

Modern Financial Modeling

Concepts and Applications for Actuaries and Other Financial Professionals

Alec Loudenback Yun-Tien Lee

2026-03-02

Modern Financial Modeling: Concepts and Applications for Actuaries and Other Financial Professionals

Special Edition

Copyright © 2026 Alec Loudenback and Yun-Tien Lee.

Published by Alec Loudenback and Yun-Tien Lee.

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License (CC BY-NC-SA 4.0), except where otherwise noted. To view a copy of this license, visit <https://creativecommons.org/licenses/by-nc-sa/4.0/>.

The chapters on developing in Julia (*Writing Julia*, *Debugging Julia*, *Sharing Julia*, and *Optimizing Julia*) are licensed under the Creative Commons Attribution-ShareAlike 4.0 International License (CC BY-SA 4.0). To view a copy of this license, visit <https://creativecommons.org/licenses/by-sa/4.0/>.

This is not a Milliman-sponsored project.

ISBN 978-1-257-97605-8

First edition (Special Edition), 2026.

Published online at <https://ModernFinancialModeling.com>.

Table of contents

Preface	1
I. Introduction	5
The Approach	7
What You Will Learn	8
The Journey Ahead	8
What to Expect	8
Prerequisites	9
The Contents of This Book	9
A Crash-Course	9
Notes on formatting	10
Colophon	11
1. Why Program?	13
1.1. Chapter Overview	13
1.2. Introduction	13
1.2.1. Market Forces	14
1.3. Why Programming Matters Now	15
1.4. The Spectrum of Programming in Finance	16
1.5. Artificial Intelligence (AI) ‘{=latex}’	16
1.6. Low Code	17
1.7. The 10x Modeler	17
1.8. Risk Governance	18
1.9. Managing and Leading the Transformation	19
1.10. Outlook	19
2. Why use Julia?	21
2.1. Chapter Overview	21
2.2. Introduction	21
2.3. Julia and This Book	22
2.4. Expressiveness and Syntax	22
2.4.1. Example 1: Retention Analysis	23
2.4.2. Example 2: Random Sampling	24
2.5. The Speed	26
2.5.1. Development Speed	28
2.6. More of Julia’s benefits	28
2.7. Tradeoffs when Using Julia	29
2.7.1. Just-In-Time Compilation	29
2.7.2. Static Binaries	30
2.8. Package Ecosystem	30
2.9. Tools in Your Toolbox	30
2.9.1. Interoperability	31

Table of contents

II. Foundations: Effective Financial Modeling	33
3. Elements of Financial Modeling	37
3.1. Chapter Overview	37
3.2. What is a Model?	37
3.2.1. Small World vs. Large World	38
3.3. What is a <i>Financial Model</i> ?	38
3.3.1. Difference from Data Science	39
3.4. Key Considerations for a Model	39
3.5. Predictive versus Explanatory Models	40
3.5.1. A Historical Example	40
3.5.2. Examples in the Financial Context	42
3.5.3. Predictive vs Explanatory Models in Practice	43
3.6. Types of Models	43
4. The Practice of Financial Modeling	47
4.1. Chapter Overview	47
4.2. What makes a good model?	47
4.2.1. Achieving original purpose	47
4.2.2. Usability	47
4.2.3. Performance	48
4.2.4. Separation of Model Logic and Data	48
4.2.5. Organization of Model Components and Architecture	49
4.2.6. Abstraction of Modeled Systems	49
4.3. What makes a good modeler?	50
4.3.1. Domain Expertise	50
4.3.2. Model Theory	50
4.3.3. Curiosity	52
4.3.4. Rigor	52
4.3.5. Clarity	53
4.3.6. Humility	53
4.3.7. Architecture	56
4.3.8. Planning	56
4.3.9. Essential Tools and Skills	57
4.4. Feeding The Model	58
4.4.1. “Garbage In, Garbage Out”	58
4.4.2. A Modeler’s Data Instincts	58
4.4.3. Data Is Never “Done”	59
4.5. Model Management	60
4.5.1. Risk Governance	60
4.5.2. Change Management	60
4.5.3. Data Controls	61
4.5.4. Peer and Technical Review	61
4.6. Conclusion	62
III. Foundations: Programming and Abstractions	63
5. Elements of Programming	67
5.1. Chapter Overview	67
5.2. Computer Science, Programming, and Coding	67
5.3. Expressions and Control Flow	69
5.3.1. Naming Values with Variables	69
5.3.2. Expressions	69
5.3.3. Equality	73

