops such as Java's use these instructions when the compiler can prove it is safe; Java also offers an explicit Vector API, still an incubating feature in JDK 25 (its tenth incubator round) and in JDK 26 [3].

15.3.3 Parallel speedup and its limits

Suppose a fraction f of a program's running time can be divided perfectly among p cores and the rest, $1-f$, cannot (reading input, starting threads, combining results, and the parts nobody parallelized). The time on p cores is $T_p = T_1\,[(1-f) + f/p]$, and the *speedup* is

$$S(p) = \frac{T_1}{T_p} = \frac{1}{(1-f) + f/p} \le \frac{1}{1-f}. \tag{15.2}$$

This is *Amdahl's law* [4]. With $f = 0.95$, eight cores give $S = 1/(0.05 + 0.11875) = 5.93$, and no number of cores gives more than 20. The serial part, however small, eventually dominates. Gustafson observed that in practice people use more cores to solve *bigger* problems in the same time, and for many numerical methods the parallel part grows with the problem while the serial part does not, so the achievable speedup grows too [5]. For the griddle both views apply: at $n = 100$ the per-sweep coordination between threads costs more than the work it divides, and at $n = 400$ or more the work dominates.

Real speedups fall below Equation (15.2) for three further reasons. *Synchronization*: every Jacobi sweep must finish on all strips before the next begins, so each sweep ends with the threads waiting for the slowest one. *Memory bandwidth*: as Equation (15.1) showed, memory-bound work stops scaling when the memory channel is full. *Overhead*: creating tasks and passing them to threads costs microseconds, which matters when a sweep takes only a hundred.

Which computations may be divided at all? A division is safe when no part writes a value that another part reads during the same step. Jacobi reads only the old grid `u` and writes only the new grid `v`, so strips never interfere: this is why the program of Section 15.4 gives identical results with one thread or many. Gauss–Seidel updates in place, so the value of point (i, j) depends on whether its upper neighbor has been updated yet; two threads on neighboring strips would produce results that depend on timing, a *race condition*. *Red-black ordering* removes the dependence. Color point (i, j) red when $i + j$ is even and black when it is odd, like a chessboard. Every neighbor of a red point is black and vice versa. So an update of all red points reads only black points, which do not change during that half-sweep, and may be done in any order and in parallel; then all black points are updated from the new red ones. With over-relaxation this is *red-black SOR*, which converges exactly as fast as SOR in the natural order on this problem [6].

> **EXAMPLE:** A sweep by hand, split between two threads. Take a griddle with only 3×3 inner points, all starting at 0, and the edges of Chapter 5 (left 200, right 60, top 80, bottom 150). One Jacobi sweep gives, row by row: $1/4\,(80 + 200) = 70$, $1/4 \cdot 80 = 20$, $1/4\,(80 + 60) = 35$; then 50, 0, 15; then $1/4\,(150 + 200) = 87.5$, $1/4 \cdot 150 = 37.5$, $1/4\,(150 + 60) = 52.5$. Give row 1 to thread A and rows 2 and 3 to thread B. Each thread reads only the old grid, which nobody changes during the sweep, so it makes no difference whether A finishes before B starts, after B finishes, or at the same time: the nine numbers are always the same. Now measure what the split buys. For red-black SOR on the 400×400 grid, Java took 0.357 seconds on one thread and 0.266 on two, a speedup of $S = 1.34$. Solving Equation (15.2) with $p = 2$ for the parallel fraction, $1/S = (1-f) + f/2$ gives $f = 2(1 - 1/1.34) = 0.51$: in effect only half of the time behaved as if it were divided, the rest being memory traffic and synchronization that the second core could not share.

15.3.4 Doing less work: the number of sweeps

All of the above makes each sweep cheaper. The number of sweeps depends on the method. Chapter 5 showed that the error of Jacobi on the $n \times n$ model problem shrinks per sweep by the

spectral radius $\rho_J = \cos(\pi/(n+1))$ [7]. For large n, $\cos x \approx 1 - x^2/2$ gives $1 - \rho_J \approx \pi^2/(2(n+1)^2)$, and reducing the error by a factor ε takes

$$k_J \approx \frac{\ln(1/\varepsilon)}{1-\rho_J} \approx \frac{2(n+1)^2}{\pi^2} \ln\frac{1}{\varepsilon} \tag{15.3}$$

sweeps. For optimally relaxed SOR, $\rho = \omega_{\text{opt}} - 1 \approx 1 - 2\pi/(n+1)$, so

$$k_{\text{SOR}} \approx \frac{n+1}{2\pi} \ln\frac{1}{\varepsilon}. \tag{15.4}$$

Jacobi's count grows with n^2 and SOR's with n; their ratio is about $4(n+1)/\pi$, which is 130 for $n = 100$ and 510 for $n = 400$. Doubling the resolution quadruples Jacobi's sweeps *and* quadruples the points per sweep: the total work grows like n^4, sixteen times per doubling. No realistic number of cores compensates for that; a better method does. For $n = 400$ and a change below 10^{-6} per sweep, the measured Jacobi change of 3.168×10^{-2} after 2000 sweeps and the rate $\rho_J = \cos(\pi/401) = 1 - 3.07 \times 10^{-5}$ give $2000 + \ln(3.168 \times 10^4)/(3.07 \times 10^{-5}) \approx 340{,}000$ sweeps, the number used in Section 15.1. Red-black SOR stops after 1171.

Better still are methods whose total work grows only like the number of unknowns $N = n^2$: *multigrid* smooths the error on a hierarchy of coarser grids and solves the model problem in a fixed number of cycles independent of n [8]. And a *sparse direct* solver factorizes the matrix of the linear system once, like the LU factorization of Chapter 6 but ordering the unknowns cleverly (nested dissection) so that the factors stay sparse; for a 2-D grid its cost grows like $N^{3/2}$ [9]. SciPy's `spsolve` does this in under a second for 160,000 unknowns, as Section 15.6 shows.

15.3.5 Accurate and reproducible sums

Add n numbers $x_1, \ldots, x_n$ left to right in floating point with unit roundoff u (Chapter 1). Each addition makes a relative error of at most u, and the error made at step k is carried to the end, so the computed sum $\hat{s}$ obeys [10]

$$|\hat{s} - s| \;\le\; (n-1)\,u \sum_{i=1}^{n} |\,x_i\,| + O(u^2). \tag{15.5}$$

For ten million copies of 0.1 the bound is $10^7 \times 1.1 \times 10^{-16} \times 10^6 \approx 1.1 \times 10^{-3}$; the actual error, 1.6×10^{-4}, is smaller, because rounding errors partly cancel, but it grows with n. The mechanism is easy to see: once the running sum is near 10^6, the spacing of doubles there is 1.16×10^{-10}, and every added 0.1 is rounded to a multiple of that spacing, losing about 10^{-11} each time.

Pairwise summation adds the two halves separately and then adds the results, recursively. Every number then takes part in only about $\log_2 n$ additions, and the bound becomes

$$|\hat{s} - s| \;\le\; \lceil \log_2 n \rceil\, u \sum_{i=1}^{n} |\,x_i\,| + O(u^2), \tag{15.6}$$

a factor $n/\log_2 n$ smaller: for $n = 10^7$, 24 instead of ten million. Pairwise summation is naturally parallel, since the two halves can be added on different cores, and it is what NumPy's `sum` uses [2]. Python's built-in `sum` went further in version 3.12: for floats it now uses a compensated algorithm

of the kind described next, so `sum([0.1] * 10_000_000)` returns exactly 1000000.0 in Python 3.14, while a plain loop does not [11].

Kahan's compensated summation [12] keeps a second variable c holding the low-order part lost by the last addition and subtracts it from the next term:

Pseudocode 15.1 Kahan's compensated summation: c recovers the rounding error of each addition and feeds it into the next.

```
KAHAN-SUM(x[1..n])
 1  s = 0;  c = 0
 2  for i = 1 to n
 3      y = x[i] - c          // the next term, corrected by what was lost
 4      t = s + y             // big + small: low bits of y are lost here
 5      c = (t - s) - y       // (t - s) is what was really added; minus y
 6      s = t                 //   leaves exactly the part that was lost
 7  return s
```

Its error bound is $2u\sum|x_i| + O(nu^2)\sum|x_i|$, essentially independent of n [10]. It costs four operations per term instead of one, and it must not be "simplified" by an optimizing compiler that believes $(t - s) - y$ is zero; Java and Python respect the written order, but C compilers with aggressive floating-point flags do not.

A parallel sum splits the terms into chunks. If the chunks are determined by the number of threads, as in `chunked(xs, parts)`, the answer depends on the machine. The remedy is to fix the chunking by the *data*, not by the hardware: for example always 64 chunks, summed pairwise or with Kahan, and the 64 partial sums combined in a fixed order, however many threads compute them. Then one core and sixty-four give the same bits.

15.4 The Programs

The programs are in `Chapter15/python` and `Chapter15/java`. Each prints the same text in both languages; the optional argument `--time` adds a line with the measured time, which of course differs from machine to machine and run to run.

15.4.1 Summation

Listing 15.1 adds ten million copies of 0.1 in seven ways: left to right, in 2, 4, 8 and 16 chunks (the way a parallel sum with that many threads would), pairwise and with Kahan's method (**Pseudocode 15.1**). The chunks are summed one after the other here, so the program is deterministic, but the arithmetic is exactly that of a parallel run.

Listing 15.1 `summation.py`: naive, chunked, pairwise and Kahan summation.

```
"""Ten million sums of 0.1: why the order of additions, and so the number of
threads, changes the answer, and how to make it not matter.
Usage: python summation.py [n]   (default n = 10000000)"""
import sys


def naive(xs, lo, hi):
    s = 0.0
    for i in range(lo, hi):
        s += xs[i]
    return s


def kahan(xs, lo, hi):
    """Compensated summation: c carries the low-order bits that s loses."""
    s = c = 0.0
    for i in range(lo, hi):
        y = xs[i] - c
        t = s + y
        c = (t - s) - y
        s = t
    return s


def pairwise(xs, lo, hi):
    """Sum each half separately and add the two halves (error ~ log n)."""
    if hi - lo <= 8:
        return naive(xs, lo, hi)
    mid = (lo + hi) // 2
    return pairwise(xs, lo, mid) + pairwise(xs, mid, hi)


def chunked(xs, parts):
    """What a parallel sum does: each thread sums one chunk, then add them."""
    n, total = len(xs), 0.0
    for p in range(parts):
        total += naive(xs, p * n // parts, (p + 1) * n // parts)
    return total
```

Listing 15.1 (Continued) *summation.py: naive, chunked, pairwise and Kahan summation.*

```
def main():
    n = int(sys.argv[1]) if len(sys.argv) > 1 else 10_000_000
    xs = [0.1] * n
    exact = n / 10
    print(f"sum of {n} copies of 0.1; exact value {exact:.1f}")
    rows = [("naive, one thread", naive(xs, 0, n))]
    for parts in (2, 4, 8, 16):
        rows.append((f"naive, {parts} chunks", chunked(xs, parts)))
    rows += [("pairwise", pairwise(xs, 0, n)), ("Kahan", kahan(xs, 0, n))]
    for name, s in rows:
        print(f"{name:18s} {s:.10f}  error {s - exact:+.3e}")


if __name__ == "__main__":
    main()
```

Listing 15.2 is the Java version; `Arrays.fill` makes the ten million copies, and the chunk boundaries are computed in `long` arithmetic so that `p * n` cannot overflow.

Listing 15.2 `Summation.java`: the same sums in Java.

```java
// Ten million sums of 0.1: why the order of additions, and so the number of
// threads, changes the answer, and how to make it not matter.
// Usage: java Summation [n]   (default n = 10000000)
import java.util.Arrays;
import java.util.Locale;

public class Summation {
    static double naive(double[] xs, int lo, int hi) {
        double s = 0.0;
        for (int i = lo; i < hi; i++) s += xs[i];
        return s;
    }

    /** Compensated summation: c carries the low-order bits that s loses. */
    static double kahan(double[] xs, int lo, int hi) {
        double s = 0.0, c = 0.0;
        for (int i = lo; i < hi; i++) {
            double y = xs[i] - c;
            double t = s + y;
            c = (t - s) - y;
            s = t;
        }
        return s;
    }

    /** Sum each half separately and add the two halves (error ~ log n). */
    static double pairwise(double[] xs, int lo, int hi) {
        if (hi - lo <= 8) return naive(xs, lo, hi);
        int mid = (lo + hi) / 2;
        return pairwise(xs, lo, mid) + pairwise(xs, mid, hi);
    }

    /** What a parallel sum does: each thread sums one chunk, then add them. */
    static double chunked(double[] xs, int parts) {
        long n = xs.length;
        double total = 0.0;
        for (int p = 0; p < parts; p++)
            total += naive(xs, (int) (p * n / parts), (int) ((p + 1) * n / parts));
        return total;
    }
```

Listing 15.2 (Continued) *Summation.java: the same sums in Java.*

```java
    public static void main(String[] args) {
        int n = args.length > 0 ? Integer.parseInt(args[0]) : 10_000_000;
        double[] xs = new double[n];
        Arrays.fill(xs, 0.1);
        double exact = n / 10.0;
        System.out.println(String.format(Locale.ROOT,
                "sum of %d copies of 0.1; exact value %.1f", n, exact));
        String[] names = {"naive, one thread", "naive, 2 chunks", "naive, 4 chunks",
                "naive, 8 chunks", "naive, 16 chunks", "pairwise", "Kahan"};
        double[] sums = {naive(xs, 0, n), chunked(xs, 2), chunked(xs, 4),
                chunked(xs, 8), chunked(xs, 16), pairwise(xs, 0, n),
                kahan(xs, 0, n)};
        for (int k = 0; k < names.length; k++)
            System.out.println(String.format(Locale.ROOT,
                    "%-18s %.10f  error %+.3e",
                    names[k], sums[k], sums[k] - exact));
    }
}
```

15.4.2 The griddle with loops

Listing 15.3 is the Jacobi method of Chapter 5, rewritten for speed in plain Python: two grids are kept and swapped after every sweep instead of copying, and the rows above, at and below the current one are fetched once per row. It runs a fixed number of sweeps, so that the implementations can be timed on the same work, and reports the temperature at the center, the mean temperature and the largest change in the last sweep.

Listing 15.3 `griddle_loops.py`: Jacobi sweeps on the fine griddle with plain loops.

```
"""Pak Joni's griddle (Chapter 5) on a fine grid, Jacobi with plain Python loops.
Usage: python griddle_loops.py [n] [sweeps] [--time]   (default 100 1000)"""
import sys
import time

LEFT, RIGHT, TOP, BOTTOM = 200.0, 60.0, 80.0, 150.0   # edge temperatures, C


def new_grid(n):
    u = [[0.0] * (n + 2) for _ in range(n + 2)]
    for i in range(1, n + 1):
        u[i][0], u[i][n + 1] = LEFT, RIGHT
        u[0][i], u[n + 1][i] = TOP, BOTTOM
    return u


def jacobi(u, n, sweeps):
    """Each sweep: every inner point becomes the mean of its OLD neighbors."""
    v = [row[:] for row in u]
    for _ in range(sweeps):
        for i in range(1, n + 1):
            up, row, down, out = u[i - 1], u[i], u[i + 1], v[i]
            for j in range(1, n + 1):
                out[j] = 0.25 * (up[j] + down[j] + row[j - 1] + row[j + 1])
        u, v = v, u                              # swap old and new
    return u, v


def report(u, v, n):
    inner = [u[i][j] for i in range(1, n + 1) for j in range(1, n + 1)]
    change = max(abs(u[i][j] - v[i][j])
                 for i in range(1, n + 1) for j in range(1, n + 1))
    c = n // 2
    print(f"center {u[c][c]:.6f} C, mean {sum(inner) / len(inner):.6f} C,"
          f" last change {change:.3e}")
```

Listing 15.3 (Continued) *griddle_loops.py: Jacobi sweeps on the fine griddle with plain loops.*

```
def main():
    args = [a for a in sys.argv[1:] if a != "--time"]
    n = int(args[0]) if args else 100
    sweeps = int(args[1]) if len(args) > 1 else 1000
    print(f"grid {n} x {n}, {sweeps} Jacobi sweeps, 1 thread(s)")
    t0 = time.perf_counter()
    u, v = jacobi(new_grid(n), n, sweeps)
    t = time.perf_counter() - t0
    report(u, v, n)
    if "--time" in sys.argv:
        print(f"time {t:.3f} s, {n * n * sweeps / t / 1e6:.1f} million updates/s")


if __name__ == "__main__":
    main()
```

15.4.3 Vectorized, and split over threads

Listing 15.4 does each sweep with the one NumPy statement of Section 15.3.2, applied to a block of rows. With more than one thread, a `ThreadPoolExecutor` gives each thread a strip of rows; NumPy releases Python's global interpreter lock while it computes on arrays, so the strips really run at the same time. Because Jacobi reads only the old grid, the result is the same for any number of threads.

Listing 15.4 `griddle_numpy.py`: vectorized Jacobi sweeps, optionally split over threads.

```
"""The same Jacobi sweeps vectorized with NumPy, and split over threads.
Usage: python griddle_numpy.py [n] [sweeps] [threads] [--time]"""
import sys
import time
from concurrent.futures import ThreadPoolExecutor
import numpy as np
from griddle_loops import LEFT, RIGHT, TOP, BOTTOM


def new_grid(n):
    u = np.zeros((n + 2, n + 2))
    u[1:-1, 0], u[1:-1, -1], u[0, 1:-1], u[-1, 1:-1] = LEFT, RIGHT, TOP, BOTTOM
    return u


def sweep_rows(u, v, lo, hi):
    """Jacobi update of inner rows lo..hi-1: one line does a whole block."""
    v[lo:hi, 1:-1] = 0.25 * (u[lo - 1:hi - 1, 1:-1] + u[lo + 1:hi + 1, 1:-1]
                             + u[lo:hi, :-2] + u[lo:hi, 2:])


def jacobi(n, sweeps, threads=1):
    u = new_grid(n)
    v = u.copy()
    cuts = [1 + k * n // threads for k in range(threads + 1)]
    with ThreadPoolExecutor(max_workers=threads) as pool:
        for _ in range(sweeps):
            if threads == 1:
                sweep_rows(u, v, 1, n + 1)
            else:                           # every thread takes a strip of rows
                list(pool.map(lambda k: sweep_rows(u, v, cuts[k], cuts[k + 1]),
                              range(threads)))
            u, v = v, u
    return u, v
```

Listing 15.4 (Continued) *griddle_numpy.py: vectorized Jacobi sweeps, optionally split over threads.*

```
def main():
    args = [a for a in sys.argv[1:] if a != "--time"]
    n = int(args[0]) if args else 100
    sweeps = int(args[1]) if len(args) > 1 else 1000
    threads = int(args[2]) if len(args) > 2 else 1
    print(f"grid {n} x {n}, {sweeps} Jacobi sweeps, {threads} thread(s)")
    t0 = time.perf_counter()
    u, v = jacobi(n, sweeps, threads)
    t = time.perf_counter() - t0
    change = np.max(np.abs(u[1:-1, 1:-1] - v[1:-1, 1:-1]))
    c = n // 2
    print(f"center {u[c, c]:.6f} C, mean {np.mean(u[1:-1, 1:-1]):.6f} C,"
          f" last change {change:.3e}")
    if "--time" in sys.argv:
        print(f"time {t:.3f} s, {n * n * sweeps / t / 1e6:.1f} million updates/s")


if __name__ == "__main__":
    main()
```

Listing 15.5 is the Java version. A fixed pool of threads is created once; in every sweep each thread receives a strip of rows as a task, and `invokeAll` waits until all strips are done before the grids are swapped. That wait is the synchronization cost of Section 15.3.3, paid once per sweep.