5.3.4. Assignment, References, and Mutability	74
5.3.5. Loops	75
5.3.6. Performance of Loops	76
5.4. Data Types	76
5.4.1. Numbers	76
5.4.2. Type Hierarchy	79
5.4.3. Collections	80
5.4.4. Parametric Types	86
5.4.5. Types for things not there	87
5.4.6. Union Types	88
5.4.7. Creating User Defined Types	88
5.4.8. Mutable structs	91
5.4.9. Constructors	91
5.5. Functions	92
5.5.1. Special Operators	93
5.5.2. Defining Functions	94
5.5.3. Defining Methods on Types	94
5.5.4. Keyword Arguments	96
5.5.5. Default Arguments	96
5.5.6. Anonymous Functions	96
5.5.7. First Class Nature	97
5.5.8. Broadcasting	97
5.5.9. Passing by Sharing	100
5.5.10. The Function Type	100
5.6. Scope	101
5.6.1. Modules and Namespaces	103
6. Functional Abstractions	105
6.1. Chapter Overview	105
6.2. Introduction	105
6.3. Imperative Style	106
6.3.1. Iterators	106
6.4. Functional Techniques and Terminology	109
6.4.1. map	112
6.4.2. accumulate	113
6.4.3. reduce	115
6.4.4. mapreduce	116
6.4.5. filter	116
6.4.6. More Tips on Functional Styles	117
6.5. Array-Oriented Styles	119
6.6. Recursion	122
7. Data and Types	125
7.1. Chapter Overview	125
7.2. Using Types to Value a Portfolio	125
7.3. Benefits of Using Types	125
7.4. Defining Types for Portfolio Valuation	126
7.4.1. Dispatch	128
7.5. Object-Oriented Design	131
7.5.1. Assigning Behavior	132
7.5.2. Inheritance	137
7.6. Data-Oriented Design	140
8. Higher Levels of Abstraction	141
8.1. Chapter Overview	141

Table of contents

8.2. Introduction	141
8.3. Principles for Abstraction	141
8.3.1. Pragmatic Considerations for Model Design	143
8.4. Interfaces	143
8.4.1. Defining Good Interfaces	144
8.4.2. Interfaces: A Financial Modeling Case Study	144
8.4.3. Abstraction Design Checklist	147
8.5. Macros & Homoiconicity	148
8.5.1. Metaprogramming in Financial Modeling	149
8.6. Key Takeaways	149
IV. Foundations: Building Performant Models	151
9. Hardware and Its Implications	155
9.1. Chapter Overview	155
9.2. Introduction	155
9.3. Memory and Moving Data Around	155
9.3.1. Memory Types and Location	156
9.3.2. Stack vs Heap	157
9.4. Processor	157
9.5. Logistics Warehouse Analogy	162
9.6. Speed of Computer Actions	163
10. Writing Performant Single-Threaded Code	167
10.1. Chapter Overview	167
10.2. Introduction	167
10.3. Patterns of Performant Sequential Code	168
10.3.1. Minimize Memory Allocations	168
10.3.2. Optimize Memory Access Patterns	169
10.3.3. Use Efficient Data Types	172
10.3.4. Avoid Type Instabilities	173
10.3.5. Optimize for Branch Prediction	174
10.3.6. Further Reading	175
10.3.7. Key Takeaways	176
11. Parallelization	177
11.1. Chapter Overview	177
11.2. Amdahl's Law and the Limits of Parallel Computing	177
11.3. Types of Parallelism	179
11.4. Vectorization	180
11.4.1. Hardware	181
11.5. Multi-Threading	182
11.5.1. Tasks	182
11.5.2. Multi-Threading Overview	184
11.6. GPUs and TPUs	189
11.6.1. Hardware	189
11.6.2. Utilizing a GPU	190
11.7. Multi-Processing / Multi-Device	196
11.8. Parallel Programming Models	199
11.8.1. Map-Reduce	199
11.8.2. Array-Based	200
11.8.3. Loop-Based	201
11.8.4. Task-Based	202
11.9. Choosing a Parallelization Strategy	202

11.10 References	203
V. Interdisciplinary Concepts and Applications	205
12. Applying Software Engineering Practices	209
12.1. Chapter Overview	209
12.2. Introduction	209
12.2.1. Regulatory Compliance and Software Practices	209
12.2.2. Chapter Structure	210
12.3. Testing	210
12.3.1. Test Driven Development	211
12.3.2. Test Coverage	212
12.3.3. Types of Tests	213
12.4. Documentation	214
12.4.1. Comments	214
12.4.2. Docstrings	215
12.4.3. Docsites	216
12.5. Version Control	217
12.5.1. Git Overview	217
12.5.2. Change governance checklist	220
12.5.3. Collaborative Workflows	221
12.5.4. Data Version Control	223
12.6. Distributing the Package	226
12.6.1. Registries	226
12.6.2. Versioning	227
12.6.3. Artifacts	229
12.7. Example Repository Structure	230
13. Elements of Computer Science	231
13.1. Chapter Overview	231
13.2. Computer Science for Financial Professionals	231
13.3. Algorithms	232
13.4. Complexity	232
13.4.1. Computational Complexity	232
13.4.2. Space Complexity	236
13.4.3. Complexity: Takeaways	236
13.4.4. Finance Example: Complexity and Portfolio Selection	237
13.5. Data Structures	241
13.5.1. Arrays	241
13.5.2. Linked Lists	242
13.5.3. Records/Structs	242
13.5.4. Dictionaries (Hash Tables)	243
13.5.5. Graphs	244
13.5.6. Trees	245
13.5.7. Data Structures Conclusion	246
13.6. Verification and Advanced Testing	246
13.6.1. Formal Verification	246
13.6.2. Property Based Testing	247
13.6.3. Fuzzing	247
14. Statistical Inference and Information Theory	249
14.1. Chapter Overview	249
14.2. Introduction	249

Table of contents

14.3. Information Theory	249
14.3.1. Example: The Missing Digit	250
14.3.2. Example: Classification	251
14.3.3. Maximum Entropy Distributions	254
14.4. Bayes' Rule	258
14.4.1. Bayes' Rule Formula	258
14.4.2. Model Selection via Likelihoods	259
14.5. Modern Bayesian Statistics	265
14.5.1. Background	265
14.5.2. Implications for Financial Modeling	268
14.5.3. Basics of Bayesian Modeling	268
14.5.4. Markov-Chain Monte Carlo	269
14.5.5. MCMC Algorithms	273
14.5.6. Rainfall Example (Continued)	274
14.5.7. Conclusion	282
14.5.8. Further Reading	282
15. Stochastic Modeling	283
15.1. Chapter Overview	283
15.2. Introduction	283
15.3. Fundamentals of Stochastic Modeling	284
15.3.1. Random Variables and Probability	284
15.3.2. The Role of Monte Carlo Simulation	284
15.3.3. Time Dimensions in Stochastic Models	284
15.3.4. State Spaces and Transitions	285
15.3.5. Aleatory vs Epistemic Uncertainty	285
15.3.6. Calibration and Validation	285
15.3.7. Computational Considerations	285
15.3.8. Kinds of Stochastic Models	286
15.3.9. Components of Stochastic Models	288
15.4. Evaluating and Applying Stochastic Models	289
15.4.1. Evaluating Stochastic Results	289
15.4.2. Variance Reduction Techniques	295
15.4.3. Stochastic Modeling: Examples	296
15.5. Pseudo-Random Number Generators	298
15.5.1. Common PRNGs	299
15.5.2. Consistent Interface	300
15.6. Conclusion	301
16. Automatic Differentiation	303
16.1. Chapter Overview	303
16.2. Motivation for (Automatic) Derivatives	303
16.3. Finite Differentiation	303
16.4. Automatic Differentiation	306
16.4.1. Dual Numbers	306
16.5. Performance of Automatic Differentiation	309
16.6. Automatic Differentiation in Practice	310
16.6.1. Performance	312
16.7. Forward Mode and Reverse Mode	313
16.8. Practical Tips for Automatic Differentiation	313
16.8.1. Choosing between Reverse Mode and Forward Mode	314
16.8.2. Mutation	314
16.8.3. Custom Rules	314
16.8.4. Available Libraries in Julia	314