Listing 15.5 `Griddle.java`: Jacobi sweeps in Java on one thread or several.

```java
// Pak Joni's griddle (Chapter 5) on a fine grid: Jacobi sweeps, one thread or
// many. Usage: java Griddle [n] [sweeps] [threads] [--time]   (default 100 1000 1)
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Griddle {
    static final double LEFT = 200.0, RIGHT = 60.0, TOP = 80.0, BOTTOM = 150.0;

    static double[][] newGrid(int n) {
        double[][] u = new double[n + 2][n + 2];
        for (int i = 1; i <= n; i++) {
            u[i][0] = LEFT;
            u[i][n + 1] = RIGHT;
            u[0][i] = TOP;
            u[n + 1][i] = BOTTOM;
        }
        return u;
    }

    /** Jacobi update of inner rows lo..hi-1 from the old grid u into v. */
    static void sweepRows(double[][] u, double[][] v, int n, int lo, int hi) {
        for (int i = lo; i < hi; i++) {
            double[] up = u[i - 1], row = u[i], down = u[i + 1], out = v[i];
            for (int j = 1; j <= n; j++)
                out[j] = 0.25 * (up[j] + down[j] + row[j - 1] + row[j + 1]);
        }
    }

    public static void main(String[] args) throws Exception {
        List<String> a = new ArrayList<>(List.of(args));
        boolean timing = a.remove("--time");
        int n = a.size() > 0 ? Integer.parseInt(a.get(0)) : 100;
        int sweeps = a.size() > 1 ? Integer.parseInt(a.get(1)) : 1000;
        int threads = a.size() > 2 ? Integer.parseInt(a.get(2)) : 1;
        System.out.println(String.format(Locale.ROOT,
                "grid %d x %d, %d Jacobi sweeps, %d thread(s)",
                n, n, sweeps, threads));
        long t0 = System.nanoTime();
        double[][] u = newGrid(n), v = newGrid(n);
        ExecutorService pool = Executors.newFixedThreadPool(threads);
        for (int s = 0; s < sweeps; s++) {
            if (threads == 1) {
```

Listing 15.5 (Continued) *Griddle.java: Jacobi sweeps in Java on one thread or several.*

```
                sweepRows(u, v, n, 1, n + 1);
            } else {                          // every thread takes a strip of rows
                List<Callable<Object>> jobs = new ArrayList<>();
                for (int k = 0; k < threads; k++) {
                    final double[][] uu = u, vv = v;
                    final int lo = 1 + k * n / threads;
                    final int hi = 1 + (k + 1) * n / threads;
                    jobs.add(() -> { sweepRows(uu, vv, n, lo, hi); return null; });
                }
                pool.invokeAll(jobs);         // waits until all strips are done
            }
            double[][] t = u; u = v; v = t;   // swap old and new
        }
        pool.shutdown();
        double t = (System.nanoTime() - t0) / 1e9;
        double sum = 0.0, change = 0.0;
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++) {
                sum += u[i][j];
                change = Math.max(change, Math.abs(u[i][j] - v[i][j]));
            }
        int c = n / 2;
        System.out.println(String.format(Locale.ROOT,
                "center %.6f C, mean %.6f C, last change %.3e",
                u[c][c], sum / (n * n), change));
        if (timing)
            System.out.println(String.format(Locale.ROOT,
                    "time %.3f s, %.1f million updates/s",
                    t, (double) n * n * sweeps / t / 1e6));
    }
}
```

15.4.4 Fewer sweeps: red-black SOR

Listing 15.6 replaces faster sweeps by fewer sweeps. It builds the two color masks once and, in each half-sweep, computes the SOR value for all inner points in one vectorized expression and stores it only where the mask is true. This does twice the necessary arithmetic (it computes both colors and keeps one), a deliberate trade of arithmetic for simplicity that Exercise 15.5 revisits. The relaxation factor is Chapter 5's optimal $\omega = 2/(1 + \sin(\pi/(n+1)))$, and the iteration stops when no point moves by 10^{-6} or more.

Listing 15.6 `griddle_rb.py`: red-black SOR with NumPy.

```
"""Fewer sweeps instead of faster sweeps: red-black SOR with NumPy.
Usage: python griddle_rb.py [n] [--time]   (default n = 100)"""
import math
import sys
import time
import numpy as np
from griddle_numpy import new_grid


def red_black_sor(n, omega, tol=1e-6, max_sweeps=100000):
    """Points with i + j even ('red') depend only on 'black' points and vice
    versa, so each half-sweep is one vectorized (and parallel) step."""
    u = new_grid(n)
    i, j = np.meshgrid(np.arange(n + 2), np.arange(n + 2), indexing="ij")
    colors = [((i + j) % 2 == c)[1:-1, 1:-1] for c in (0, 1)]
    for k in range(1, max_sweeps + 1):
        change = 0.0
        for mask in colors:
            inner = u[1:-1, 1:-1]
            gs = 0.25 * (u[:-2, 1:-1] + u[2:, 1:-1] + u[1:-1, :-2] + u[1:-1, 2:])
            new = inner + omega * (gs - inner)
            change = max(change, float(np.max(np.abs(new - inner)[mask])))
            inner[mask] = new[mask]
        if change < tol:
            return u, k
    return u, max_sweeps


def main():
    args = [a for a in sys.argv[1:] if a != "--time"]
    n = int(args[0]) if args else 100
    omega = 2.0 / (1.0 + math.sin(math.pi / (n + 1)))    # Chapter 5
    t0 = time.perf_counter()
    u, k = red_black_sor(n, omega)
    t = time.perf_counter() - t0
    c = n // 2
    print(f"grid {n} x {n}, red-black SOR, omega {omega:.6f}: {k} sweeps")
    print(f"center {u[c, c]:.6f} C, mean {np.mean(u[1:-1, 1:-1]):.6f} C")
    if "--time" in sys.argv:
        print(f"time {t:.3f} s")


if __name__ == "__main__":
    main()
```

Listing 15.7 does the same in Java, visiting only the points of the current color (`j` runs in steps of 2 from the first point of that color in the row). Each thread returns the largest change in its strip, and the main thread takes the maximum. The maximum of a set of numbers, unlike their sum, does not depend on the order, so the result is again identical for any number of threads.

Listing 15.7 `GriddleRb.java`: red-black SOR in Java on one thread or several.

```java
// Fewer sweeps instead of faster sweeps: red-black SOR, one thread or many.
// Usage: java GriddleRb [n] [threads] [--time]   (default 100 1)
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class GriddleRb {
    /** Update the points of one color in rows lo..hi-1; return the largest move.
     *  A red point (i + j even) reads only black neighbors, and vice versa. */
    static double half(double[][] u, int n, double omega, int color,
                       int lo, int hi) {
        double change = 0.0;
        for (int i = lo; i < hi; i++) {
            int j0 = ((i + color) % 2 == 0) ? 2 : 1;   // first j: (i+j)%2 == color
            for (int j = j0; j <= n; j += 2) {
                double gs = 0.25 * (u[i - 1][j] + u[i + 1][j]
                        + u[i][j - 1] + u[i][j + 1]);
                double nu = u[i][j] + omega * (gs - u[i][j]);
                change = Math.max(change, Math.abs(nu - u[i][j]));
                u[i][j] = nu;
            }
        }
        return change;
    }

    public static void main(String[] args) throws Exception {
        List<String> a = new ArrayList<>(List.of(args));
        boolean timing = a.remove("--time");
        int n = a.size() > 0 ? Integer.parseInt(a.get(0)) : 100;
        int threads = a.size() > 1 ? Integer.parseInt(a.get(1)) : 1;
        double omega = 2.0 / (1.0 + Math.sin(Math.PI / (n + 1)));   // Chapter 5
        long t0 = System.nanoTime();
        double[][] u = Griddle.newGrid(n);
        ExecutorService pool = Executors.newFixedThreadPool(threads);
        int k = 0;
        while (k < 100000) {
            k++;
            double change = 0.0;
            for (int color = 0; color < 2; color++) {
                List<Callable<Double>> jobs = new ArrayList<>();
                for (int p = 0; p < threads; p++) {
                    final int lo = 1 + p * n / threads;
```

Listing 15.7 (Continued) *GriddleRb.java: red-black SOR in Java on one thread or several.*

```
                    final int hi = 1 + (p + 1) * n / threads;
                    final int col = color;
                    jobs.add(() -> half(u, n, omega, col, lo, hi));
                }
                for (Future<Double> f : pool.invokeAll(jobs))
                    change = Math.max(change, f.get());
            }
            if (change < 1e-6) break;
        }
        pool.shutdown();
        double t = (System.nanoTime() - t0) / 1e9, sum = 0.0;
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++) sum += u[i][j];
        int c = n / 2;
        System.out.println(String.format(Locale.ROOT,
                "grid %d x %d, red-black SOR, omega %.6f: %d sweeps",
                n, n, omega, k));
        System.out.println(String.format(Locale.ROOT, "center %.6f C, mean %.6f C",
                u[c][c], sum / (n * n)));
        if (timing)
            System.out.println(String.format(Locale.ROOT, "time %.3f s", t));
    }
}
```

15.5 Running the Programs

```
cd Chapter15/python
python3 summation.py
python3 griddle_loops.py
python3 griddle_numpy.py
python3 griddle_rb.py
cd ../java
javac -d out *.java
java -cp out Summation
java -cp out Griddle
java -cp out GriddleRb
```

The summation program prints:

```
sum of 10000000 copies of 0.1; exact value 1000000.0
naive, one thread  999999.9998389754  error -1.610e-04
naive, 2 chunks    999999.9999107814  error -8.922e-05
naive, 4 chunks    1000000.0000402561  error +4.026e-05
naive, 8 chunks    1000000.0000223045  error +2.230e-05
naive, 16 chunks   999999.9999899360  error -1.006e-05
pairwise           1000000.0000000000  error +0.000e+00
Kahan              1000000.0000000000  error +0.000e+00
```

Left-to-right addition misses by 1.6×10^{-4}, and every chunking gives a different wrong answer, sometimes too small and sometimes too large. A program that computed this sum with “as many threads as there are cores” would print a different result on a laptop with 4 cores and a server with 16. Pairwise and Kahan summation are both correct to all printed digits, and both would stay correct however the terms were divided, as long as the division is fixed.

The griddle programs, with the default grid of 100×100 and 1000 sweeps, all print the same three lines, whichever language and however many threads:

```
grid 100 x 100, 1000 Jacobi sweeps, 1 thread(s)
center 11.559613 C, mean 70.261992 C, last change 6.405e-02
```

The center of the plate is still at 11.6 degrees after 1000 sweeps, and the last sweep still changed some point by 0.064: Jacobi has barely begun. By Equation (15.3), reaching a change of 10^{-6} needs about 24,000 sweeps at this size. Red-black SOR gets there in 316:

```
grid 100 x 100, red-black SOR, omega 1.939676: 316 sweeps
center 122.789173 C, mean 122.499995 C
```

The mean, 122.5 degrees, is the average of the four edge temperatures, as the symmetry of the problem requires; the center, 122.8, is close to it.

With `--time` and the fine 400×400 grid, the programs report their speed. On the two-core test machine (an Intel Xeon at 2.1 GHz, Python 3.13 with NumPy 2.5.3, Java 21), the Python programs printed:

```
grid 400 x 400, 200 Jacobi sweeps, 1 thread(s)
center 0.000000 C, mean 9.008161 C, last change 3.172e-01
time 2.050 s, 15.6 million updates/s
grid 400 x 400, 2000 Jacobi sweeps, 1 thread(s)
center 0.000000 C, mean 28.428053 C, last change 3.168e-02
time 2.404 s, 133.1 million updates/s
grid 400 x 400, 2000 Jacobi sweeps, 2 thread(s)
center 0.000000 C, mean 28.428053 C, last change 3.168e-02
time 1.766 s, 181.2 million updates/s
```

and the Java programs:

```
grid 400 x 400, 2000 Jacobi sweeps, 1 thread(s)
center 0.000000 C, mean 28.428053 C, last change 3.168e-02
time 0.236 s, 1356.4 million updates/s
grid 400 x 400, 2000 Jacobi sweeps, 2 thread(s)
center 0.000000 C, mean 28.428053 C, last change 3.168e-02
time 0.247 s, 1295.8 million updates/s
grid 400 x 400, red-black SOR, omega 1.984453: 1171 sweeps
center 122.572805 C, mean 122.499983 C
time 0.362 s
grid 400 x 400, red-black SOR, omega 1.984453: 1171 sweeps
center 122.572805 C, mean 122.499983 C
time 0.305 s
```

The numbers show every idea of Section 15.3. Plain Python makes 15.6 million updates per second; NumPy, 133 million on one thread and 181 million on two; Java, 1.36 billion on one thread and no more on two, because at that rate the loop is limited by memory (Equation (15.1)), not by arithmetic. NumPy, running well below the memory bound, still has room to gain from a second core; Java, already at the bound, does not. Red-black SOR in Java converges in about a third of a second, with a small gain from the second thread. On a machine with more cores and more memory channels the parallel gains are larger; the ranking of the methods is not.

Figure 15.1 combines these measurements into the time to *solve* the fine griddle. The Jacobi bars multiply the measured time per sweep by the 340,000 sweeps of Section 15.3.4 (Jacobi was not run to the end; an hour of plain Python proves nothing new), and the SOR and direct bars are measured runs. From the slowest to the fastest there is a factor of more than ten thousand, and the largest single factor, several hundred, comes from the algorithm, not from the language or the cores. **Figure 15.2** shows the answer the blacksmith asked for.

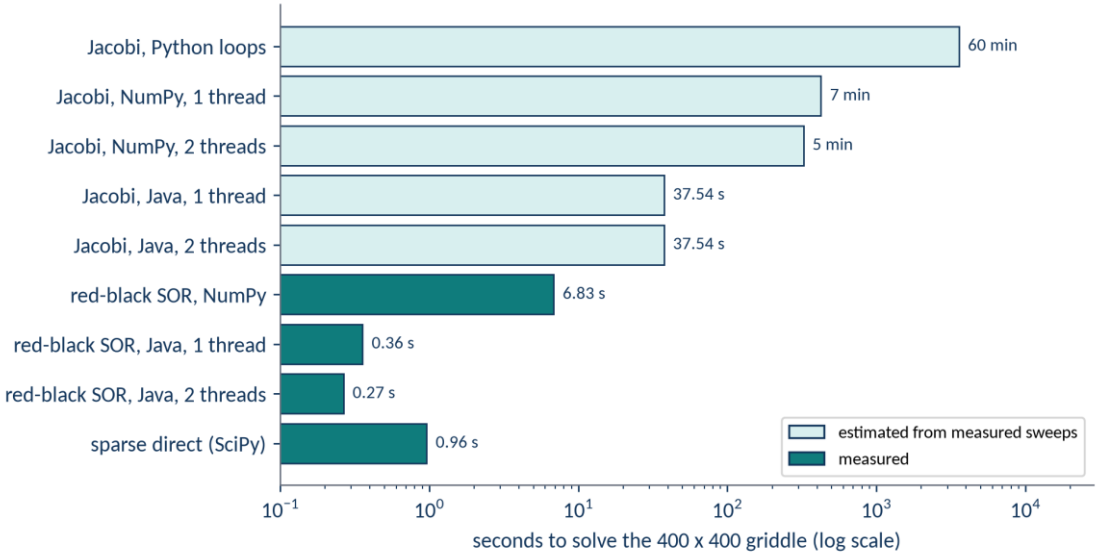


Figure 15.1 Time to solve the 400 × 400 griddle on the two-core test machine: Jacobi (estimated from the measured time per sweep and the number of sweeps needed), red-black SOR and SciPy's sparse direct solver (measured). The scale is logarithmic.

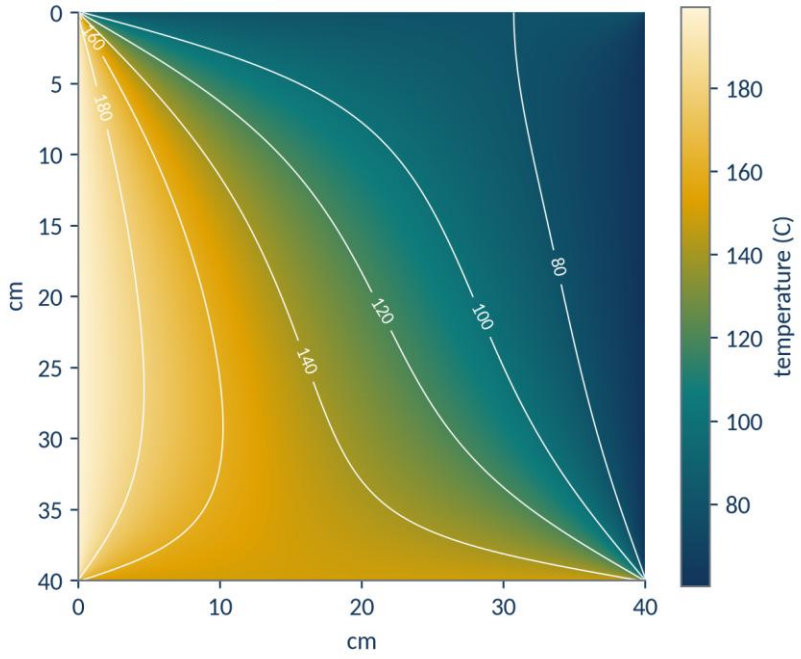


Figure 15.2 Steady temperature of the 40 cm griddle on a 1 mm grid, with the left edge at 200, the right at 60, the top at 80 and the bottom at 150 degrees Celsius; white lines are isotherms.

15.6 Comparing with NumPy and SciPy

Listing 15.8 checks the sum with Python's `math.fsum`, which returns the correctly rounded sum of the exact values [13], and with NumPy's pairwise `sum`, and solves the griddle as the sparse linear system of Chapter 5, $A\mathbf{u} = \mathbf{b}$, with SciPy's sparse direct solver [14]. The matrix is built with `kronsum`, which forms the 2-D five-point operator from the 1-D second-difference matrix tridiag$(-1,2,-1)$.

Listing 15.8 `check_numpy.py`: exact and pairwise sums, and the griddle by a sparse direct solver.

```
"""Library answers for Chapter 15: exact and pairwise sums, and the griddle
solved directly by a sparse factorization.   Usage: python check_numpy.py [n]"""
import math
import sys
import time
import numpy as np
import scipy.sparse as sp
from scipy.sparse.linalg import spsolve
from griddle_loops import LEFT, RIGHT, TOP, BOTTOM

xs = [0.1] * 10_000_000
print(f"math.fsum (correctly rounded): {math.fsum(xs):.10f}")
print(f"numpy.sum (pairwise)         : {np.sum(np.array(xs)):.10f}")

n = int(sys.argv[1]) if len(sys.argv) > 1 else 400
t0 = time.perf_counter()
T = sp.diags([-1.0, 2.0, -1.0], [-1, 0, 1], shape=(n, n))
A = sp.kronsum(T, T, format="csc")        # 4 on the diagonal, -1 for neighbors
b = np.zeros((n, n))
b[:, 0] += LEFT
b[:, -1] += RIGHT
b[0, :] += TOP
b[-1, :] += BOTTOM
u = spsolve(A, b.ravel()).reshape(n, n)
t = time.perf_counter() - t0
c = n // 2 - 1                            # inner point (c+1, c+1) of the grid
print(f"sparse direct solve, {n * n} unknowns: center {u[c, c]:.6f} C,"
      f" mean {u.mean():.6f} C, {t:.2f} s")
```

```
math.fsum (correctly rounded): 1000000.0000000000
numpy.sum (pairwise)         : 1000000.0000000000
sparse direct solve, 160000 unknowns: center 122.572847 C, mean 122.500000 C, 0.87 s
```

Both library sums agree with our pairwise and Kahan results. The direct solve, in 0.87 seconds, gives a center temperature of 122.572847; red-black SOR on the same grid stopped at 122.572805:

```
grid 400 x 400, red-black SOR, omega 1.984453: 1171 sweeps
center 122.572805 C, mean 122.499983 C
```

The two differ by 4×10^{-5} although SOR stopped when no point moved by 10^{-6}. That is the warning of Chapter 5: a small change per sweep is not a small error when the iteration converges slowly, because the remaining error is roughly the change divided by $1 - \rho$. For the griddle 4×10^{-5} degrees is irrelevant; for a problem that needs the digits, tighten the tolerance or estimate the error from ρ.

15.7 Making It Better

- **Measure before optimizing.** Profilers (`cProfile` in Python, Java Flight Recorder in the JDK) show where the time really goes; it is rarely where one guesses. Time a run several times and report the best or the median; the first Java run includes compilation, and other programs on the machine disturb every run.
- **Use processes or free-threaded Python for CPU-bound Python code.** Python's threads run pure-Python code one at a time because of the global interpreter lock; they help only when the work happens inside NumPy or waits for input and output. For pure-Python work, `concurrent.futures.ProcessPoolExecutor` runs separate processes, and Python 3.14's officially supported *free-threaded* build removes the lock, at a cost to single-thread speed of about 1% on macOS aarch64 to 8% on x86-64 Linux, according to the Python documentation [15].