16.9. References	314
17. Optimization	315
17.1. Chapter Overview	315
17.2. Introduction	315
17.3. Objective and Loss Functions	317
17.4. Optimization Techniques	317
17.4.1. Nonlinear Optimization	317
17.4.2. Linear optimization	330
17.5. Example: Model Fitting	332
17.6. More Resources	334
18. Visualizations	335
18.1. Chapter Overview	335
18.2. Introduction	335
18.3. Developing Visualizations	338
18.3.1. Define Your Message	338
18.3.2. Emphasize Accuracy and Integrity	338
18.3.3. Prioritize Clarity Over Complexity	339
18.3.4. Organize Data Thoughtfully	339
18.3.5. Enhance Readability	339
18.3.6. Validate and Iterate	339
18.3.7. Example: Improving a Disease Funding Visualization	339
18.4. Principles of Good Visualization	342
18.5. Types of visualization tools	343
18.5.1. Additional Examples	348
18.6. Julia Plotting Packages	349
18.6.1. CairoMakie.jl and GLMakie.jl	349
18.6.2. Plots.jl	349
18.6.3. GraphPlot.jl	349
18.6.4. UnicodePlots.jl	349
18.7. References	349
19. Matrices and Their Uses	351
19.1. Chapter Overview	351
19.2. Matrix manipulation	351
19.2.1. Addition and subtraction	351
19.2.2. Transpose	352
19.2.3. Determinant	352
19.2.4. Trace	353
19.2.5. Norm	353
19.2.6. Multiplication	354
19.2.7. Inversion	355
19.3. Matrix decomposition	356
19.3.1. Eigenvalues	356
19.3.2. Singular values	357
19.3.3. Matrix Factorization and Factorization Machines	358
19.3.4. Principal component analysis	360
20. Learning from Data	361
20.1. Chapter Overview	361
20.2. How to learn from data	361
20.2.1. Understand the problem and define goals	361
20.2.2. Collect data	361
20.2.3. Explore and preprocess the data	361

Table of contents

20.2.4. Exploratory data analysis (EDA)	362
20.2.5. Select and engineer features	362
20.2.6. Choose the right algorithm or model	362
20.2.7. Train and evaluate the model	362
20.2.8. Tune hyperparameters	363
20.2.9. Deploy and monitor the model	363
20.2.10. Draw insights and make decisions	363
20.2.11. Limitations	363
20.3. Applications	363
20.3.1. Parameter fitting	364
20.3.2. Forecasting	364
VI. Developing in Julia	367
21. Writing Julia Code	371
21.1. Chapter Overview	371
21.2. Getting help	371
21.3. Installation	371
21.4. REPL	372
21.4.1. Help mode (?)	372
21.4.2. Package mode (])	373
21.4.3. Shell mode (;)	374
21.5. Editor	374
21.6. Running code	374
21.7. Notebooks	375
21.8. Markdown	376
21.8.1. Plain Text Markdown	376
21.8.2. Quarto	376
21.9. Environments and Dependencies	377
21.9.1. Project.toml	377
21.9.2. Manifest.toml	379
21.9.3. Reproducibility	380
21.9.4. A Recommended Environment Set-up	380
21.10. Creating Local packages	381
21.10.1. PkgTemplates.jl	382
21.11. Development workflow	382
21.12. Configuration	383
21.13. Interactivity	384
21.14. Testing	385
22. Troubleshooting Julia Code	389
22.1. Chapter Overview	389
22.2. Error Messages and Stack Traces	389
22.2.1. Error Types	390
22.3. Logging	390
22.4. Commonly Encountered Macros	392
22.5. Debugging	393
22.5.1. Setting	393
22.5.2. Infiltrator.jl	393
22.5.3. Debugger.jl	394
23. Distributing and Sharing Julia Code	397
23.1. Chapter Overview	397
23.2. Setup	397