- **Move to a better algorithm before a bigger machine.** For elliptic problems like the griddle, SOR, conjugate gradients with a preconditioner, multigrid [8] or a sparse direct solver [9] each beat Jacobi by orders of magnitude. Parallelize the good algorithm, not the bad one.
- **Keep results reproducible.** Fix the chunking of every reduction by the data, not by the thread count; seed every random-number stream from the task, not from the thread (Chapter 16 does this for its Monte Carlo runs); and keep a single-threaded reference run to compare with.
- **Consider the graphics processor when the data are large and regular.** A GPU has thousands of simple cores and very high memory bandwidth, and stencil sweeps like Jacobi's run on it with little change; array libraries with a NumPy-like interface exist for GPUs. The ideas of this chapter, the roofline, Amdahl's law and reproducible reductions, apply there unchanged.

15.8 Using It for Real

Most readers will not solve heat equations for blacksmiths, but everyone who computes eventually waits for a program. The same checklist applies to a spreadsheet macro that recalculates a cooperative's loan schedules, a script that simulates a thousand seasons of an orchard, or the training of a model.

- **Estimate before you run.** Count the unknowns, the operations per unknown and the number of iterations, as Section 15.3.4 did. If the estimate says hours, no amount of tuning will make it seconds; look for a better method first.
- **Measure the actual time**, with `--time`-style lines or a profiler, on a problem small enough to finish and large enough to be realistic, and extrapolate with the estimate.
- **Vectorize the inner loop** in Python with NumPy, or move it to a compiled language. Keep the arithmetic in the same order if you want identical results.
- **Parallelize over independent tasks first.** The blacksmith's twenty griddle designs are twenty separate problems: they share nothing, need no synchronization and divide perfectly among cores, which is the easiest parallelism there is. On the two-core test machine, twenty red-black SOR solves in Java take about $20 \times 0.35 = 7$ seconds on one core and half that on two; in NumPy about $20 \times 6.8 = 136$ seconds on one core, and roughly half with two processes of `ProcessPoolExecutor`. Parallelizing *within* one solve, as in Section 15.4.3, is harder and gains less.
- **Keep a reference answer.** Run a small case single-threaded and keep its output; after every optimization, the fast version must print the same digits (or differ only in the last places, for a known reason such as a different summation order).

For a cooperative (*koperasi simpan pinjam*) that recomputes the instalment schedules of five thousand members every month, the same reasoning gives a clear plan. Each member's schedule is independent, so the members can be divided among cores with no interaction. Each schedule is a short loop of multiplications and additions, which is cheap in a compiled language and in vectorized NumPy. And the monthly totals that go into the books must be summed reproducibly, in a fixed order or with Kahan's method, so that the report printed on the treasurer's laptop matches the one printed on the office computer to the last rupiah.

15.9 Numerical Methods in Practice: Profitina

Profitina's AI Visibility platform is a pipeline of measurements over websites, and the ideas of this chapter decide how fast it can answer [16]. Scanning a site is mostly *waiting*, for pages to arrive over the network, so many scans can be in progress at once even on few cores; that is

concurrency limited by input and output, not by arithmetic, and the analysis of Equation (15.1) is replaced by the question of how many requests a site can fairly be sent at once. Scoring the downloaded pages and running the local Qwen2.5 model are *computation*, limited by processor and memory, where Amdahl's law and the roofline apply: the model's speed on the server is set largely by how fast its weights can be read from memory, which is why a very small model (qwen3:1.7b) serves as a fast lane for short classification tasks.

The platform's release gate is a parallel workload of another kind: the full automated suite, well over a thousand Python and JavaScript tests, must pass before a release [16]. Tests are independent of each other, like the Monte Carlo seasons of Chapter 16, so they can be spread over cores with almost perfect speedup, provided no two tests share state, the software counterpart of the race condition of Section 15.3.3. And the weekly score trends of the historical dashboard are sums over growing tables, which must come out the same whatever the order of the rows: the reproducible summation of Section 15.3.5.

15.10 Common Mistakes, and How to Avoid Them

- **Optimizing the wrong thing.** Making Jacobi's sweep ten times faster still leaves it hundreds of times slower than SOR. Count sweeps (or iterations) before counting nanoseconds.
- **Timing once.** A single run includes start-up, compilation and whatever else the machine was doing. Repeat, take the best or the median, and report the machine.
- **Expecting p cores to give p times the speed.** Amdahl's law Equation (15.2), synchronization and memory bandwidth (Equation (15.1)) all take their share; on memory-bound work, two cores may be no faster than one.
- **Parallelizing an in-place update.** Gauss–Seidel or SOR split naively over threads has a race condition and gives timing-dependent results. Use Jacobi, red-black ordering, or give each thread its own output.
- **Letting the thread count decide a sum.** Chunked sums change with the number of chunks. Fix the chunks by the data and use pairwise or Kahan summation within them.
- **Using Python threads for pure-Python arithmetic.** With the global interpreter lock they take turns; use NumPy, processes, or the free-threaded build.
- **"Simplifying" Kahan's algorithm.** $(t - s) - y$ is not zero in floating point; it is the lost part. Keep the formula as written and do not compile C versions with flags that allow reassociation.

15.11 Chapter Summary

- Time is spent on arithmetic, on moving data and on bookkeeping; Equation (15.1) bounds the time by operations over the arithmetic rate and bytes over the bandwidth. Jacobi, at 0.25 operations per byte, is memory-bound.
- Vectorizing with NumPy removed the interpreter from the inner loop (15.6 to 133 million updates per second), and compiled Java reached 1.36 billion, the memory bound of the test machine; all three gave identical digits because they perform the same operations in the same order.
- Amdahl's law Equation (15.2) limits parallel speedup by the serial fraction; synchronization, overhead and memory bandwidth reduce it further. Jacobi divides safely into strips; Gauss–Seidel needs red-black ordering.

- Jacobi needs about $2(n+1)^2\ln(1/\varepsilon)/\pi^2$ sweeps (Equation (15.3)) and SOR about $(n+1)\ln(1/\varepsilon)/(2\pi)$ (Equation (15.4)); for the 400×400 griddle that is 340,000 against 1171 sweeps, and the better algorithm beat the faster language and the extra core by far.
- Floating-point sums depend on the order; naive summation errs by up to $(n-1)u\sum|x_i|$ (Equation (15.5)), pairwise by $\lceil\log_2 n\rceil u\sum|x_i|$ (Equation (15.6)), and Kahan's method (**Pseudocode 15.1**) by about $2u\sum|x_i|$. Chunks fixed by the data make parallel sums reproducible.

15.12 Exercises

1. A program spends 90% of its time in a loop that parallelizes perfectly. What speedup does Amdahl's law predict on 4, 16 and infinitely many cores? What fraction must be parallel for 16 cores to give a speedup of 12?
2. For the red-black SOR update $u \leftarrow u + \omega(1/4\,(u_N + u_S + u_W + u_E) - u)$, count the operations and the bytes read and written per point in the best case, and compute the arithmetic intensity. Is red-black SOR memory-bound on the machine of Section 15.3.1?
3. Explain with a 3×3 grid of inner points why splitting Gauss–Seidel into two strips of rows can give different results depending on which thread runs first, and why red-black ordering cannot.
4. By hand, add $x = (1, 10^{-16}, 10^{-16}, \ldots, 10^{-16})$ with ten copies of 10^{-16}, first left to right and then with Kahan's method, in double precision. What is the exact sum and which method gets it?
5. `griddle_rb.py` computes the SOR value for all points in each half-sweep and keeps half. Estimate how much time this wastes, and describe how strided slices such as `u[1:-1:2, 1:-1:2]` could update only one color.
6. Modify `chunked` into a function `reproducible_sum(xs, threads)` that always gives the same answer for any number of threads. Explain your design.

15.13 Solutions to the Exercises

1. With $f = 0.9$: $S(4) = 1/(0.1 + 0.225) = 3.08$; $S(16) = 1/(0.1 + 0.05625) = 6.40$; $S(\infty) = 1/0.1 = 10$. For $S(16) = 12$ we need $(1-f) + f/16 = 1/12$, that is $1 - \frac{15}{16}f = 0.08333$, so $f = 0.9778$: more than 97.8% of the time must be parallel. Twelve-fold speedups on sixteen cores are rare for this reason.

2. Per point: the mean of four neighbors takes 3 additions and 1 multiplication, then one subtraction, one multiplication by ω and one addition, so 7 operations (plus the comparison for the change). In the best case each point of the grid is read once and written once per half-sweep that updates it, and each half-sweep reads the whole grid: about 16 bytes read and 8 written per updated point, roughly 24 bytes. The intensity is $7/24 \approx 0.3$ operations per byte, below the $F/B \approx 0.5$ of one core: red-black SOR is also memory-bound, which is why its second thread gained little. Its advantage is not a faster sweep but 290 times fewer of them.

3. Number the inner points row by row, 1, … ,9, and give rows 1–2 to thread A and row 3 to thread B. Gauss–Seidel updates point 7 (row 3) using point 4 (row 2). If B runs before A has updated point 4, it uses the old value; if after, the new one. The two orders give different values at point 7, and everything computed from it differs too: the result depends on timing. In red-black ordering every point's neighbors have the other color, and during the red half-sweep only red points change; each red point reads only black values, which are fixed for the whole half-sweep. The order of the red updates, and hence the split into threads, cannot matter.

4. Left to right: $1 + 10^{-16}$ rounds to 1, because 10^{-16} is less than half the spacing 2.22×10^{-16} of doubles just above 1; the same happens ten times, and the result is exactly 1. Kahan: at the first small term $y = 10^{-16}$, $t = 1$, and $c = (1 - 1) - 10^{-16} = -10^{-16}$, the whole term remembered as lost. At the second, $y = 10^{-16} - (-10^{-16}) = 2 \times 10^{-16}$, and now $t = 1 + 2 \times 10^{-16}$ rounds up to the next double, $1 + 2.22 \times 10^{-16}$, so $c = 2.22 \times 10^{-16} - 2 \times 10^{-16} = 2.2 \times 10^{-17}$: this time a little too much was added, and the next term is reduced accordingly. Continuing, the sum steps up by one spacing about every second term and ends at $1 + 5 \times 2.22 \times 10^{-16} = 1 + 1.11 \times 10^{-15}$, printed as 1.000000000000001, the double nearest to the exact sum $1 + 10^{-15}$. `math.fsum` returns the same value, while the naive sum returns 1.

5. Each half-sweep computes the SOR expression for all n^2 points instead of the $n^2/2$ of one color, so the arithmetic is doubled; since the computation is memory-bound, the time grows by less than a factor of 2 but noticeably (the masked assignment also costs a pass). The red points with $i + j$ even form two regular sublattices: i odd and j odd, and i even and j even. With `r = u[1:-1, 1:-1]`, the first sublattice is `r[0::2, 0::2]` and its north neighbors are `u[0:-2:2, 1:-1:2]`, its south neighbors `u[2::2, 1:-1:2]`, and so on; updating the two sublattices with such strided views touches each red point once. The same indexing works for black. This is how production codes vectorize red-black methods.

6. Choose the chunk boundaries from the data length alone, for example $K = 64$ fixed chunks with boundaries $\lfloor kn/K \rfloor$. Each chunk is summed with Kahan's method (or pairwise), giving 64 partial sums; the threads may compute them in any order and in any number, because each partial sum depends only on its own chunk. Finally the 64 partial sums are added in chunk order, pairwise. The answer is then a function of the data only, identical on 1 thread or 64, and it is accurate to a few units in the last place.

References

[1] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[2] C. R. Harris *et al.*, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, 2020.

[3] OpenJDK, "JEP 508: Vector API (Tenth Incubator)," JDK 25, 2025; and "JEP 529: Vector API (Eleventh Incubator)," JDK 26, 2026. [Online]. Available: https://openjdk.org/jeps/508

[4] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. AFIPS Spring Joint Computer Conference*, 1967, pp. 483–485.

[5] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[6] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: SIAM, 2003.

[7] S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers*, 8th ed. New York, NY, USA: McGraw-Hill Education, 2021.

[8] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial*, 2nd ed. Philadelphia, PA, USA: SIAM, 2000.

[9] A. George, "Nested dissection of a regular finite element mesh," *SIAM Journal on Numerical Analysis*, vol. 10, no. 2, pp. 345–363, 1973.

[10] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: SIAM, 2002.

[11] Python Software Foundation, "Built-in Functions: sum," Python 3.14 documentation, 2026. [Online]. Available: https://docs.python.org/3/library/functions.html#sum

[12] W. Kahan, “Further remarks on reducing truncation errors,” *Communications of the ACM*, vol. 8, no. 1, p. 40, 1965.

[13] Python Software Foundation, “math — Mathematical functions: math.fsum,” Python 3.14 documentation, 2026. [Online]. Available: https://docs.python.org/3/library/math.html#math.fsum

[14] The SciPy community, “scipy.sparse.linalg.spsolve,” SciPy 1.18.1 documentation, 2026. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.spsolve.html

[15] Python Software Foundation, “Python support for free threading,” Python 3.14 documentation, 2026. [Online]. Available: https://docs.python.org/3/howto/free-threading-python.html

[16] Profitina, “Profitina — AI Visibility (GEO) platform,” 2026. [Online]. Available: https://profitina.com

Chapter 16

Capstone: Simulating a Real System End to End

Every chapter of this book solved one kind of problem: a root, a linear system, an interpolant, a fit, a derivative, an integral, a differential equation, an eigenvector, an optimum, a trained model, a fast sum. Real questions rarely come in one piece. This chapter takes a single real question, how much water storage a family orchard needs to get its trees through the dry season, and answers it by building a simulation in which almost every method of the book has a job. By the end of the chapter you will be able to break a real system into parts that each match a numerical method; build each part and check it against a second, independent method before trusting it; assemble the parts into a day-by-day simulation; run hundreds of possible seasons in parallel with reproducible random numbers; turn the results into a decision with an optimizer, and understand the decision through a simple formula; check the whole against SciPy; and say honestly what the simulation can and cannot tell the person who asked.

16.1 Pak Hadi’s Dry Season

Pak Hadi’s family keeps a fruit and durian orchard on a hillside east of Jember, in East Java. The oldest part of the orchard is a block of sixty durian trees, planted by his father and now in full bearing, with rambutan, mangosteen and avocado on the terraces below. Durian is the orchard’s income, and its fruit depends on the trees getting through the dry season from June to October without drought stress, especially in August and September, when the flowers and young fruit are forming.

Water comes from three places. A well at the bottom of the slope has an electric pump that lifts water into a 13,000-liter tank on a stand above the trees. The roof of the storage shed, 120 square meters of corrugated metal, drains into the same tank when it rains. From the tank, a drip line runs by gravity to every tree; each morning at six Pak Hadi opens the valve and closes it when the day’s ration has gone out, or at half past eight at the latest, when he has to leave for the market.

Last year the system failed in September. The well, as every year, gave less water as the dry season went on; the trees, as every year, needed more. For three weeks the tank was nearly empty every morning, the pressure in the drip line was low, and the trees got perhaps two thirds of their ration. The harvest in January was smaller. A neighbor suggested a second tank. A dealer in town offered plastic tanks at about Rp 1,500 per liter of capacity. But how many liters? Five thousand more? Twenty thousand? Would it have helped at all, if the well was the real problem?

No formula answers this. The answer depends on the rain, which is random; on the well, which slows down through the season; on the trees’ needs, which grow; on the tank’s shape, which sets the pressure; and on the pump, whose flow depends on how high it must lift. Each of these is a small numerical problem that an earlier chapter knows how to solve. Put together, day by day, and repeated over hundreds of possible dry seasons, they answer Pak Hadi’s question. **Figure 16.1** shows the plan.

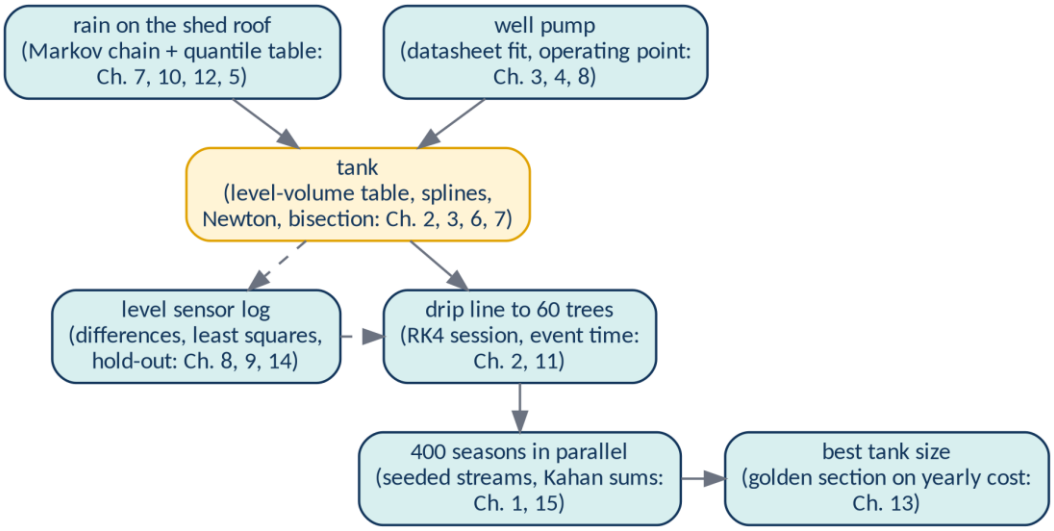


Figure 16.1 The orchard's water system as a simulation: every part is computed by methods of earlier chapters, the parts meet in the tank, and many simulated seasons feed the choice of tank size.

16.2 The Idea in Plain Words

A *simulation* imitates a system step by step in time, using a model of each part and of how the parts interact. Ours advances one day at a time. Each day it decides the weather, adds the rain from the roof to the tank, runs the morning drip session and records whether the trees got their full ration, and runs the pump in the evening to refill the tank as far as the well allows. After 153 days, June 1 to October 31, it reports how much water the trees missed and how much electricity the pump used.

One simulated season is one possible future, drawn with random weather. Pak Hadi's decision must be good across the futures he may meet, so the simulation is repeated for hundreds of seasons, each with its own random weather; this is the *Monte Carlo method* [1]. The averages over seasons estimate what to expect, and the spread tells how much luck is involved.

The discipline that makes a simulation trustworthy is to build it from *checked parts*. Before any season is simulated, every part is computed two ways, by the method the simulation will use and by an independent one, and the two must agree. A tank level from a spline must agree with Newton's method on the same spline; a stationary weather distribution from the power method must agree with Gauss–Seidel; a drip session integrated with a large step must agree with the same session integrated with a small one. Only then are the parts assembled. This is the verification that textbooks of numerical methods ask for every single method [2], [3], applied to a whole system. **Table 16.1** lists where each chapter's methods work.

Table 16.1 Where each chapter's methods work in the orchard simulation.

Part of the system	Question	Methods (chapters)
Tank	How much water at a given level, and what level for a given volume?	cubic spline, tridiagonal LU, Newton, bisection (7, 6, 3, 2)
Pump	How much does it deliver when lifting to the tank?	least squares, Gaussian elimination, Newton (8, 4, 3)
Weather	How often does it rain, and how much?	Markov chain, power method, Gauss–Seidel, interpolation, trapezoid rule (12, 5, 7, 10)
Storm record	How much rain did one storm bring?	trapezoid and Simpson rules (10)
Drip line	How long does the ration take at the tank's pressure?	Runge–Kutta, regula falsi for the end time (11, 2)
Level sensor	Does the drip model match the orchard?	central differences, least squares, held-back data (9, 8, 14)
Seasons	What happens over many dry seasons?	seeded random streams, parallel workers, compensated sums (1, 15)
Decision	Which tank size costs least per year?	golden-section search (13)

Let us do one day by hand to see the balance. On September 15, day 106 of the season, each tree needs 42 liters, so the ration is $60 \times 42 = 2520$ liters, while the well can give only 1600 liters that evening. If it does not rain, the tank ends the day 920 liters lower than it began. Over the whole season, adding up every day on which the trees need more than the well gives, the shortfall is 45,665 liters; on the other days the well could give 50,595 liters more than needed, but a full tank cannot hold any of that surplus. So the tank starts the dry season full and must carry the orchard through a shortfall of about 45,000 liters, minus whatever the rain adds once the tank has room for it. A 13,000-liter tank clearly cannot; the simulation will tell how large one must be, and whether the extra liters pay for themselves.

> **KEY INSIGHT:** A trustworthy simulation is built, not written. Model each part with the method that fits it, check each part against an independent method, assemble, then check the whole against simple hand calculations (the season's shortfall, the expected rain) before believing any answer it gives.

16.3 The Model, Part by Part

16.3.1 The tank: a table and two splines

The tank stands on a two-meter stand and is slightly wider at the top than at the bottom. Pak Hadi calibrated it once with a 20-liter bucket, writing down the level in centimeters every 25 cm of height: 0, 1190, 2500, 3940, 5500, 7190, 9000, 10940 and 13000 liters at levels 0, 25, …, 200 cm. The simulation needs the table both ways. The drip line's pressure depends on the *level* h of the water, while the water balance is kept in *volume* V. A natural cubic spline through the points (h_i, V_i) (Chapter 7) gives $V(h)$; its second derivatives come from a tridiagonal linear system solved by the Thomas algorithm, the LU factorization of Chapter 6 for tridiagonal matrices.

For $h(V)$ there are two ways. One is to solve $V(h) = V$ for h with Newton's method (Chapter 3), using the spline's derivative, or with bisection (Chapter 2). The other is *inverse interpolation*: fit a second spline through the swapped points (V_i, h_i). The simulation evaluates $h(V)$ thousands of times per season, so it uses the inverse spline, which costs one evaluation instead of several Newton steps. The two are different interpolants of the same data, so they need not agree exactly;

the component check in Section 16.5 shows that they agree to 0.03 mm at 6500 liters, far below the precision of a bucket.

Larger capacities are modeled as several tanks of the same shape side by side, filled and drained together, so a tank of capacity C has the level

$$h_C(V) = h\left(\frac{V}{C/13000}\right), \tag{16.1}$$

the same height of two meters with a proportionally larger area.

16.3.2 The pump: a fitted curve and an operating point

The pump's datasheet gives the *head* it produces, the height in meters to which it can push water, at eight flows from 0 to 70 liters per minute. A quadratic

$$H_p(Q) = a + bQ + cQ^2 \tag{16.2}$$

is fitted by least squares (Chapter 8): the three normal equations are solved by Gaussian elimination with partial pivoting (Chapter 4). The pipe from the well to the tank demands a head that grows with the flow: a static lift of 12 m from the water in the well to the top of the tank, plus friction losses that grow with the square of the flow, $H_s(Q) = 12 + 0.0011\,Q^2$. The pump settles at the *operating point* where supply meets demand,

$$g(Q) = H_p(Q) - H_s(Q) = 0, \tag{16.3}$$

a root found by Newton's method. By hand from $Q_0 = 50$: $g(50) = 31.9958 - 0.023155 \times 50 - 0.0041131 \times 2500 - (12 + 2.75) = 5.8054$ and $g'(50) = -0.023155 - 2 \times 0.0041131 \times 50 - 2 \times 0.0011 \times 50 = -0.5445$, so $Q_1 = 50 + 5.8054/0.5445 = 60.66$ liters per minute; two more steps settle at 59.752. At that flow, 1000 liters take $1000/59.752 = 16.74$ minutes, and a pump drawing 0.8 kW uses

$$E = \frac{1000}{60\,Q^*}\,P = \frac{1000}{60 \times 59.752} \times 0.8 = 0.2231 \text{ kWh per 1000 liters}, \tag{16.4}$$

which at the household tariff of Rp 1,445 per kWh for July to September 2026 [4] costs Rp 322 per 1000 liters.

16.3.3 Weather: a Markov chain and a quantile table

The orchard's rain gauge notebook, kept for many dry seasons, shows that days are not independent: a rainy day is more likely to be followed by another. The simulation classifies days into three states following BMKG's daily rain categories [5]: *dry* (below 0.5 mm), *light* (0.5 to 20 mm) and *heavy* (20 mm and more, which lumps BMKG's moderate, heavy and very heavy classes). Tomorrow's state depends only on today's, through a transition matrix P estimated from the notebook,

$$P = \begin{pmatrix} 0.88 & 0.10 & 0.02 \\ 0.60 & 0.32 & 0.08 \\ 0.50 & 0.35 & 0.15 \end{pmatrix}, \tag{16.5}$$

where row i gives the probabilities of tomorrow's states after a day in state i. This is a *Markov chain*, the same kind of object as the web-surfer chain of PageRank in Chapter 12. Its long-run

share of days in each state is the *stationary distribution* $\boldsymbol{\pi}$, the left eigenvector of P for the eigenvalue 1:

$$\boldsymbol{\pi} P = \boldsymbol{\pi}, \qquad \pi_1 + \pi_2 + \pi_3 = 1. \tag{16.6}$$

The power method finds it by repeating $\boldsymbol{\pi} \leftarrow \boldsymbol{\pi} P$; the error shrinks each step by the second-largest eigenvalue in size, 0.305 here, so 40 steps give full double precision. As a check, Equation (16.6) is also the linear system $(I - P^{\mathsf{T}})\boldsymbol{\pi} = \mathbf{0}$, which Gauss–Seidel (Chapter 5) solves up to scale, and the scale is fixed afterward by the sum. Both give $\boldsymbol{\pi} = (0.8288, 0.1386, 0.0326)$: in the long run 17% of dry-season days bring rain.

How much rain? On a light day, the depth is drawn from a *quantile table* of the notebook: the smallest light day had 0.5 mm, a quarter of them had less than 2 mm, half less than 5, three quarters less than 10, and the largest 20. A uniform random number u in $[0,1)$ is turned into a depth by linear interpolation in this table (Chapter 7), which reproduces the notebook's distribution. The mean depth of such a day is the integral of the quantile function over u from 0 to 1, and because the function is piecewise linear the trapezoid rule of Chapter 10 gives it exactly: $0.25 \times (0.5/2 + 2 + 5 + 10 + 20/2) = 6.8125$ mm. For heavy days it is 41.375 mm. So the expected rain is $0.1386 \times 6.8125 + 0.0326 \times 41.375 = 2.29$ mm a day, 351 mm a season, and with a runoff coefficient of 0.85 the 120 m^2 roof delivers 102 liters per millimeter, about 35,800 liters a season. That hand estimate will be compared with the simulation.

16.3.4 The trees and the well through the season

Pak Hadi's own watering practice gives the ration per tree in the middle of each month: 25 liters a day in June, 32 in July, 38 in August, 42 in September and 35 in October. What the well can give in a day falls from 2800 liters in June to 2400, 1900, 1600 and then, after the first rains in October, 1800. Between the middle days the simulation interpolates linearly, a choice that is simple, never overshoots and is as accurate as the numbers themselves. After a day with 10 mm of rain or more the soil is wet and the trees are not watered.

16.3.5 The drip line: a small differential equation

The water runs to the trees by gravity, so the flow depends on the height of the water surface above the drip line: the level h in the tank plus the height $Z = 1.5$ m of the stand. For drippers and valves the flow grows with the square root of the pressure head, as for the tank of Chapter 11 draining through an opening, so while the valve is open

$$\frac{dV}{dt} = -K\sqrt{\frac{h(V)}{100} + Z}, \qquad K = 12 \text{ L/min per } \sqrt{\text{m}}, \tag{16.7}$$

with h in centimeters and t in minutes. With a full tank the flow is $12\sqrt{3.5} = 22.4$ liters per minute; with 30 cm of water left, $12\sqrt{1.8} = 16.1$. The morning ration of up to 2520 liters therefore takes between about 115 and 150 minutes, and the window from six to half past eight is exactly 150: when the tank is low, the trees get less than their ration *even if the tank still holds enough water*, because the pressure is too low to deliver it in time. That effect is why the problem needs a differential equation and not just a subtraction.

The simulation integrates Equation (16.7) with the classical Runge–Kutta method (Chapter 11) in steps of 10 minutes. The session ends when the delivered volume reaches the ration, at a time

inside some step. That time is the root of $V(\tau) - V_{\text{target}} = 0$ for the step length τ, where $V(\tau)$ is one Runge–Kutta step of length τ from the step's start; regula falsi (Chapter 2) finds it in a few evaluations because the function is almost linear over ten minutes. The session also ends at the end of the window, whatever has been delivered.

16.3.6 One day, and a season

Let V_d be the water in the tank at dawn of day d, r_d the rain in millimeters, R_d the ration and W_d what the well can give. One day of the simulation is

$$\begin{aligned} V' &= \min(C,\ V_d + 102\,r_d), \\ V'' &= V' - G_d, \qquad G_d = \text{drip session from } V' \text{ for } R_d \ (0 \text{ if } r_d \ge 10), \\ V_{d+1} &= V'' + \min(C - V'',\ W_d), \end{aligned} \tag{16.8}$$

where C is the capacity and G_d the water the trees actually got. The water missed that day is $R_d - G_d$, the overflow is $V_d + 102\,r_d - C$ when positive, and the pump's energy follows from the volume pumped and Equation (16.4). The tank is full on June 1, after the rainy season.

16.3.7 Many seasons, reproducibly

Let X_s be a result of season s, for example the liters missed. The Monte Carlo estimate from N seasons and its standard error are

$$\bar{X} = \frac{1}{N}\sum_{s=1}^{N} X_s\,, \qquad \mathrm{SE}(\bar{X}) = \frac{\sigma_X}{\sqrt{N}}, \tag{16.9}$$

where σ_X is the standard deviation of X over seasons. Halving the error needs four times as many seasons, which is why speed (Chapter 15) matters. For a probability p, such as the chance that a season has any shortage, $\sigma = \sqrt{p(1-p)}$, so with $N = 400$ and $p \approx 0.36$ the standard error is $\sqrt{0.36 \times 0.64/400} = 0.024$.

Each season gets its own random stream, the SplitMix64 generator of Chapter 14 seeded with $20260601 + s$. Because a season's randomness depends only on its number, the seasons can be computed in any order by any number of workers, and the results, collected in season order and summed with Kahan's method (Chapter 15), are the same bit for bit on one core or many, and in Java and Python. The same seeding gives *common random numbers*: when two tank sizes are compared, season s has exactly the same weather for both, so the difference between them is due to the tank, not to luck.

16.3.8 The decision and a formula behind it

Pak Hadi's yearly cost of a tank of capacity C is the tank's price spread over its life, the pump's electricity and the value of the fruit lost to missed water:

$$\text{cost}(C) = a\,C + e\,\mathbb{E}[\text{kWh}] + p\,\mathbb{E}[D(C)], \tag{16.10}$$

with $a = 150$ rupiah per liter of capacity per year (Rp 1,500 per liter over ten years), $e = 1445$ rupiah per kWh, $p = 500$ rupiah per liter the trees missed (his own estimate of lost fruit, which we take as given), and $D(C)$ the liters missed in a season. The first term grows with C, the last

falls, and the golden-section search of Chapter 13 finds the minimum, using the same simulated seasons for every C.

There is a way to predict the answer. Adding one liter of capacity costs a a year. It saves one liter of missed water in exactly those seasons in which the tank runs short, and nothing in the others, so it saves p times the probability of a shortage. At the optimum the two must balance:

$$a = p \cdot \Pr[D(C^*) > 0] \quad \Leftrightarrow \quad \Pr[\text{shortage with } C^*] = \frac{a}{p} = \frac{150}{500} = 0.30. \tag{16.11}$$

This is the *newsvendor rule* of operations research, first derived for a newspaper seller deciding how many papers to buy [6]: stock up to the point where the chance of running out equals the ratio of the cost of stocking to the cost of running short. The best tank is the one that runs short in 30% of dry seasons. That sounds like a lot, but a shortage in a season that is only slightly worse than usual costs little; making the tank big enough never to run short would cost more every year than the rare bad seasons lose.

16.4 The Programs

The chapter's programs are in `Chapter16/python` and `Chapter16/java`, with the data in `Chapter16/data`: `tank_table.csv`, `pump_curve.csv`, `storm.csv` (a storm recorded every 10 minutes) and `sensor_log.csv` (the level sensor during one watering, generated by `make_sensor.py`). Every program prints the same text in both languages.

16.4.1 The toolbox

Listing 16.1 collects, in their shortest honest form, the methods of Chapters 2 to 15 that the simulation uses. Each is a few lines, and each is labeled with its chapter; the reader who has worked through the book will recognize every one. Two details are new. `Spline` provides `slope` for Newton's method, and `false_position` is the regula falsi of Chapter 2, used for the end of a drip session.

Listing 16.1 `toolbox.py`: the numerical methods of Chapters 2 to 15 that the orchard simulation reuses.

```
"""The numerical tools of Chapters 2-15 that the orchard simulation reuses,
each in its shortest honest form. Plain Python, no libraries."""
import math

MASK = 0xFFFFFFFFFFFFFFFF


def add(values):
    """Left-to-right sum, the same additions as a Java loop. (Since Python
    3.12 the built-in sum() compensates rounding errors, so it can differ.)"""
    s = 0.0
    for v in values:
        s += v
    return s


class Rng:
    """SplitMix64 (Chapters 14-15): the same numbers in Python and Java."""

    def __init__(self, seed):
        self.s = seed & MASK

    def next64(self):
        self.s = (self.s + 0x9E3779B97F4A7C15) & MASK
        z = self.s
        z = ((z ^ (z >> 30)) * 0xBF58476D1CE4E5B9) & MASK
        z = ((z ^ (z >> 27)) * 0x94D049BB133111EB) & MASK
        return z ^ (z >> 31)

    def uniform(self):
        """A double in [0, 1) from the top 53 bits."""
        return (self.next64() >> 11) * (1.0 / 9007199254740992.0)


def bisection(f, a, b, tol):
    """Chapter 2: halve [a, b] keeping the sign change of f."""
    fa = f(a)
    while b - a > tol:
        m = (a + b) / 2.0
        fm = f(m)
        if (fm < 0.0) == (fa < 0.0):
            a, fa = m, fm
        else:
            b = m
    return (a + b) / 2.0
```

Listing 16.1 (Continued) *toolbox.py: the numerical methods of Chapters 2 to 15 that the orchard simulation reuses.*

```
def false_position(f, a, b, tol, kmax=100):
    """Chapter 2: regula falsi, the secant through the bracket's ends."""
    fa, fb = f(a), f(b)
    for _ in range(kmax):
        c = b - fb * (b - a) / (fb - fa)
        fc = f(c)
        if abs(fc) < tol:
            return c
        if (fc < 0.0) == (fa < 0.0):
            a, fa = c, fc
        else:
            b, fb = c, fc
    return c


def newton(f, df, x, tol, kmax=50):
    """Chapter 3: x <- x - f(x)/f'(x); returns (root, steps)."""
    for k in range(1, kmax + 1):
        dx = f(x) / df(x)
        x -= dx
        if abs(dx) < tol:
            return x, k
    return x, kmax


def gauss_solve(A, b):
    """Chapter 4: Gaussian elimination with partial pivoting."""
    n = len(b)
    M = [row[:] + [b[i]] for i, row in enumerate(A)]
    for k in range(n):
        p = max(range(k, n), key=lambda i: abs(M[i][k]))
        M[k], M[p] = M[p], M[k]
        for i in range(k + 1, n):
            f = M[i][k] / M[k][k]
            for j in range(k, n + 1):
                M[i][j] -= f * M[k][j]
    x = [0.0] * n
    for i in range(n - 1, -1, -1):
        s = M[i][n]
        for j in range(i + 1, n):
            s -= M[i][j] * x[j]
        x[i] = s / M[i][i]
    return x
```

Listing 16.1 (Continued) *toolbox.py: the numerical methods of Chapters 2 to 15 that the orchard simulation reuses.*

```
def gauss_seidel(A, b, x, tol, kmax=1000):
    """Chapter 5: sweep x_i <- (b_i - sum_{j != i} a_ij x_j) / a_ii."""
    n = len(b)
    for k in range(1, kmax + 1):
        change = 0.0
        for i in range(n):
            s = b[i]
            for j in range(n):
                if j != i:
                    s -= A[i][j] * x[j]
            new = s / A[i][i]
            change = max(change, abs(new - x[i]))
            x[i] = new
        if change < tol:
            return x, k
    return x, kmax


def thomas(sub, diag, sup, rhs):
    """Chapter 6: LU for a tridiagonal system (the Thomas algorithm)."""
    n = len(diag)
    c, d = [0.0] * n, [0.0] * n
    c[0], d[0] = sup[0] / diag[0], rhs[0] / diag[0]
    for i in range(1, n):
        m = diag[i] - sub[i] * c[i - 1]
        c[i] = sup[i] / m if i < n - 1 else 0.0
        d[i] = (rhs[i] - sub[i] * d[i - 1]) / m
    x = [0.0] * n
    x[n - 1] = d[n - 1]
    for i in range(n - 2, -1, -1):
        x[i] = d[i] - c[i] * x[i + 1]
    return x


class Spline:
    """Chapter 7: natural cubic spline through (xs, ys), xs increasing."""
```

Listing 16.1 (Continued) *toolbox.py: the numerical methods of Chapters 2 to 15 that the orchard simulation reuses.*

```
    def __init__(self, xs, ys):
        n = len(xs)
        h = [xs[i + 1] - xs[i] for i in range(n - 1)]
        sub, diag, sup, rhs = [0.0] * n, [1.0] * n, [0.0] * n, [0.0] * n
        for i in range(1, n - 1):
            sub[i], diag[i], sup[i] = h[i - 1], 2.0 * (h[i - 1] + h[i]), h[i]
            rhs[i] = 6.0 * ((ys[i + 1] - ys[i]) / h[i]
                            - (ys[i] - ys[i - 1]) / h[i - 1])
        self.xs, self.ys, self.h, self.m = xs, ys, h, thomas(sub, diag, sup, rhs)

    def _piece(self, x):
        i = 0
        while i < len(self.h) - 1 and x > self.xs[i + 1]:
            i += 1
        return i

    def __call__(self, x):
        i = self._piece(x)
        h, m, y = self.h[i], self.m, self.ys
        a, b = self.xs[i + 1] - x, x - self.xs[i]
        return (m[i] * a * a * a + m[i + 1] * b * b * b) / (6.0 * h) \
            + (y[i] / h - m[i] * h / 6.0) * a \
            + (y[i + 1] / h - m[i + 1] * h / 6.0) * b

    def slope(self, x):
        i = self._piece(x)
        h, m, y = self.h[i], self.m, self.ys
        a, b = self.xs[i + 1] - x, x - self.xs[i]
        return (-m[i] * a * a + m[i + 1] * b * b) / (2.0 * h) \
            + (y[i + 1] - y[i]) / h - (m[i + 1] - m[i]) * h / 6.0


def interp(xs, ys, x):
    """Chapter 7: piecewise-linear interpolation, constant outside the table."""
    if x <= xs[0]:
        return ys[0]
    for i in range(1, len(xs)):
        if x <= xs[i]:
            t = (x - xs[i - 1]) / (xs[i] - xs[i - 1])
            return ys[i - 1] + t * (ys[i] - ys[i - 1])
    return ys[-1]
```

Listing 16.1 (Continued) *toolbox.py: the numerical methods of Chapters 2 to 15 that the orchard simulation reuses.*

```
def lstsq(rows, ys):
    """Chapter 8: least squares by the normal equations, solved by Chapter 4."""
    n = len(rows[0])
    A = [[add(r[i] * r[j] for r in rows) for j in range(n)] for i in range(n)]
    b = [add(r[i] * y for r, y in zip(rows, ys)) for i in range(n)]
    return gauss_solve(A, b)


def central_diff(ts, vs):
    """Chapter 9: dv/dt at the inner points by central differences."""
    return [(vs[i + 1] - vs[i - 1]) / (ts[i + 1] - ts[i - 1])
            for i in range(1, len(ts) - 1)]


def trapezoid(ys, h):
    """Chapter 10: composite trapezoid rule on equally spaced samples."""
    return h * (add(ys) - (ys[0] + ys[-1]) / 2.0)


def simpson(ys, h):
    """Chapter 10: composite Simpson rule (an even number of intervals)."""
    s = ys[0] + ys[-1]
    for i in range(1, len(ys) - 1):
        s += (4.0 if i % 2 == 1 else 2.0) * ys[i]
    return h * s / 3.0


def rk4_step(f, t, y, dt):
    """Chapter 11: one classical Runge-Kutta step for y' = f(t, y)."""
    k1 = f(t, y)
    k2 = f(t + dt / 2.0, y + dt / 2.0 * k1)
    k3 = f(t + dt / 2.0, y + dt / 2.0 * k2)
    k4 = f(t + dt, y + dt * k3)
    return y + dt / 6.0 * (k1 + 2.0 * k2 + 2.0 * k3 + k4)


def power_method(P, iters):
    """Chapter 12: left eigenvector of a transition matrix, pi <- pi P."""
    n = len(P)
    pi = [1.0 / n] * n
    for _ in range(iters):
        pi = [add(pi[i] * P[i][j] for i in range(n)) for j in range(n)]
    return pi
```

Listing 16.1 (Continued) *toolbox.py: the numerical methods of Chapters 2 to 15 that the orchard simulation reuses.*

```
def golden_min(f, a, b, tol):
    """Chapter 13: golden-section search for the minimum of f on [a, b]."""
    r = (math.sqrt(5.0) - 1.0) / 2.0
    x1, x2 = b - r * (b - a), a + r * (b - a)
    f1, f2 = f(x1), f(x2)
    while b - a > tol:
        if f1 < f2:
            b, x2, f2 = x2, x1, f1
            x1 = b - r * (b - a)
            f1 = f(x1)
        else:
            a, x1, f1 = x1, x2, f2
            x2 = a + r * (b - a)
            f2 = f(x2)
    return (a + b) / 2.0


def kahan_sum(xs):
    """Chapter 15: compensated summation."""
    s = c = 0.0
    for x in xs:
        y = x - c
        t = s + y
        c = (t - s) - y
        s = t
    return s
```

Listing 16.2 is the Java toolbox. Functions are passed as `DoubleUnaryOperator`, and a small interface `Ode` stands for the right-hand side $f(t, y)$ of a differential equation. Every loop adds in the same order as in Python, and the pivot search takes the first of equal candidates, as Python's `max` does, so that the two toolboxes give identical results.