23.3. GitHub Actions	398
23.4. Testing	398
23.4.1. Code Coverage	399
23.5. Code Style	399
23.5.1. Formatters	400
23.6. Code quality	400
23.7. Documentation	401
23.8. Literate programming	401
23.9. Versions and registration	402
23.9.1. Versions and Compatibility	402
23.9.2. Registration	402
23.10 Reproducibility	403
23.11 Interoperability	404
23.12 Customization	404
23.13 Collaboration	404
24. Optimizing Julia Code	405
24.1. Chapter Overview	405
24.2. Type Inference	405
24.3. Avoiding Heap Allocations	405
24.4. Measuring performance	406
24.4.1. BenchmarkTools	406
24.4.2. Other tools	407
24.5. Profiling	407
24.5.1. Sampling	407
24.5.2. Visualizing Profile Results	408
24.5.3. External profilers	409
24.6. Type stability	409
24.6.1. Detecting Instabilities	409
24.6.2. Fixing Instabilities	412
24.7. Memory management	412
24.8. Compilation	413
24.8.1. Precompilation	413
24.8.2. Package compilation	413
24.8.3. Static compilation	414
24.9. Parallelism	415
24.9.1. Multithreading	415
24.9.2. Distributed computing	416
24.9.3. GPU programming	418
24.9.4. SIMD instructions	418
24.9.5. Additional Packages	419
24.10 Efficient types	419
24.10.1. Static arrays	419
24.10.2. Classic data structures	419
24.10.3. Bits types	420
24.10.4. Putting it into practice	420
VII. Applied Financial Modeling Techniques	421
25. Scenario Generation	425
25.1. Chapter Overview	425
25.2. Common Use Cases of Scenario Generators	425
25.2.1. Risk management, especially Value at Risk (VaR) & Expected Shortfall (ES) .	425
25.2.2. Stress Testing & Regulatory Compliance	425

Table of contents

25.2.3. Portfolio Optimization & Asset Allocation	425
25.2.4. Pension & Insurance Risk Modeling	426
25.2.5. Economic & Macro-Financial Forecasting	426
25.2.6. Asset Pricing & Hedging	426
25.2.7. Fixed Income & Interest Rate Modeling	426
25.2.8. Regulatory & Accounting Valuations	426
25.3. Common Economic Scenario Generation Approaches	426
25.3.1. Interest Rate Models	426
25.3.2. Equity Models	434
25.3.3. Copulas	438
25.4. Where to Go Next	441
26. Similarity Analysis	443
26.1. Chapter Overview	443
26.2. The Data	443
26.3. Common Similarity Measures	445
26.3.1. Euclidean Distance (L2 norm)	445
26.3.2. Manhattan Distance (L1 Norm)	445
26.3.3. Cosine Similarity	446
26.3.4. Jaccard Similarity	446
26.3.5. Hamming Distance	446
26.3.6. Choosing a Metric	447
26.4. k-Nearest Neighbor (kNN) Clustering	447
27. Portfolio Optimization	451
27.1. Chapter Overview	451
27.2. The Data	451
27.3. Theory	451
27.4. Mathematical tools	452
27.4.1. Mean-variance optimization model	452
27.4.2. Efficient frontier analysis	454
27.4.3. Black-Litterman	455
27.4.4. Risk Parity	457
27.4.5. Sharpe Ratio Maximization	458
27.4.6. Robust Optimization	459
27.4.7. Asset weights from different methodologies	460
27.5. Key Takeaways	461
27.6. Practical considerations	461
27.6.1. Fractional purchases of assets	461
27.6.2. Large number of assets	461
28. Sensitivity Analysis	463
28.1. Chapter Overview	463
28.2. Setup	463
28.3. The Data	464
28.4. Common Sensitivity Analysis Methodologies	465
28.4.1. Finite Differences	465
28.4.2. Regression Analyses	466
28.4.3. Sobol Indices	466
28.4.4. Morris Method	467
28.4.5. Fourier Amplitude Sensitivity Tests	468
28.4.6. Automatic Differentiation	469
28.4.7. Scenario Analyses	469