Listing 16.2 `Toolbox.java`: the same methods in Java.

```java
// The numerical tools of Chapters 2-15 that the orchard simulation reuses,
// each in its shortest honest form. Plain Java, no libraries.
import java.util.function.DoubleUnaryOperator;

public class Toolbox {
    /** SplitMix64 (Chapters 14-15): the same numbers in Python and Java. */
    static class Rng {
        long s;

        Rng(long seed) { s = seed; }

        long next64() {
            s += 0x9E3779B97F4A7C15L;
            long z = s;
            z = (z ^ (z >>> 30)) * 0xBF58476D1CE4E5B9L;
            z = (z ^ (z >>> 27)) * 0x94D049BB133111EBL;
            return z ^ (z >>> 31);
        }

        /** A double in [0, 1) from the top 53 bits. */
        double uniform() { return (next64() >>> 11) * (1.0 / 9007199254740992.0); }
    }

    /** Chapter 2: halve [a, b] keeping the sign change of f. */
    static double bisection(DoubleUnaryOperator f, double a, double b, double tol) {
        double fa = f.applyAsDouble(a);
        while (b - a > tol) {
            double m = (a + b) / 2.0, fm = f.applyAsDouble(m);
            if ((fm < 0.0) == (fa < 0.0)) { a = m; fa = fm; } else b = m;
        }
        return (a + b) / 2.0;
    }

    /** Chapter 2: regula falsi, the secant through the bracket's ends. */
    static double falsePosition(DoubleUnaryOperator f, double a, double b,
                                double tol) {
        double fa = f.applyAsDouble(a), fb = f.applyAsDouble(b), c = a;
        for (int k = 0; k < 100; k++) {
            c = b - fb * (b - a) / (fb - fa);
            double fc = f.applyAsDouble(c);
            if (Math.abs(fc) < tol) return c;
            if ((fc < 0.0) == (fa < 0.0)) { a = c; fa = fc; }
            else { b = c; fb = fc; }
        }
        return c;
    }
```

Listing 16.2 (Continued) *Toolbox.java: the same methods in Java.*

```
    /** Chapter 3: x <- x - f(x)/f'(x); returns {root, steps}. */
    static double[] newton(DoubleUnaryOperator f, DoubleUnaryOperator df,
                           double x, double tol) {
        for (int k = 1; k <= 50; k++) {
            double dx = f.applyAsDouble(x) / df.applyAsDouble(x);
            x -= dx;
            if (Math.abs(dx) < tol) return new double[] {x, k};
        }
        return new double[] {x, 50};
    }

    /** Chapter 4: Gaussian elimination with partial pivoting. */
    static double[] gaussSolve(double[][] A, double[] b) {
        int n = b.length;
        double[][] M = new double[n][n + 1];
        for (int i = 0; i < n; i++) {
            System.arraycopy(A[i], 0, M[i], 0, n);
            M[i][n] = b[i];
        }
        for (int k = 0; k < n; k++) {
            int p = k;
            for (int i = k + 1; i < n; i++)
                if (Math.abs(M[i][k]) > Math.abs(M[p][k])) p = i;
            double[] t = M[k]; M[k] = M[p]; M[p] = t;
            for (int i = k + 1; i < n; i++) {
                double f = M[i][k] / M[k][k];
                for (int j = k; j <= n; j++) M[i][j] -= f * M[k][j];
            }
        }
        double[] x = new double[n];
        for (int i = n - 1; i >= 0; i--) {
            double s = M[i][n];
            for (int j = i + 1; j < n; j++) s -= M[i][j] * x[j];
            x[i] = s / M[i][i];
        }
        return x;
    }
```

Listing 16.2 (Continued) *Toolbox.java: the same methods in Java.*

```
    /** Chapter 5: sweep x_i <- (b_i - sum_{j != i} a_ij x_j) / a_ii.
     *  Returns the number of sweeps; x is updated in place. */
    static int gaussSeidel(double[][] A, double[] b, double[] x, double tol) {
        int n = b.length;
        for (int k = 1; k <= 1000; k++) {
            double change = 0.0;
            for (int i = 0; i < n; i++) {
                double s = b[i];
                for (int j = 0; j < n; j++) if (j != i) s -= A[i][j] * x[j];
                double nu = s / A[i][i];
                change = Math.max(change, Math.abs(nu - x[i]));
                x[i] = nu;
            }
            if (change < tol) return k;
        }
        return 1000;
    }

    /** Chapter 6: LU for a tridiagonal system (the Thomas algorithm). */
    static double[] thomas(double[] sub, double[] diag, double[] sup,
                           double[] rhs) {
        int n = diag.length;
        double[] c = new double[n], d = new double[n], x = new double[n];
        c[0] = sup[0] / diag[0];
        d[0] = rhs[0] / diag[0];
        for (int i = 1; i < n; i++) {
            double m = diag[i] - sub[i] * c[i - 1];
            c[i] = i < n - 1 ? sup[i] / m : 0.0;
            d[i] = (rhs[i] - sub[i] * d[i - 1]) / m;
        }
        x[n - 1] = d[n - 1];
        for (int i = n - 2; i >= 0; i--) x[i] = d[i] - c[i] * x[i + 1];
        return x;
    }

    /** Chapter 7: natural cubic spline through (xs, ys), xs increasing. */
    static class Spline implements DoubleUnaryOperator {
        final double[] xs, ys, h, m;
```

Listing 16.2 (Continued) *Toolbox.java: the same methods in Java.*

```
        Spline(double[] xs, double[] ys) {
            int n = xs.length;
            this.xs = xs;
            this.ys = ys;
            h = new double[n - 1];
            for (int i = 0; i < n - 1; i++) h[i] = xs[i + 1] - xs[i];
            double[] sub = new double[n], diag = new double[n];
            double[] sup = new double[n], rhs = new double[n];
            java.util.Arrays.fill(diag, 1.0);
            for (int i = 1; i < n - 1; i++) {
                sub[i] = h[i - 1];
                diag[i] = 2.0 * (h[i - 1] + h[i]);
                sup[i] = h[i];
                rhs[i] = 6.0 * ((ys[i + 1] - ys[i]) / h[i]
                                - (ys[i] - ys[i - 1]) / h[i - 1]);
            }
            m = thomas(sub, diag, sup, rhs);
        }

        int piece(double x) {
            int i = 0;
            while (i < h.length - 1 && x > xs[i + 1]) i++;
            return i;
        }

        public double applyAsDouble(double x) {
            int i = piece(x);
            double hi = h[i], a = xs[i + 1] - x, b = x - xs[i];
            return (m[i] * a * a * a + m[i + 1] * b * b * b) / (6.0 * hi)
                    + (ys[i] / hi - m[i] * hi / 6.0) * a
                    + (ys[i + 1] / hi - m[i + 1] * hi / 6.0) * b;
        }

        double slope(double x) {
            int i = piece(x);
            double hi = h[i], a = xs[i + 1] - x, b = x - xs[i];
            return (-m[i] * a * a + m[i + 1] * b * b) / (2.0 * hi)
                    + (ys[i + 1] - ys[i]) / hi - (m[i + 1] - m[i]) * hi / 6.0;
        }
    }
```

Listing 16.2 (Continued) *Toolbox.java: the same methods in Java.*

```
    /** Chapter 7: piecewise-linear interpolation, constant outside the table. */
    static double interp(double[] xs, double[] ys, double x) {
        if (x <= xs[0]) return ys[0];
        for (int i = 1; i < xs.length; i++)
            if (x <= xs[i]) {
                double t = (x - xs[i - 1]) / (xs[i] - xs[i - 1]);
                return ys[i - 1] + t * (ys[i] - ys[i - 1]);
            }
        return ys[ys.length - 1];
    }

    /** Chapter 8: least squares by the normal equations, solved by Chapter 4. */
    static double[] lstsq(double[][] rows, double[] ys) {
        int n = rows[0].length;
        double[][] A = new double[n][n];
        double[] b = new double[n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++)
                for (double[] r : rows) A[i][j] += r[i] * r[j];
            for (int k = 0; k < rows.length; k++) b[i] += rows[k][i] * ys[k];
        }
        return gaussSolve(A, b);
    }

    /** Chapter 10: composite trapezoid rule on equally spaced samples. */
    static double trapezoid(double[] ys, double h) {
        double s = 0.0;
        for (double y : ys) s += y;
        return h * (s - (ys[0] + ys[ys.length - 1]) / 2.0);
    }

    /** Chapter 10: composite Simpson rule (an even number of intervals). */
    static double simpson(double[] ys, double h) {
        double s = ys[0] + ys[ys.length - 1];
        for (int i = 1; i < ys.length - 1; i++)
            s += (i % 2 == 1 ? 4.0 : 2.0) * ys[i];
        return h * s / 3.0;
    }

    interface Ode { double f(double t, double y); }
```

Listing 16.2 (Continued) *Toolbox.java: the same methods in Java.*

```
    /** Chapter 11: one classical Runge-Kutta step for y' = f(t, y). */
    static double rk4Step(Ode f, double t, double y, double dt) {
        double k1 = f.f(t, y);
        double k2 = f.f(t + dt / 2.0, y + dt / 2.0 * k1);
        double k3 = f.f(t + dt / 2.0, y + dt / 2.0 * k2);
        double k4 = f.f(t + dt, y + dt * k3);
        return y + dt / 6.0 * (k1 + 2.0 * k2 + 2.0 * k3 + k4);
    }

    /** Chapter 12: left eigenvector of a transition matrix, pi <- pi P. */
    static double[] powerMethod(double[][] P, int iters) {
        int n = P.length;
        double[] pi = new double[n];
        java.util.Arrays.fill(pi, 1.0 / n);
        for (int it = 0; it < iters; it++) {
            double[] nu = new double[n];
            for (int j = 0; j < n; j++)
                for (int i = 0; i < n; i++) nu[j] += pi[i] * P[i][j];
            pi = nu;
        }
        return pi;
    }

    /** Chapter 13: golden-section search for the minimum of f on [a, b]. */
    static double goldenMin(DoubleUnaryOperator f, double a, double b, double tol) {
        double r = (Math.sqrt(5.0) - 1.0) / 2.0;
        double x1 = b - r * (b - a), x2 = a + r * (b - a);
        double f1 = f.applyAsDouble(x1), f2 = f.applyAsDouble(x2);
        while (b - a > tol) {
            if (f1 < f2) {
                b = x2; x2 = x1; f2 = f1;
                x1 = b - r * (b - a);
                f1 = f.applyAsDouble(x1);
            } else {
                a = x1; x1 = x2; f1 = f2;
                x2 = a + r * (b - a);
                f2 = f.applyAsDouble(x2);
            }
        }
        return (a + b) / 2.0;
    }
```

Listing 16.2 (Continued) *Toolbox.java: the same methods in Java.*

```
    /** Chapter 15: compensated summation. */
    static double kahanSum(double[] xs) {
        double s = 0.0, c = 0.0;
        for (double x : xs) {
            double y = x - c, t = s + y;
            c = (t - s) - y;
            s = t;
        }
        return s;
    }
}
```

16.4.2 The orchard model

Listing 16.3 builds the model of Section 16.3 from the toolbox: the two tank splines and the scaled `Tank` of Equation (16.1); the pump fit Equation (16.2) and its operating point Equation (16.3); the weather chain Equation (16.5) and the rain table; the season's tables; the

drip session Equation (16.7); and `season`, the daily balance Equation (16.8) for 153 days. All constants are named at the top of their section, so that Section 16.8 can change them.

Listing 16.3 `orchard.py`: the orchard's water system as one model.

```
"""Pak Hadi's orchard as one model: rain on the shed roof, a tank, a well pump,
a drip line to 60 trees, and the dry season from June 1 to October 31."""
import math
import os
from toolbox import Rng, Spline, interp, lstsq, newton, false_position, rk4_step

DATA = os.path.join(os.path.dirname(os.path.abspath(__file__)), "..", "data")


def read_csv(name):
    with open(os.path.join(DATA, name), encoding="utf-8") as f:
        next(f)
        return [[float(x) for x in line.split(",")] for line in f if line.strip()]


# --- the tank: a bucket-calibrated table, a spline each way (Chapter 7) ---
TABLE = read_csv("tank_table.csv")
LEVELS, LITRES = [r[0] for r in TABLE], [r[1] for r in TABLE]
VOLUME = Spline(LEVELS, LITRES)          # liters at a level (cm)
LEVEL = Spline(LITRES, LEVELS)           # level (cm) at a volume: inverse table
BASE_CAPACITY = LITRES[-1]               # 13,000 L, 2 m high


class Tank:
    """A battery of tanks of the calibrated shape, capacity in liters."""

    def __init__(self, capacity):
        self.capacity, self.scale = capacity, capacity / BASE_CAPACITY

    def level(self, v):
        return LEVEL(min(max(v / self.scale, 0.0), BASE_CAPACITY))


# --- the pump: datasheet curve by least squares (Chapter 8), operating point
#     where pump head equals the pipe's need, by Newton (Chapter 3) ---
CURVE = read_csv("pump_curve.csv")
PA, PB, PC = lstsq([[1.0, q, q * q] for q, _ in CURVE], [h for _, h in CURVE])
STATIC, FRICTION = 12.0, 0.0011          # m lift, m per (L/min)^2 of pipe loss
PUMP_KW, TARIFF = 0.8, 1445.0            # kW drawn, rupiah per kWh (PLN R-1)


def pump_gap(q):
    return PA + PB * q + PC * q * q - (STATIC + FRICTION * q * q)
```

Listing 16.3 (Continued) *orchard.py: the orchard's water system as one model.*

```
def pump_gap_slope(q):
    return PB + 2.0 * PC * q - 2.0 * FRICTION * q


Q_PUMP, _ = newton(pump_gap, pump_gap_slope, 50.0, 1e-10)     # L/min

# --- weather: a 3-state Markov chain (Chapter 12), rain depth drawn from
#     a quantile table by linear interpolation (Chapter 7) ---
STATES = ("dry", "light", "heavy")       # BMKG: < 0.5, 0.5-20, > 20 mm/day
P = [[0.88, 0.10, 0.02],
     [0.60, 0.32, 0.08],
     [0.50, 0.35, 0.15]]
QUANT = [0.0, 0.25, 0.5, 0.75, 1.0]
RAIN_MM = {1: [0.5, 2.0, 5.0, 10.0, 20.0], 2: [20.0, 26.0, 34.0, 48.0, 95.0]}
ROOF_L_PER_MM = 0.85 * 120.0             # runoff coefficient x roof area (m^2)


def next_state(rng, s):
    u, acc = rng.uniform(), 0.0
    for j in range(3):
        acc += P[s][j]
        if u < acc:
            return j
    return 2


def rain_mm(rng, s):
    return 0.0 if s == 0 else interp(QUANT, RAIN_MM[s], rng.uniform())


# --- trees and well through the season (day 0 = June 1) ---
MID = [14.0, 45.0, 76.0, 106.0, 137.0]  # mid-June ... mid-October
PER_TREE = [25.0, 32.0, 38.0, 42.0, 35.0]        # liters per tree per day
WELL = [2800.0, 2400.0, 1900.0, 1600.0, 1800.0]  # liters the well gives a day
TREES, DAYS, WET_SKIP = 60, 153, 10.0            # no watering after >= 10 mm

# --- the drip line: gravity flow q = K sqrt(level + Z) (Chapter 11) ---
K, Z, WINDOW, DT = 12.0, 1.5, 150.0, 10.0         # L/min/sqrt(m), m, min, min


def session(tank, v0, ration, dt=DT, method="rk4"):
    """Open the valve at 06:00: deliver `ration` liters or stop at the end of
    the window. Returns (liters delivered, minutes the valve was open)."""
    def f(t, v):
        return -K * math.sqrt(tank.level(v) / 100.0 + Z)
```

Listing 16.3 (Continued) *orchard.py: the orchard's water system as one model.*

```
    def step(v, h):
        return rk4_step(f, 0.0, v, h) if method == "rk4" else v + h * f(0.0, v)

    target, t, v = max(v0 - ration, 0.0), 0.0, v0
    while t < WINDOW:
        h = min(dt, WINDOW - t)
        w = step(v, h)
        if w <= target:                 # the ration is out within this step:
            tau = false_position(lambda s: step(v, s) - target, 0.0, h, 1e-6)
            return v0 - target, t + tau  # when exactly? (Chapter 2)
        t, v = t + h, w
    return v0 - v, WINDOW


def season(seed, capacity):
    """One dry season, day by day. Returns the season's totals in liters,
    kilowatt-hours and days."""
    rng, tank = Rng(seed), Tank(capacity)
    v, s = capacity, 0                  # full tank at the end of the rains
    tot = {"rain": 0.0, "overflow": 0.0, "given": 0.0, "deficit": 0.0,
           "pumped": 0.0, "kwh": 0.0, "short_days": 0}
    for d in range(DAYS):
        s = next_state(rng, s)
        mm = rain_mm(rng, s)
        v += ROOF_L_PER_MM * mm
        tot["rain"] += ROOF_L_PER_MM * mm
        if v > capacity:
            tot["overflow"] += v - capacity
            v = capacity
        if mm < WET_SKIP:
            ration = TREES * interp(MID, PER_TREE, d)
            got, _ = session(tank, v, ration)
            v -= got
            tot["given"] += got
            tot["deficit"] += ration - got
            if ration - got > 1e-6:
                tot["short_days"] += 1
        p = min(capacity - v, interp(MID, WELL, d))   # refill in the evening
        v += p
        tot["pumped"] += p
        tot["kwh"] += p / (Q_PUMP * 60.0) * PUMP_KW
    return tot
```

Listing 16.4 is the same model in Java. The pump's coefficients and operating point are computed in a `static` block when the class is loaded, as Python computes them when the module is imported. `season` returns its seven totals as an array in the order `rain, overflow, given, deficit, pumped, kwh, short_days`.

Listing 16.4 `Orchard.java`: the orchard model in Java.

```java
// Pak Hadi's orchard as one model: rain on the shed roof, a tank, a well pump,
// a drip line to 60 trees, and the dry season from June 1 to October 31.
import java.io.IOException;
import java.io.UncheckedIOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;

public class Orchard {
    static final Path DATA = Path.of("..", "data");

    static double[][] readCsv(String name) {
        try {
            List<double[]> rows = new ArrayList<>();
            List<String> lines = Files.readAllLines(DATA.resolve(name));
            for (String line : lines.subList(1, lines.size())) {
                if (line.isBlank()) continue;
                String[] p = line.split(",");
                double[] r = new double[p.length];
                for (int i = 0; i < p.length; i++) r[i] = Double.parseDouble(p[i]);
                rows.add(r);
            }
            return rows.toArray(new double[0][]);
        } catch (IOException e) {
            throw new UncheckedIOException(e);
        }
    }

    static double[] column(double[][] rows, int j) {
        double[] c = new double[rows.length];
        for (int i = 0; i < rows.length; i++) c[i] = rows[i][j];
        return c;
    }

    static String f(String fmt, Object... xs) {
        return String.format(Locale.ROOT, fmt, xs);
    }
```

Listing 16.4 (Continued) *Orchard.java: the orchard model in Java.*

```
    // --- the tank: a bucket-calibrated table, a spline each way (Chapter 7) ---
    static final double[][] TABLE = readCsv("tank_table.csv");
    static final double[] LEVELS = column(TABLE, 0), LITRES = column(TABLE, 1);
    static final Toolbox.Spline VOLUME = new Toolbox.Spline(LEVELS, LITRES);
    static final Toolbox.Spline LEVEL = new Toolbox.Spline(LITRES, LEVELS);
    static final double BASE_CAPACITY = LITRES[LITRES.length - 1];

    /** A battery of tanks of the calibrated shape, capacity in liters. */
    static class Tank {
        final double capacity, scale;

        Tank(double capacity) {
            this.capacity = capacity;
            scale = capacity / BASE_CAPACITY;
        }

        double level(double v) {
            double x = Math.min(Math.max(v / scale, 0.0), BASE_CAPACITY);
            return LEVEL.applyAsDouble(x);
        }
    }

    // --- the pump: datasheet curve by least squares (Chapter 8), operating
    //     point where pump head equals the pipe's need, by Newton (Chapter 3) ---
    static final double[][] CURVE = readCsv("pump_curve.csv");
    static final double[] PCOEF;
    static final double STATIC = 12.0, FRICTION = 0.0011;
    static final double PUMP_KW = 0.8, TARIFF = 1445.0;
    static final double Q_PUMP;

    static {
        double[][] rows = new double[CURVE.length][];
        for (int i = 0; i < CURVE.length; i++) {
            double q = CURVE[i][0];
            rows[i] = new double[] {1.0, q, q * q};
        }
        PCOEF = Toolbox.lstsq(rows, column(CURVE, 1));
        Q_PUMP = Toolbox.newton(Orchard::pumpGap, Orchard::pumpGapSlope,
                                50.0, 1e-10)[0];
    }

    static double pumpGap(double q) {
        return PCOEF[0] + PCOEF[1] * q + PCOEF[2] * q * q
                - (STATIC + FRICTION * q * q);
    }
```

Listing 16.4 (Continued) *Orchard.java: the orchard model in Java.*

```
    static double pumpGapSlope(double q) {
        return PCOEF[1] + 2.0 * PCOEF[2] * q - 2.0 * FRICTION * q;
    }

    // --- weather: a 3-state Markov chain (Chapter 12), rain depth drawn from
    //     a quantile table by linear interpolation (Chapter 7) ---
    static final double[][] P = {{0.88, 0.10, 0.02},
                                 {0.60, 0.32, 0.08},
                                 {0.50, 0.35, 0.15}};
    static final double[] QUANT = {0.0, 0.25, 0.5, 0.75, 1.0};
    static final double[][] RAIN_MM = {null, {0.5, 2.0, 5.0, 10.0, 20.0},
                                       {20.0, 26.0, 34.0, 48.0, 95.0}};
    static final double ROOF_L_PER_MM = 0.85 * 120.0;

    static int nextState(Toolbox.Rng rng, int s) {
        double u = rng.uniform(), acc = 0.0;
        for (int j = 0; j < 3; j++) {
            acc += P[s][j];
            if (u < acc) return j;
        }
        return 2;
    }

    static double rainMm(Toolbox.Rng rng, int s) {
        return s == 0 ? 0.0 : Toolbox.interp(QUANT, RAIN_MM[s], rng.uniform());
    }

    // --- trees and well through the season (day 0 = June 1) ---
    static final double[] MID = {14.0, 45.0, 76.0, 106.0, 137.0};
    static final double[] PER_TREE = {25.0, 32.0, 38.0, 42.0, 35.0};
    static final double[] WELL = {2800.0, 2400.0, 1900.0, 1600.0, 1800.0};
    static final int TREES = 60, DAYS = 153;
    static final double WET_SKIP = 10.0;

    // --- the drip line: gravity flow q = K sqrt(level + Z) (Chapter 11) ---
    static final double K = 12.0, Z = 1.5, WINDOW = 150.0, DT = 10.0;
```

Listing 16.4 (Continued) *Orchard.java: the orchard model in Java.*

```
    /** Open the valve at 06:00: deliver `ration` liters or stop at the end of
     *  the window. Returns {liters delivered, minutes the valve was open}. */
    static double[] session(Tank tank, double v0, double ration, double dt,
                            String method) {
        Toolbox.Ode fn = (t, v) -> -K * Math.sqrt(tank.level(v) / 100.0 + Z);
        boolean rk4 = method.equals("rk4");
        double target = Math.max(v0 - ration, 0.0), t = 0.0, v = v0;
        while (t < WINDOW) {
            double h = Math.min(dt, WINDOW - t), v1 = v;
            double w = rk4 ? Toolbox.rk4Step(fn, 0.0, v, h) : v + h * fn.f(0.0, v);
            if (w <= target) {                  // the ration is out within this step:
                double tau = Toolbox.falsePosition(s -> (rk4
                        ? Toolbox.rk4Step(fn, 0.0, v1, s) : v1 + s * fn.f(0.0, v1))
                        - target, 0.0, h, 1e-6);
                return new double[] {v0 - target, t + tau};  // when? (Chapter 2)
            }
            t += h;
            v = w;
        }
        return new double[] {v0 - v, WINDOW};
    }

    /** One dry season, day by day: {rain, overflow, given, deficit, pumped,
     *  kwh, short_days}. */
    static double[] season(long seed, double capacity) {
        Toolbox.Rng rng = new Toolbox.Rng(seed);
        Tank tank = new Tank(capacity);
        double v = capacity;                  // full tank at the end of the rains
        int s = 0;
        double rain = 0, overflow = 0, given = 0, deficit = 0, pumped = 0, kwh = 0;
        int shortDays = 0;
        for (int d = 0; d < DAYS; d++) {
            s = nextState(rng, s);
            double mm = rainMm(rng, s);
            v += ROOF_L_PER_MM * mm;
            rain += ROOF_L_PER_MM * mm;
            if (v > capacity) {
                overflow += v - capacity;
                v = capacity;
            }
            if (mm < WET_SKIP) {
                double ration = TREES * Toolbox.interp(MID, PER_TREE, d);
                double got = session(tank, v, ration, DT, "rk4")[0];
                v -= got;
                given += got;
                deficit += ration - got;
```

Listing 16.4 (Continued) *Orchard.java: the orchard model in Java.*

```
                if (ration - got > 1e-6) shortDays++;
            }
            double p = Math.min(capacity - v, Toolbox.interp(MID, WELL, d));
            v += p;                           // refill in the evening
            pumped += p;
            kwh += p / (Q_PUMP * 60.0) * PUMP_KW;
        }
        return new double[] {rain, overflow, given, deficit, pumped, kwh,
                             shortDays};
    }
}
```

16.4.3 Checking the parts

Listing 16.5 is the first program to run: it builds every part and checks it against a second method, as Section 16.2 demanded. The tank level is computed by the inverse spline, by Newton and bisection on the forward spline, and by straight lines through the table; the pump fit reports its largest misfit on the datasheet; the stationary weather is computed by the power method and by Gauss–Seidel; a recorded storm is integrated by the trapezoid rule and Simpson's rule; and the drip session is computed with Euler's method and with Runge–Kutta at two step sizes, from a full tank, from 4000 liters and from 3000 liters.

Listing 16.5 `components.py`: every part of the model, checked against a second method.

```
"""Step 1 of the capstone: build every part of the orchard model and check it
against a second method before trusting the whole.   Usage: python components.py"""
import orchard as o
from toolbox import (newton, bisection, interp, gauss_seidel, power_method,
                     trapezoid, simpson, add)

print("1. Tank: level for 6,500 liters (Chapters 2, 3, 6, 7)")
v = 6500.0
print(f"   inverse spline h(V)          {o.LEVEL(v):9.4f} cm")
h, k = newton(lambda x: o.VOLUME(x) - v, o.VOLUME.slope, 100.0, 1e-10)
print(f"   Newton on V(h) = 6500        {h:9.4f} cm in {k} steps")
h2 = bisection(lambda x: o.VOLUME(x) - v, 0.0, 200.0, 1e-10)
print(f"   bisection on V(h) = 6500     {h2:9.4f} cm")
print(f"   straight lines through table {interp(o.LITRES, o.LEVELS, v):9.4f} cm")

print("2. Pump: datasheet fit and operating point (Chapters 3, 4, 8)")
print(f"   head = {o.PA:.4f} {o.PB:+.6f} Q {o.PC:+.8f} Q^2  (m, Q in L/min)")
worst = max(abs(o.PA + o.PB * q + o.PC * q * q - hd) for q, hd in o.CURVE)
print(f"   largest misfit on the datasheet {worst:.3f} m")
head = o.STATIC + o.FRICTION * o.Q_PUMP * o.Q_PUMP
print(f"   operating point {o.Q_PUMP:.3f} L/min at {head:.3f} m")
kwh = 1000.0 / (o.Q_PUMP * 60.0) * o.PUMP_KW
print(f"   1000 liters: {kwh:.4f} kWh, Rp {kwh * o.TARIFF:.0f}")

print("3. Weather: long-run share of dry, light and heavy days (5, 7, 10, 12)")
pi = power_method(o.P, 40)
print("   power method, 40 steps   " + "  ".join(f"{x:.6f}" for x in pi))
A = [[(1.0 if i == j else 0.0) - o.P[j][i] for j in range(3)] for i in range(3)]
x, k = gauss_seidel(A, [0.0, 0.0, 0.0], [1 / 3, 1 / 3, 1 / 3], 1e-13)
tot = add(x)
print(f"   Gauss-Seidel, {k} sweeps   " + "  ".join(f"{v / tot:.6f}" for v in x))
mean_mm = [0.0] + [trapezoid(o.RAIN_MM[s], 0.25) for s in (1, 2)]
per_day = add(p * m for p, m in zip(pi, mean_mm))
print(f"   mean rain: light day {mean_mm[1]:.4f} mm, heavy day {mean_mm[2]:.4f} mm")
print(f"   expected {per_day:.4f} mm a day, {per_day * o.DAYS:.1f} mm a season,"
      f" {per_day * o.DAYS * o.ROOF_L_PER_MM:.0f} L from the roof")

print("4. A storm on the rain gauge, every 10 minutes (Chapter 10)")
storm = o.read_csv("storm.csv")
rate = [r[1] / 60.0 for r in storm]              # mm per minute
t_mm, s_mm = trapezoid(rate, 10.0), simpson(rate, 10.0)
print(f"   trapezoid {t_mm:.4f} mm, Simpson {s_mm:.4f} mm,"
      f" roof catches {s_mm * o.ROOF_L_PER_MM:.0f} L")
```

Listing 16.5 (Continued) *components.py: every part of the model, checked against a second method.*

```
print("5. Drip session for 2,520 liters (Chapters 2, 11)")
tank = o.Tank(13000.0)
for start in (13000.0, 4000.0, 3000.0):
    for method, dt in (("euler", 10.0), ("rk4", 10.0), ("rk4", 0.5)):
        got, minutes = o.session(tank, start, 2520.0, dt, method)
        print(f"   start {start:7.0f} L, {method:5s} dt {dt:3.1f}: {got:8.2f} L"
              f" in {minutes:8.4f} min")
```

Listing 16.6 makes the same checks in Java, in the same order and with the same printed lines.

Listing 16.6 `Components.java`: the same checks in Java.

```
// Step 1 of the capstone: build every part of the orchard model and check it
// against a second method before trusting the whole.   Usage: java Components
public class Components {
    static void out(String fmt, Object... xs) {
        System.out.println(Orchard.f(fmt, xs));
    }

    public static void main(String[] args) {
        System.out.println("1. Tank: level for 6,500 liters (Chapters 2, 3, 6, 7)");
        double v = 6500.0;
        out("   inverse spline h(V)          %9.4f cm",
                Orchard.LEVEL.applyAsDouble(v));
        double[] nw = Toolbox.newton(x -> Orchard.VOLUME.applyAsDouble(x) - v,
                Orchard.VOLUME::slope, 100.0, 1e-10);
        out("   Newton on V(h) = 6500        %9.4f cm in %d steps",
                nw[0], (int) nw[1]);
        double h2 = Toolbox.bisection(x -> Orchard.VOLUME.applyAsDouble(x) - v,
                0.0, 200.0, 1e-10);
        out("   bisection on V(h) = 6500     %9.4f cm", h2);
        out("   straight lines through table %9.4f cm",
                Toolbox.interp(Orchard.LITRES, Orchard.LEVELS, v));

        System.out.println(
                "2. Pump: datasheet fit and operating point (Chapters 3, 4, 8)");
        double[] c = Orchard.PCOEF;
        out("   head = %.4f %+.6f Q %+.8f Q^2  (m, Q in L/min)", c[0], c[1], c[2]);
        double worst = 0.0;
        for (double[] r : Orchard.CURVE)
            worst = Math.max(worst, Math.abs(c[0] + c[1] * r[0] + c[2] * r[0] * r[0]
                    - r[1]));
        out("   largest misfit on the datasheet %.3f m", worst);
        double q = Orchard.Q_PUMP;
        out("   operating point %.3f L/min at %.3f m", q,
                Orchard.STATIC + Orchard.FRICTION * q * q);
        double kwh = 1000.0 / (q * 60.0) * Orchard.PUMP_KW;
        out("   1000 liters: %.4f kWh, Rp %.0f", kwh, kwh * Orchard.TARIFF);
```

Listing 16.6 (Continued) *Components.java: the same checks in Java.*

```
        System.out.println("3. Weather: long-run share of dry, light and heavy days"
                + " (5, 7, 10, 12)");
        double[] pi = Toolbox.powerMethod(Orchard.P, 40);
        out("   power method, 40 steps   %.6f  %.6f  %.6f", pi[0], pi[1], pi[2]);
        double[][] A = new double[3][3];
        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++)
                A[i][j] = (i == j ? 1.0 : 0.0) - Orchard.P[j][i];
        double[] x = {1 / 3.0, 1 / 3.0, 1 / 3.0};
        int k = Toolbox.gaussSeidel(A, new double[3], x, 1e-13);
        double tot = x[0] + x[1] + x[2];
        out("   Gauss-Seidel, %d sweeps   %.6f  %.6f  %.6f",
                k, x[0] / tot, x[1] / tot, x[2] / tot);
        double[] meanMm = {0.0, Toolbox.trapezoid(Orchard.RAIN_MM[1], 0.25),
                           Toolbox.trapezoid(Orchard.RAIN_MM[2], 0.25)};
        double perDay = 0.0;
        for (int s = 0; s < 3; s++) perDay += pi[s] * meanMm[s];
        out("   mean rain: light day %.4f mm, heavy day %.4f mm",
                meanMm[1], meanMm[2]);
        out("   expected %.4f mm a day, %.1f mm a season, %.0f L from the roof",
                perDay, perDay * Orchard.DAYS,
                perDay * Orchard.DAYS * Orchard.ROOF_L_PER_MM);

        System.out.println(
                "4. A storm on the rain gauge, every 10 minutes (Chapter 10)");
        double[][] storm = Orchard.readCsv("storm.csv");
        double[] rate = new double[storm.length];
        for (int i = 0; i < storm.length; i++) rate[i] = storm[i][1] / 60.0;
        double tMm = Toolbox.trapezoid(rate, 10.0);
        double sMm = Toolbox.simpson(rate, 10.0);
        out("   trapezoid %.4f mm, Simpson %.4f mm, roof catches %.0f L",
                tMm, sMm, sMm * Orchard.ROOF_L_PER_MM);

        System.out.println("5. Drip session for 2,520 liters (Chapters 2, 11)");
        Orchard.Tank tank = new Orchard.Tank(13000.0);
        String[] methods = {"euler", "rk4", "rk4"};
        double[] dts = {10.0, 10.0, 0.5};
        for (double start : new double[] {13000.0, 4000.0, 3000.0})
            for (int m = 0; m < 3; m++) {
                double[] r = Orchard.session(tank, start, 2520.0,
                        dts[m], methods[m]);
                out("   start %7.0f L, %-5s dt %3.1f: %8.2f L in %8.4f min",
                        start, methods[m], dts[m], r[0], r[1]);
            }
    }
}
```

16.4.4 Checking the drip model against the orchard

The constant K in Equation (16.7) came from the drip line's specifications. Does the orchard agree? Pak Hadi's tank has an ultrasonic level sensor that logs the level every two minutes to 0.5 cm. **Listing 16.7** turns one morning's log into volumes with the tank spline, estimates the outflow by central differences (Chapter 9), and fits K by least squares (Chapter 8) to $q = Ks$ with $s = \sqrt{h/100 + Z}$, a one-parameter fit whose normal equation is simply $K = \sum s_i q_i / \sum s_i^2$. It fits on the first half of the morning and tests on the second, the held-back check of Chapter 14, with neighbors 2 minutes apart and 10 minutes apart.

Listing 16.7 `sensor.py`: the drip coefficient estimated from the level sensor.

```
"""Check the drip coefficient K against the level sensor (Chapters 8, 9, 14):
outflow from central differences of the logged volume, then least squares for
q = K sqrt(level + Z), fitted on one half and tested on the other.
Usage: python sensor.py"""
import math
import orchard as o
from toolbox import add

log = o.read_csv("sensor_log.csv")
ts = [r[0] for r in log]
vol = [o.VOLUME(r[1]) for r in log]         # liters, from the calibrated table
print(f"{len(log)} readings, {ts[0]:.0f} to {ts[-1]:.0f} min,"
      f" level {log[0][1]:.1f} -> {log[-1][1]:.1f} cm")
for gap in (1, 5):                            # neighbours 2 or 10 minutes away
    pts = []
    for i in range(gap, len(ts) - gap):
        q = -(vol[i + gap] - vol[i - gap]) / (ts[i + gap] - ts[i - gap])
        pts.append((math.sqrt(log[i][1] / 100.0 + o.Z), q))
    half = len(pts) // 2
    fit, test = pts[:half], pts[half:]
    k_fit = add(s * q for s, q in fit) / add(s * s for s, _ in fit)   # 1x1 LS
    rms = math.sqrt(add((q - k_fit * s) * (q - k_fit * s) for s, q in test)
                    / len(test))
    k_all = add(s * q for s, q in pts) / add(s * s for s, _ in pts)
    print(f"step {2 * gap:2d} min: {len(pts)} rates, K (first half)"
          f" {k_fit:.3f}, test RMS {rms:.3f} L/min, K (all) {k_all:.3f}")
print(f"model value K = {o.K:.3f} L/min per sqrt(m)")
```

Listing 16.8 writes the sums of the one-parameter fit as explicit loops, in the order in which Python's `sum` adds them.

Listing 16.8 `Sensor.java`: the same estimate in Java.

```
// Check the drip coefficient K against the level sensor (Chapters 8, 9, 14):
// outflow from central differences of the logged volume, then least squares for
// q = K sqrt(level + Z), fitted on one half and tested on the other.
// Usage: java Sensor
public class Sensor {
    public static void main(String[] args) {
        double[][] log = Orchard.readCsv("sensor_log.csv");
        int n = log.length;
        double[] ts = new double[n], vol = new double[n];
        for (int i = 0; i < n; i++) {
            ts[i] = log[i][0];
            vol[i] = Orchard.VOLUME.applyAsDouble(log[i][1]);
        }
        System.out.println(Orchard.f(
                "%d readings, %.0f to %.0f min, level %.1f -> %.1f cm",
                n, ts[0], ts[n - 1], log[0][1], log[n - 1][1]));
        for (int gap : new int[] {1, 5}) {     // neighbours 2 or 10 minutes away
            int m = n - 2 * gap;
            double[] s = new double[m], q = new double[m];
            for (int i = gap; i < n - gap; i++) {
                q[i - gap] = -(vol[i + gap] - vol[i - gap])
                        / (ts[i + gap] - ts[i - gap]);
                s[i - gap] = Math.sqrt(log[i][1] / 100.0 + Orchard.Z);
            }
            int half = m / 2;
            double sq = 0.0, ss = 0.0;
            for (int i = 0; i < half; i++) sq += s[i] * q[i];
            for (int i = 0; i < half; i++) ss += s[i] * s[i];
            double kFit = sq / ss;                      // 1x1 least squares
            double e2 = 0.0;
            for (int i = half; i < m; i++) {
                double e = q[i] - kFit * s[i];
                e2 += e * e;
            }
            double rms = Math.sqrt(e2 / (m - half));
            double sqa = 0.0, ssa = 0.0;
            for (int i = 0; i < m; i++) sqa += s[i] * q[i];
            for (int i = 0; i < m; i++) ssa += s[i] * s[i];
            System.out.println(Orchard.f("step %2d min: %d rates, K (first half)"
                    + " %.3f, test RMS %.3f L/min, K (all) %.3f",
                    2 * gap, m, kFit, rms, sqa / ssa));
        }
        System.out.println(
                Orchard.f("model value K = %.3f L/min per sqrt(m)", Orchard.K));
    }
}
```

16.4.5 Many seasons in parallel

Listing 16.9 runs *N* seasons for a given capacity. The seasons are cut into one chunk per worker; each chunk runs in its own process (`ProcessPoolExecutor`, because the simulation is pure Python, and Python threads would take turns, Chapter 15), and the results come back in season order. `report` computes the means of Equation (16.9) with Kahan's sum, counts the seasons with any shortage, and reads the 90th percentile of the missed water from the sorted list.