29. Automatic Differentiation for Asset Liability Management	471
29.1. Chapter Overview	471
29.2. Interest Rate Curve Setup	471
29.3. Asset Valuation Framework	472
29.4. Liability Modeling	473
29.5. Computing Derivatives with Autodiff	477
29.5.1. Gradients and Hessians in ALM	478
29.6. Optimizing an Asset Portfolio	480
29.6.1. Define Asset Universe	480
29.6.2. Optimization Routine	481
29.6.3. Results	482
29.7. Computational Benefits	484
29.8. Conclusion	485
29.9. Appendix: Even more performance (Advanced)	485
30. Stochastic Mortality Projections	489
30.1. Chapter Overview	489
30.2. Introduction	489
30.3. Data and Types	489
30.3.1. @enums and the Policy Type	489
30.3.2. The Data	491
30.4. Model Assumptions	492
30.4.1. Mortality Assumption	492
30.5. Model Structure and Mechanics	493
30.5.1. Core Model Behavior	493
30.5.2. Inputs and Outputs	494
30.5.3. Threading	494
30.5.4. Simulation Control	494
30.6. Running the projection	495
30.6.1. Stochastic Projection	495
30.7. Benchmarking	497
30.8. Conclusion	498
31. Bayesian Mortality Modeling	499
31.1. Chapter Overview	499
31.2. Generating fake data	499
31.3. 1: A single binomial parameter model	502
31.3.1. Sampling from the posterior	503
31.4. 2. Parametric model	504
31.4.1. Plotting samples from the posterior	506
31.5. 3. Multi-level model	507
31.6. Handling non-unit exposures	509
31.7. Model Predictions	511
32. More Useful Techniques	513
32.1. Chapter Overview	513
32.2. General Modeling Techniques	513
32.2.1. Taking Things to the Extreme	513
32.2.2. Range Bounding	513
32.2.3. Pseudo-Monte Carlo Sanity Checks	514
32.2.4. Model Validation	514
32.2.5. Predictive vs. Explanatory Models	516
32.2.6. Causal Modeling	516
32.2.7. Other Techniques Worth Knowing	517

Table of contents

32.3. Programming Techniques	517
32.3.1. Serialization	517
32.3.2. Memoization	518
32.3.3. Automated Benchmarks	519
VIII Appendices	521
33. The Julia Ecosystem for Financial Modeling	523
33.1. Chapter Overview	523
33.2. Why Composability Matters	523
33.3. Data Handling	524
33.4. Data Structures and Utilities	524
33.5. Time and Dates	524
33.6. Statistics and Probability	524
33.7. Optimization	525
33.8. Automatic Differentiation	525
33.9. JuliaActuary	525
33.10. Visualization	526
33.11. Numerical Computing	526
33.12. Parallel and Concurrent Computing	527
33.13. Machine Learning	527
33.14. Developer Utilities	527
33.15. Finding More Packages	527
References	529
Index	531
Index	533

Preface

This open-access book enables financial professionals and advanced students to apply the tools and concepts of computational and related sciences to their work.

Throughout these pages, you'll see how to effectively represent financial modeling ideas so they remain scalable, intelligible to collaborators, and robust against changing conditions. The book covers fundamentals in programming to advanced topics like parallelization and data visualization, always returning to practical applications in insurance, asset modeling, and beyond.

The goal is straightforward: to empower financial professionals with the knowledge and confidence to manipulate computational tools in ways that reveal ever richer insights into the systems we seek to understand and influence.

This book is available in HTML and PDF.

A print version is available for purchase at <https://ModernFinancialModeling.com/purchase.html>.

Preface

Dedication

Alec Loudenback

To Camille, Sophie, and Claire.

Yun-Tien Lee

To The Lord, Emily, Esther, and Eunice.

Dedication

Part I.