Listing 16.9 `montecarlo.py`: many dry seasons in parallel, reproducibly.

```
"""Many dry seasons at once (Chapter 15): every season has its own random
stream, seeded by its number, so the results do not depend on the number of
worker processes.   Usage: python montecarlo.py [seasons] [workers] [liters]"""
import sys
from concurrent.futures import ProcessPoolExecutor
import orchard as o
from toolbox import kahan_sum

SEED0 = 20260601
KEYS = ("rain", "overflow", "given", "deficit", "pumped", "kwh", "short_days")


def chunk(job):
    capacity, first, count = job
    rows = []
    for i in range(first, first + count):
        tot = o.season(SEED0 + i, capacity)
        rows.append([tot[k] for k in KEYS])
    return rows


def run_seasons(capacity, n, workers):
    """Totals of seasons 0..n-1, in season order, computed by `workers`."""
    size = -(-n // workers)                          # ceiling division
    jobs = [(capacity, a, min(size, n - a)) for a in range(0, n, size)]
    if workers == 1:
        parts = [chunk(j) for j in jobs]
    else:
        with ProcessPoolExecutor(max_workers=workers) as pool:
            parts = list(pool.map(chunk, jobs))
    return [row for part in parts for row in part]


def report(capacity, rows):
    n = len(rows)
    mean = {k: kahan_sum([r[i] for r in rows]) / n for i, k in enumerate(KEYS)}
    deficits = sorted(r[3] for r in rows)
    short = sum(1 for d in deficits if d > 1.0)
    print(f"tank {capacity:.0f} L, {n} seasons:")
    print(f"  rain caught {mean['rain']:9.0f} L,"
          f" overflow {mean['overflow']:9.0f} L")
    print(f"  pumped      {mean['pumped']:9.0f} L,"
          f" electricity {mean['kwh']:.1f} kWh = Rp {mean['kwh'] * o.TARIFF:.0f}")
    print(f"  trees got   {mean['given']:9.0f} L, missed {mean['deficit']:.0f} L on"
          f" {mean['short_days']:.1f} days on average")
    print(f"  seasons with any shortage: {short} of {n};"
          f" 9 in 10 miss less than {deficits[int(0.9 * n)]:.0f} L")


def main():
    n = int(sys.argv[1]) if len(sys.argv) > 1 else 400
    workers = int(sys.argv[2]) if len(sys.argv) > 2 else 2
    caps = [float(sys.argv[3])] if len(sys.argv) > 3 else [13000.0, 30000.0]
    for cap in caps:
        report(cap, run_seasons(cap, n, workers))


if __name__ == "__main__":
    main()
```

Listing 16.9 (Continued) *montecarlo.py: many dry seasons in parallel, reproducibly.*

Listing 16.10 does the same with a pool of Java threads; each task simulates a chunk of seasons and returns their totals, and `invokeAll` returns the futures in the order the tasks were given, so the results are assembled in season order whatever thread finished first.

Listing 16.10 `MonteCarlo.java`: the same Monte Carlo run with Java threads.

```
// Many dry seasons at once (Chapter 15): every season has its own random
// stream, seeded by its number, so the results do not depend on the number of
// threads.   Usage: java MonteCarlo [seasons] [threads] [liters]
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class MonteCarlo {
    static final long SEED0 = 20260601L;

    /** Totals of seasons 0..n-1, in season order, computed by `threads`. */
    static double[][] runSeasons(double capacity, int n, int threads)
            throws Exception {
        int size = (n + threads - 1) / threads;
        ExecutorService pool = Executors.newFixedThreadPool(threads);
        List<Callable<double[][]>> jobs = new ArrayList<>();
        for (int a = 0; a < n; a += size) {
            final int first = a, count = Math.min(size, n - a);
            jobs.add(() -> {
                double[][] rows = new double[count][];
                for (int i = 0; i < count; i++)
                    rows[i] = Orchard.season(SEED0 + first + i, capacity);
                return rows;
            });
        }
        double[][] all = new double[n][];
        int k = 0;
        for (Future<double[][]> f : pool.invokeAll(jobs))
            for (double[] row : f.get()) all[k++] = row;
        pool.shutdown();
        return all;
    }

    static double mean(double[][] rows, int j) {
        double[] col = new double[rows.length];
        for (int i = 0; i < rows.length; i++) col[i] = rows[i][j];
        return Toolbox.kahanSum(col) / rows.length;
    }
```

Listing 16.10 (Continued) *MonteCarlo.java: the same Monte Carlo run with Java threads.*

```
    static void report(double capacity, double[][] rows) {
        int n = rows.length;
        double[] deficits = new double[n];
        int shortN = 0;
        for (int i = 0; i < n; i++) {
            deficits[i] = rows[i][3];
            if (deficits[i] > 1.0) shortN++;
        }
        Arrays.sort(deficits);
        System.out.println(Orchard.f("tank %.0f L, %d seasons:", capacity, n));
        System.out.println(Orchard.f("  rain caught %9.0f L, overflow %9.0f L",
                mean(rows, 0), mean(rows, 1)));
        System.out.println(Orchard.f(
                "  pumped      %9.0f L, electricity %.1f kWh = Rp %.0f",
                mean(rows, 4), mean(rows, 5), mean(rows, 5) * Orchard.TARIFF));
        System.out.println(Orchard.f(
                "  trees got   %9.0f L, missed %.0f L on %.1f days on average",
                mean(rows, 2), mean(rows, 3), mean(rows, 6)));
        System.out.println(Orchard.f(
                "  seasons with any shortage: %d of %d;"
                + " 9 in 10 miss less than %.0f L",
                shortN, n, deficits[(int) (0.9 * n)]));
    }

    public static void main(String[] args) throws Exception {
        int n = args.length > 0 ? Integer.parseInt(args[0]) : 400;
        int threads = args.length > 1 ? Integer.parseInt(args[1]) : 2;
        double[] caps = args.length > 2 ? new double[] {Double.parseDouble(args[2])}
                : new double[] {13000.0, 30000.0};
        for (double cap : caps) report(cap, runSeasons(cap, n, threads));
    }
}
```

16.4.6 Choosing the tank

Listing 16.11 wraps Equation (16.10) in a function of the capacity that runs the same 200 seasons each time, hands it to the golden-section search of the toolbox on the interval from 10,000 to 50,000 liters with a tolerance of 500 liters, and prints every capacity the search tried with its cost parts, then the best, then the newsvendor prediction Equation (16.11).

Listing 16.11 `design.py`: the tank size by golden-section search on the yearly cost.

```
"""How big should the tank be? Golden-section search (Chapter 13) on the
yearly cost, estimated from the same simulated seasons for every size.
Usage: python design.py [seasons] [workers]   (default 200 seasons, 2 workers)"""
import sys
import orchard as o
from montecarlo import run_seasons
from toolbox import golden_min, kahan_sum, add

TANK_RP_PER_L_YEAR = 150.0      # Rp 1,500 per liter of tank, over 10 years
MISSED_RP_PER_L = 500.0         # fruit lost per liter the trees did not get


def main():
    n = int(sys.argv[1]) if len(sys.argv) > 1 else 200
    workers = int(sys.argv[2]) if len(sys.argv) > 2 else 2
    seen = []

    def cost(capacity):
        rows = run_seasons(capacity, n, workers)    # common random numbers
        kwh = kahan_sum([r[5] for r in rows]) / n
        missed = kahan_sum([r[3] for r in rows]) / n
        short = sum(1 for r in rows if r[3] > 1.0) / n
        parts = (TANK_RP_PER_L_YEAR * capacity, kwh * o.TARIFF,
                 MISSED_RP_PER_L * missed)
        seen.append((capacity, short) + parts + (add(parts),))
        return add(parts)

    best = golden_min(cost, 10000.0, 50000.0, 500.0)
    print(f"{n} seasons per size; yearly cost in thousand rupiah")
    print("  tank (L)  P(short)   tank  power  missed   total")
    for cap, short, t, e, m, total in seen:
        print(f"  {cap:8.0f}  {short:8.3f}  {t / 1e3:5.0f}  {e / 1e3:5.1f}"
              f"  {m / 1e3:6.0f}  {total / 1e3:6.0f}")
    total = cost(best)
    print(f"best tank about {best:.0f} L, yearly cost {total / 1e3:.0f} thousand;"
          f" P(short) there {seen[-1][1]:.3f}")
    print(f"newsvendor check: P(short) should be near"
          f" {TANK_RP_PER_L_YEAR / MISSED_RP_PER_L:.3f}")


if __name__ == "__main__":
    main()
```

Listing 16.12 passes the cost to the toolbox's `goldenMin` as a lambda that records every evaluation, as the Python version does with its list `seen`.

Listing 16.12 `Design.java`: the same search in Java.

```
// How big should the tank be? Golden-section search (Chapter 13) on the
// yearly cost, estimated from the same simulated seasons for every size.
// Usage: java Design [seasons] [threads]   (default 200 seasons, 2 threads)
import java.util.ArrayList;
import java.util.List;

public class Design {
    static final double TANK_RP_PER_L_YEAR = 150.0;  // Rp 1,500 a liter, 10 years
    static final double MISSED_RP_PER_L = 500.0;      // fruit lost per liter missed

    public static void main(String[] args) throws Exception {
        int n = args.length > 0 ? Integer.parseInt(args[0]) : 200;
        int threads = args.length > 1 ? Integer.parseInt(args[1]) : 2;
        List<double[]> seen = new ArrayList<>();
        java.util.function.DoubleUnaryOperator cost = capacity -> {
            double[][] rows;
            try {
                rows = MonteCarlo.runSeasons(capacity, n, threads);  // same seasons
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
            double kwh = MonteCarlo.mean(rows, 5);
            double missed = MonteCarlo.mean(rows, 3);
            int shortN = 0;
            for (double[] r : rows) if (r[3] > 1.0) shortN++;
            double t = TANK_RP_PER_L_YEAR * capacity, e = kwh * Orchard.TARIFF;
            double m = MISSED_RP_PER_L * missed, total = t + e + m;
            seen.add(new double[] {capacity, (double) shortN / n, t, e, m, total});
            return total;
        };
        double best = Toolbox.goldenMin(cost, 10000.0, 50000.0, 500.0);
        System.out.println(n + " seasons per size; yearly cost in thousand rupiah");
        System.out.println("  tank (L)  P(short)   tank  power  missed   total");
        for (double[] r : seen)
            System.out.println(Orchard.f(
                    "  %8.0f  %8.3f  %5.0f  %5.1f  %6.0f  %6.0f",
                    r[0], r[1], r[2] / 1e3, r[3] / 1e3, r[4] / 1e3, r[5] / 1e3));
        double total = cost.applyAsDouble(best);
        System.out.println(Orchard.f(
                "best tank about %.0f L, yearly cost %.0f thousand;"
                + " P(short) there %.3f",
                best, total / 1e3, seen.get(seen.size() - 1)[1]));
        System.out.println(Orchard.f(
                "newsvendor check: P(short) should be near %.3f",
                TANK_RP_PER_L_YEAR / MISSED_RP_PER_L));
    }
```

Listing 16.12 (Continued) *Design.java: the same search in Java.*

```
}
```

16.5 Running the Simulation

```
cd Chapter16/python
python3 components.py
python3 sensor.py
python3 montecarlo.py
python3 design.py
cd ../java
javac -d out *.java
java -cp out Components
java -cp out Sensor
java -cp out MonteCarlo
java -cp out Design
```

16.5.1 The parts

`components.py` prints:

```
1. Tank: level for 6,500 liters (Chapters 2, 3, 6, 7)
   inverse spline h(V)           115.0174 cm
   Newton on V(h) = 6500         115.0144 cm in 4 steps
   bisection on V(h) = 6500      115.0144 cm
   straight lines through table  114.7929 cm
2. Pump: datasheet fit and operating point (Chapters 3, 4, 8)
   head = 31.9958 -0.023155 Q -0.00411310 Q^2  (m, Q in L/min)
   largest misfit on the datasheet 0.087 m
   operating point 59.752 L/min at 15.927 m
   1000 liters: 0.2231 kWh, Rp 322
3. Weather: long-run share of dry, light and heavy days (5, 7, 10, 12)
   power method, 40 steps   0.828813  0.138638  0.032550
   Gauss-Seidel, 13 sweeps   0.828813  0.138638  0.032550
   mean rain: light day 6.8125 mm, heavy day 41.3750 mm
   expected 2.2912 mm a day, 350.6 mm a season, 35757 L from the roof
4. A storm on the rain gauge, every 10 minutes (Chapter 10)
   trapezoid 38.9167 mm, Simpson 39.1111 mm, roof catches 3989 L
5. Drip session for 2,520 liters (Chapters 2, 11)
   start   13000 L, euler dt 10.0:  2520.00 L in 114.5825 min
   start   13000 L, rk4   dt 10.0:  2520.00 L in 114.8071 min
   start   13000 L, rk4   dt 0.5:  2520.00 L in 114.8071 min
   start    4000 L, euler dt 10.0:  2520.00 L in 146.7040 min
   start    4000 L, rk4   dt 10.0:  2520.00 L in 147.2514 min
   start    4000 L, rk4   dt 0.5:  2520.00 L in 147.2514 min
   start    3000 L, euler dt 10.0:  2460.34 L in 150.0000 min
   start    3000 L, rk4   dt 10.0:  2450.54 L in 150.0000 min
   start    3000 L, rk4   dt 0.5:  2450.54 L in 150.0000 min
```

Every part passes its check. The tank level for 6500 liters is 115.017 cm by the inverse spline and 115.014 cm by Newton (four steps) and bisection on the forward spline, a difference of 0.03 mm between two reasonable interpolants of the same bucket table; straight lines through the table would be off by 2 mm, still harmless. The pump's quadratic misses the datasheet by at most 9 cm of head, and the operating point, 59.75 liters per minute at 15.9 m, matches the hand computation of Section 16.3.2. Power method and Gauss–Seidel agree on the weather to six digits, and the expected 35,757 liters of roof water per season is the benchmark for the simulation. The storm integrates to 38.9 mm by the trapezoid rule and 39.1 mm by Simpson's rule; the difference, half a percent, is the size of the error one expects from 10-minute samples of a peaked storm, and the roof caught about 4000 liters from it.

The drip sessions show both the physics and the numerics. From a full tank the ration takes 114.8 minutes, from 4000 liters 147.3 minutes, just inside the window, and from 3000 liters the window closes after 2450.5 liters: 70 liters short with 550 liters still in the tank. Runge–Kutta with 10-minute steps agrees with 0.5-minute steps to all printed digits, while Euler's method with 10-minute steps is off by a quarter of a minute and by 10 liters, an error of order Δt as Chapter 11 predicts.

16.5.2 The drip model against the sensor

`sensor.py` prints:

```
46 readings, 0 to 90 min, level 175.5 -> 151.0 cm
step  2 min: 44 rates, K (first half) 11.819, test RMS 6.839 L/min, K (all) 11.856
step 10 min: 36 rates, K (first half) 11.910, test RMS 1.253 L/min, K (all) 11.899
model value K = 12.000 L/min per sqrt(m)
```

With neighbors 2 minutes apart the level changes by only about half a centimeter between readings, the size of the sensor's resolution, so each rate is mostly rounding noise: the test error is 6.8 liters per minute on rates of about 22. With neighbors 10 minutes apart the change is five times larger and the noise five times smaller relative to it (Chapter 9's trade-off between truncation and noise), and the test error drops to 1.25 liters per minute. Both estimates of K, 11.82 to 11.91, lie within 1.5% of the model's 12, and the half that was not used for fitting is predicted as well as the half that was. The drip model is consistent with the orchard.

16.5.3 Four hundred seasons

`montecarlo.py` simulates 400 seasons for the present tank and for a 30,000-liter one:

```
tank 13000 L, 400 seasons:
  rain caught     35870 L, overflow     20496 L
  pumped         258005 L, electricity 57.6 kWh = Rp 83192
  trees got      280146 L, missed 14166 L on 26.2 days on average
  seasons with any shortage: 390 of 400; 9 in 10 miss less than 24446 L
tank 30000 L, 400 seasons:
  rain caught     35870 L, overflow     19543 L
  pumped         258745 L, electricity 57.7 kWh = Rp 83431
  trees got      292307 L, missed 2005 L on 5.8 days on average
  seasons with any shortage: 145 of 400; 9 in 10 miss less than 7446 L
```

The roof water, 35,870 liters on average, agrees with the hand estimate of 35,757 within the Monte Carlo error: the first check of the whole. With the present 13,000-liter tank, 390 of 400 seasons end with a shortage, and on average the trees miss 14,166 liters on 26 days. With 30,000 liters, 145 seasons in 400 have any shortage, the average missed water falls to 2005 liters, and in nine seasons out of ten the trees miss less than 7446 liters. The electricity hardly changes (Rp 83,000 a season): the pump lifts about the same water whatever the tank, because the water the trees drink has to come from somewhere. More than half of the rain overflows, because the evening refill keeps the tank full whenever the well can; Section 16.7 takes that up.

Figure 16.2 shows one of the 400 seasons with both tanks. Until early August the well keeps up and both tanks stay nearly full. Then the well falls behind, and the tanks drain steadily at the same rate, about 900 liters a day by September. The small tank is empty by day 93 (September 2) and the trees go short every day for three weeks, exactly Pak Hadi's experience of last year; the large tank lasts until day 111, when a short dry spell before the first October rains empties it too. The rains of mid-October refill both.

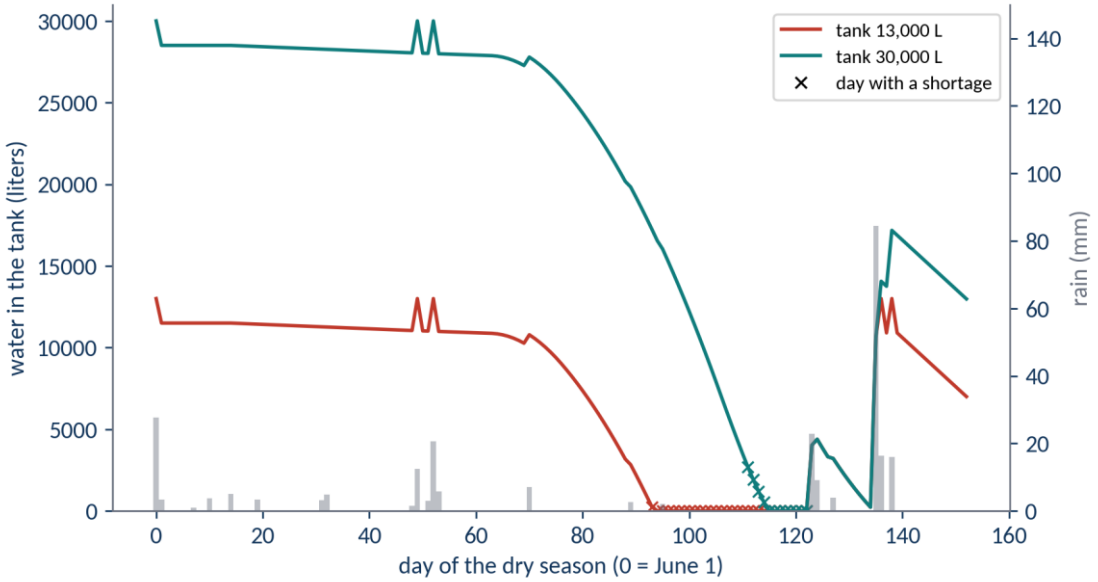


Figure 16.2 One simulated dry season (the same weather) with a 13,000-liter and a 30,000-liter tank: water in the tank after the morning watering, days on which the trees got less than their ration (crosses), and daily rain (bars, right axis).

16.5.4 The best tank

`design.py` prints:

```
200 seasons per size; yearly cost in thousand rupiah
  tank (L)  P(short)   tank  power  missed   total
     25279     0.590   3792   83.4    2082    5958
     34721     0.175   5208   83.4     347    5639
     40557     0.060   6084   83.4      64    6231
     31115     0.315   4667   83.4     804    5555
     28885     0.415   4333   83.4    1199    5616
     32492     0.270   4874   83.4     601    5558
     30263     0.335   4539   83.4     939    5562
     31641     0.305   4746   83.4     723    5552
     31966     0.285   4795   83.4     675    5553
     31440     0.310   4716   83.4     753    5553
     31765     0.300   4765   83.4     704    5552
     31564     0.305   4735   83.4     734    5552
best tank about 31602 L, yearly cost 5552 thousand; P(short) there 0.305
newsvendor check: P(short) should be near 0.300
```

The golden-section search tries twelve capacities and settles at about 31,600 liters, with a yearly cost of 5.55 million rupiah: 4.7 million for the tank, 0.08 million for electricity and 0.7 million of lost fruit. The present tank would cost 1.95 million a year but lose an expected $500 \times 14{,}166 \approx 7.1$ million of fruit, a total of about 9 million. **Figure 16.3** draws the three parts of the cost over the whole range: the tank line rises steadily, the missed-water curve falls steeply and flattens, and their sum has a broad, flat bottom between about 29,000 and 34,000 liters, where the cost varies by less than 2%. Among the tried capacities the probability of a shortage at the optimum is 0.305, as the newsvendor rule Equation (16.11) predicted: the simulation and the one-line theory confirm each other.

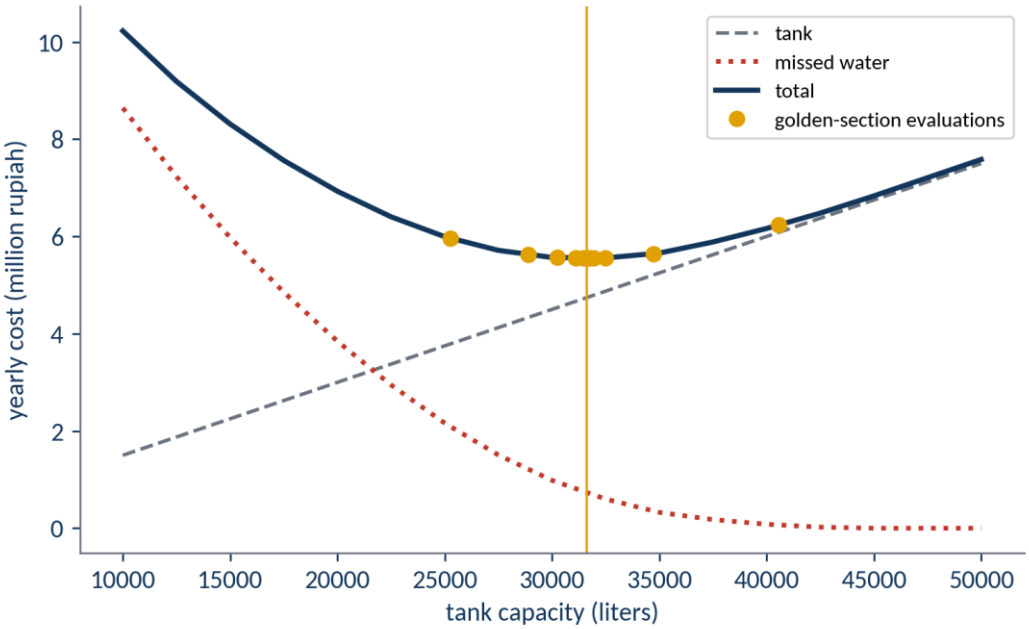


Figure 16.3 Yearly cost against tank capacity from 200 simulated seasons per size: tank cost, value of missed water, and total, with the capacities tried by the golden-section search and the chosen size.

The flat bottom is the most useful part of the answer. It means that Pak Hadi does not need to buy exactly 31,602 liters, a number that carries more digits than the rain notebook deserves. If tanks of 5,000 liters are available, three more beside the present one give 28,000 liters and four give 33,000, both within 2% of the minimum cost. The choice between them can rest on things the model does not know: the space on the stand, the money available this year, and how much he trusts the notebook's September.

16.6 Comparing with SciPy

Listing 16.13 recomputes the parts with library routines: `CubicSpline` with natural end conditions for the tank, `polyfit` for the pump curve, `brentq` (Brent's bracketed root finder) for the operating point, `eig` for the stationary weather, `simpson` for the storm, and `solve_ivp` with an event function and tight tolerances for the drip session [7], [8].

Listing 16.13 `check_scipy.py`: the parts of the model recomputed with NumPy and SciPy.

```
"""The capstone's parts recomputed with NumPy and SciPy.
Usage: python check_scipy.py   (needs numpy and scipy)"""
import numpy as np
from scipy.integrate import simpson, solve_ivp
from scipy.interpolate import CubicSpline
from scipy.optimize import brentq
import orchard as o

inv = CubicSpline(o.LITRES, o.LEVELS, bc_type="natural")
print(f"tank: level for 6500 L {float(inv(6500.0)):.4f} cm"
      f" (ours {o.LEVEL(6500.0):.4f})")
q, h = np.array(o.CURVE).T
c2, c1, c0 = np.polyfit(q, h, 2)
print(f"pump: polyfit {c0:.4f} {c1:+.6f} Q {c2:+.8f} Q^2")
qop = brentq(o.pump_gap, 0.0, 100.0, xtol=1e-12)
print(f"      operating point {qop:.6f} L/min (ours {o.Q_PUMP:.6f})")
w, v = np.linalg.eig(np.array(o.P).T)
pi = np.real(v[:, np.argmin(np.abs(w - 1.0))])
print("weather: eigenvector for eigenvalue 1 "
      + "  ".join(f"{x:.6f}" for x in pi / pi.sum()))
storm = np.array(o.read_csv("storm.csv"))
print(f"storm: scipy simpson {simpson(storm[:, 1] / 60.0, x=storm[:, 0]):.4f} mm")
tank = o.Tank(13000.0)
sol = solve_ivp(lambda t, y: [-o.K * np.sqrt(tank.level(y[0]) / 100.0 + o.Z)],
                (0.0, 150.0), [13000.0], rtol=1e-10, atol=1e-8,
                events=lambda t, y: y[0] - (13000.0 - 2520.0))
print(f"drip: solve_ivp reaches 2520 L after {sol.t_events[0][0]:.4f} min"
      f" (ours {o.session(tank, 13000.0, 2520.0)[1]:.4f})")
```

```
tank: level for 6500 L 115.0174 cm (ours 115.0174)
pump: polyfit 31.9958 -0.023155 Q -0.00411310 Q^2
      operating point 59.751951 L/min (ours 59.751951)
weather: eigenvector for eigenvalue 1 0.828813  0.138638  0.032550
storm: scipy simpson 39.1111 mm
drip: solve_ivp reaches 2520 L after 114.8071 min (ours 114.8071)
```

Every number agrees with ours to the printed digits, including the drip session's end at 114.8071 minutes, which SciPy finds with an adaptive Runge–Kutta method of order 5 and a root finder on its dense output. The toolbox of a few hundred lines is therefore as accurate as the library for this problem; what the library adds is generality, adaptivity and years of testing, which is why for new problems it is the first choice and a hand-written method the second opinion.

16.7 Making It Better

A simulation is never finished; it is good enough for a decision. Four improvements would make this one better, and each is a small change to `orchard.py`.

- **Leave room for the rain.** The evening refill keeps the tank full, so more than half of the roof water overflows. Refilling only to, say, 90% of capacity in August and September leaves room for a storm; whether it helps depends on how often a storm comes before the tank is needed, which is exactly what the simulation can answer. Exercise 16.4 shows that here it does not.
- **Test the sensitivity.** The model's numbers are uncertain: the well's September yield, the transition probabilities, the value of lost fruit. Changing each by 10% and rerunning `design.py` shows which ones move the answer. For the newsvendor rule the answer is predictable: the best capacity is where $\Pr[\text{shortage}] = a/p$, so a higher value of fruit moves it to larger tanks, and the move is large when the distribution of shortfalls is wide.

- **Reduce the variance.** Common random numbers already make comparisons between capacities precise. Running more seasons reduces the remaining error only as $1/\sqrt{N}$ (Equation (16.9)); for a final report, 2000 seasons in Java take a few seconds.
- **Model the well better.** The well's yield is a table of five numbers. If the level of the water in the well were logged, its recovery after pumping could be fitted as a differential equation of its own (Chapters 8 and 11), and the pump's daily volume would then depend on how much was pumped the day before.

16.8 Using It for Real

The model is written so that another orchard, a household, a school or a small clinic with a rainwater tank can use it by changing the named constants and the data files.

- **Calibrate your tank.** Fill it with a bucket of known size and write down the level every few centimeters; a table of eight to ten points and the spline do the rest. For a standard cylindrical tank the table is a straight line and the spline reproduces it exactly.
- **Read your pump's datasheet** into `pump_curve.csv` and measure the lift from the water level in the well to the top of the tank. Check the computed flow once with a bucket and a stopwatch at the tank inlet.
- **Keep a rain notebook.** Count, for each dry-season month, how often a dry day is followed by a dry, a light or a heavy day, and note the depths; the transition matrix and the quantile tables come straight from the counts. BMKG's categories [5] make the notebook comparable with the official forecasts.
- **Write down the water you give**, per tree, per plant bed or per household, month by month, and what the well or the municipal supply gives. These two tables, not the rain, usually decide the answer.
- **Set the two prices honestly**: what a liter of storage costs per year, and what a liter of missed water costs you. The newsvendor rule Equation (16.11) then tells you, before any simulation, what risk of shortage the best tank will carry; the simulation tells you how big that tank is.

For a household the "missed water" is buying from a water truck. Suppose a family pays Rp 150,000 for a 5,000-liter truck delivery, Rp 30 per liter, and a tank costs Rp 150 per liter per year. Then $a/p = 150/30 = 5 > 1$: no tank beyond the smallest practical one pays for itself on this account, and the rule says to buy water when short. The same rule with drinking water that cannot be replaced at any price (p very large) says to make shortages very rare. The model advises; the family decides, and for drinking water safety comes first.

16.9 Numerical Methods in Practice: Profitina

The Profitina AI Visibility platform is, like this chapter's orchard, a system of parts, each a numerical method, assembled into one pipeline [9]. A scan of a website measures many signals, each computed by its own method and each checked on its own; the signals are normalized and combined by weights into a score, as the tank, pump and weather are combined into a season; and the score is followed over time in a weekly trend, with a scheduler that re-scans each brand at a cadence set by its subscription tier. The discipline of Section 16.2, check every part against an independent method before trusting the whole, is the discipline of a release gate: the full automated suite of well over a thousand tests must pass before a release, and the release is reported successful only when the deployed version answers its health check.

The PROFIT utility coin is simulated in the same spirit as the orchard. Its supply follows a decaying emission with burning, and its price in rupiah comes from an oracle [9]. How the supply evolves under different patterns of use is a question of a small dynamical model stepped through time, run for many possible futures, and summarized by averages and ranges: Section 16.3.6 and Section 16.3.7 in another setting. The rule that Profitina's answers are advice and not guarantees is the rule of this chapter's last paragraph: a simulation shows what is likely under stated assumptions, and the person who owns the decision makes it.

16.10 Common Mistakes, and How to Avoid Them

- **Assembling before checking.** A wrong sign in the pump fit or a spline built on swapped columns produces a simulation that runs and prints plausible numbers. Check every part against a second method first, as `components.py` does.
- **Too few seasons, or different weather for each option.** Comparing a 25,000-liter tank on one set of 50 seasons with a 30,000-liter tank on another set compares luck. Use the same seeds for every option and enough seasons that the standard error Equation (16.9) is small compared with the differences you care about.
- **Seeding from the clock or from the thread.** A simulation whose results change from run to run, or with the number of cores, cannot be tested or reproduced. Seed each season from its number.
- **Trusting digits the data do not have.** The best tank is "about 30,000 liters", not 31,602. Report the flat range of **Figure 16.3**, not the exact optimum.
- **Averages alone.** The average missed water hides the bad seasons. Report also the probability of any shortage and a high percentile, as `montecarlo.py` does; the decision may depend on them.
- **A step size copied from somewhere.** The drip session looks smooth, but Euler's method with 10-minute steps was already off by 10 liters. Check the integrator against a smaller step, as in Section 16.5.1.
- **Forgetting what the model leaves out.** Pests, a broken pump, a neighbor who draws from the same well, or a dry season unlike any in the notebook are not in the model. Say so when presenting the answer.

16.11 Chapter Summary

- A real question, how large a water tank a durian orchard needs for the dry season, was answered by a simulation built from the methods of the whole book (**Table 16.1**, **Figure 16.1**).
- Each part was modeled by the fitting method and checked by a second one: splines and Newton for the tank (Equation (16.1)), least squares and Newton for the pump (Equation (16.2), Equation (16.3)), a Markov chain solved by the power method and Gauss–Seidel for the weather (Equation (16.5), Equation (16.6)), Runge–Kutta with an event time for the drip line (Equation (16.7)), and differences with least squares against the level sensor.
- The daily water balance Equation (16.8) ran for 153 days per season and 400 seasons in parallel, reproducibly, with seeded streams and compensated sums; the Monte Carlo error falls like $1/\sqrt{N}$ (Equation (16.9)).
- The present 13,000-liter tank runs short in almost every season; golden-section search on the yearly cost Equation (16.10) chose about 31,600 liters, with a flat optimum from about 29,000 to 34,000 liters.

- At the optimum the probability of a shortage was 0.305, as the newsvendor rule Equation (16.11) predicts from the ratio of the price of storage to the price of missed water; theory and simulation confirm each other.
- Java and Python printed identical results, and SciPy agreed with every part.

16.12 Exercises

1. By hand, find the stationary distribution of Equation (16.5) from the balance equations of the first two states and the sum condition, and check it against the program.
2. Using Equation (16.7), estimate by hand with one Euler step of 10 minutes how much water a full 13,000-liter tank delivers in the first 10 minutes. Why is the answer larger than the Runge–Kutta value, and by roughly how much?
3. Show that for a tank whose table is a straight line through the origin, $V = \alpha h$, the natural cubic spline reproduces the line exactly. What does this mean for cylindrical tanks?
4. Change `season` so that in August and September (days 61 to 121) the evening refill stops at 90% of capacity, and rerun `montecarlo.py` for 30,000 liters. Explain the change in overflow and in missed water.
5. The value of lost fruit is uncertain. Without running the simulation, use the newsvendor rule to say what shortage probability the best tank should have if the value is Rp 300 or Rp 1,000 per liter, and in which direction the best capacity moves. Then check one case with `design.py`.
6. How many seasons would be needed to estimate the probability of a shortage at 30,000 liters with a standard error of 0.01? How long would that take with the Java program, and with the Python program on two workers?

16.13 Solutions to the Exercises

1. The stationary equations $\pi_j = \sum_i \pi_i P_{ij}$ for $j = 2$ and $j = 3$ are $\pi_2 = 0.10\pi_1 + 0.32\pi_2 + 0.35\pi_3$ and $\pi_3 = 0.02\pi_1 + 0.08\pi_2 + 0.15\pi_3$. The second gives $0.85\pi_3 = 0.02\pi_1 + 0.08\pi_2$. The first gives $0.68\pi_2 = 0.10\pi_1 + 0.35\pi_3$. Substituting $\pi_3 = (0.02\pi_1 + 0.08\pi_2)/0.85$: $0.68\pi_2 = 0.10\pi_1 + 0.008235\pi_1 + 0.032941\pi_2$, so $0.647059\pi_2 = 0.108235\pi_1$ and $\pi_2 = 0.167273\pi_1$; then $\pi_3 = (0.02 + 0.08 \times 0.167273)\pi_1/0.85 = 0.039273\pi_1$. The sum condition gives $\pi_1(1 + 0.167273 + 0.039273) = 1$, so $\pi_1 = 0.828813$, $\pi_2 = 0.138638$, $\pi_3 = 0.032550$, exactly the program's values.

2. A full tank has $h = 200$ cm, so the flow is $12\sqrt{2.0 + 1.5} = 22.45$ liters per minute, and one Euler step of 10 minutes delivers 224.5 liters. The true flow decreases during the 10 minutes as the level falls (by about 2.6 cm in these 10 minutes, since $V'(h) \approx 85$ liters per cm near the top), so the exact delivery is slightly less; Euler uses the flow at the start of the step for the whole step and therefore overestimates, by about half of the flow's change over the step times the step, a few tenths of a liter here. Over a whole session these errors add up to the quarter of a minute by which Euler's session ends early in Section 16.5.1.

3. For data on the line $V = \alpha h$, all divided differences of the second order are zero, so the right-hand side of the spline's tridiagonal system is zero, the second derivatives m_i are all zero, and each piece reduces to the linear interpolant, which is the line itself. A cylindrical tank has a linear table, so its spline is exact, and the same code handles cylindrical, tapered and irregular tanks; only an irregular table needs the spline's curvature.

4. Replace the refill line by `top = 0.9 * capacity if 61 <= d <= 121 else capacity` and `p = min(top - v, interp(MID, WELL, d))` with `max(p, 0.0)` in case the tank is already above `top`.

Over the same 400 seasons the average overflow falls from 19,543 to 16,266 liters, since storms now find up to 3000 liters of room, but the missed water *rises* from 2005 to 3142 liters, and the seasons with a shortage from 145 to 200. The tank enters the long September shortfall with 10% less water, and the few storms of August and September do not return that much. With this notebook's weather, leaving room for rain costs more than it saves; it would pay only where storms are frequent during the shortage.

5. The rule Equation (16.11) gives Pr[shortage] $= 150/300 = 0.5$ for Rp 300 and $150/1000 = 0.15$ for Rp 1,000. A lower value of fruit tolerates shortages in half of the seasons, so the best tank is smaller; the table printed by `design.py` shows a shortage probability of 0.59 at 25,279 liters and 0.415 at 28,885, so the new optimum lies near 27,000 liters. For Rp 1,000 the table shows 0.175 at 34,721 and 0.060 at 40,557, so the optimum moves to about 35,000 liters. Running `design.py` with the changed constant confirms both: 26,819 liters with a shortage probability of 0.515, and 35,085 liters with 0.155.

6. From Equation (16.9), $\mathrm{SE} = \sqrt{p(1-p)/N} = 0.01$ with $p \approx 0.36$ gives $N = 0.36 \times 0.64 / 0.0001 = 2304$ seasons. The Python simulation takes about 9 ms per season on the test machine, so 2304 seasons take about 21 seconds of processor time, about 11 seconds on two workers; the Java program does 400 seasons for two capacities in well under a second, so 2304 seasons take about a second.

References

[1] S. M. Ross, *Simulation*, 6th ed. London, U.K.: Academic Press, 2023.

[2] S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers*, 8th ed. New York, NY, USA: McGraw-Hill Education, 2021.

[3] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical Analysis*, 10th ed. Boston, MA, USA: Cengage Learning, 2016.

[4] CNBC Indonesia, "Tarif Listrik Terbaru PLN per kWh, Berlaku Hingga September 2026," Aug. 10, 2026. [Online]. Available: https://www.cnbcindonesia.com/news/20260810101545-4-757861/tarif-listrik-terbaru-pln-per-kwh-berlaku-hingga-september-2026

[5] Badan Meteorologi, Klimatologi, dan Geofisika, "Data Prakiraan Cuaca Terbuka," accessed Sep. 2026. [Online]. Available: https://data.bmkg.go.id/prakiraan-cuaca/

[6] S. Nahmias and T. L. Olsen, *Production and Operations Analysis*, 7th ed. Long Grove, IL, USA: Waveland Press, 2015.

[7] The SciPy community, "scipy.integrate.solve_ivp," SciPy 1.18.1 documentation, 2026. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html

[8] P. Virtanen *et al.*, "SciPy 1.0: Fundamental algorithms for scientific computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[9] Profitina, "Profitina — AI Visibility (GEO) platform," 2026. [Online]. Available: https://profitina.com