Introduction

"I think one of the things that really separates us from the high primates is that we're tool builders. I read a study that measured the efficiency of locomotion for various species on the planet. The condor used the least energy to move a kilometer. And, humans came in with a rather unimpressive showing, about a third of the way down the list. It was not too proud a showing for the crown of creation. So, that didn't look so good. But, then somebody at Scientific American had the insight to test the efficiency of locomotion for a man on a bicycle. And, a man on a bicycle, a human on a bicycle, blew the condor away, completely off the top of the charts.

And that's what a computer is to me. What a computer is to me is it's the most remarkable tool that we've ever come up with, and it's the equivalent of a bicycle for our minds." - Steve Jobs (1990)

The world of financial modeling is complex and variegated. Like many sciences, financial modeling blends practical goals with computational tools to arrive at answers that (we hope) are meaningful in a way that tells us more about the world we live in. Practitioners utilize computers to do the heavy work of processing data or running simulations that reveal insights about the complex systems we seek to represent. In this way, then, financial modelers must also be craftsmen who seek not only to design new products, but must also think carefully about the tools and the process used therein.

This book helps you develop that workmanship: We will develop new ways to look at the *process*, think about how to most clearly represent ideas, dive into details about computer hardware, connect the low-level details with high level abstractions, and adopt a vocabulary to more clearly express and communicate these concepts. The book contains a large number of practical examples to demonstrate that the end result is superior when adopting computer science concepts.

This book addresses programming for the applied financial professional, starting with a fundamental question: "*Why is this relevant for financial modeling?*" The answer is simple: Financial modeling is complex, data-intensive, and often very abstract. Programming is the best tool humans have so far developed for rigorously transforming ideas and data into results. A builder may be the most skilled person in the world with a hammer, but another builder with some basic training in a richer set of tools will build a better house. This book will enhance your toolkit with a language to talk about solving problems, a deeper understanding of specific problem solving techniques, how to architect a thoughtful solution for complex problems, and practical advice from experienced practitioners.

The Approach

The authors of the book are practicing actuaries, but we intend for the content to be applicable to nearly all practitioners in the financial industry. The discussion and examples may have an orientation towards insurance topics, but the concepts and patterns generalize to asset management, risk, treasury, and valuation.

We will pull from examples on both sides of the balance sheet: the left (assets) and the right (liabilities). We may also, at times, take liberties with traditional accounting conventions: a liability is just an asset with the obligor and obligee switched. When the accounting conventions are important (such as modeling a total balance sheet) we will be mindful in explaining the accounting perspective. In practice, this means that we'll present examples of assets (fixed income, equity, derivatives) and liabilities (life insurance, annuities, long-term care) and show that similar modeling techniques can be used for both.

What You Will Learn

It is our hope that with the help of this book, you will find it more efficient to discuss aspects of modeling with colleagues, borrow problem-solving language from computer science, spot recurring structural patterns in problems that arise, and understand how best to make use of the “bicycle for your mind” in the context of financial modeling.

It is the experience of the authors that many professionals that do complex modeling as a part of their work have become very proficient *despite* not having substantive formal training on problem solving, algorithms, or model architecture. This book serves to fill that gap and provide the “missing semester” (and years of practical learning). After reading this book, we hope that you will appreciate the attributes of Microsoft Excel that made it so ubiquitous, but that you prefer to use a programming language for the ability to more naturally express the relevant abstractions which make your models simpler, faster, or more usable by others.

Even if your role does not entail hands-on coding (e.g., management or low-code tools), the ideas and language should help guide the work to a cleaner, more efficient solution.

The Journey Ahead

Learning a new topic, especially one that’s not well-trodden in a given field, can be intimidating. There are many resources available online. This book will recommend some additional resources to point you in the right direction. There is also community support available - check the chat and forums and look for the users talking about the topics that interest you. One of the wonderful things about the technology community is the degree to which content is available online for learning and reference.

Moving substantial parts of the financial services industry towards a digital-first, modern workflow is a monumental effort and you should seek partners on both the finance and information technology side. In general, good ideas and processes will prevail. The trick to encouraging adoption is finding the right place to plug a new idea or suggestion.

Additionally, this book provides the language and technical knowledge to partner with others (such as peers and IT) to make pragmatic decisions about the trade-offs that will need to be made.

What to Expect

This book will guide you through:

1. Core programming concepts applied to finance
2. Modern software development practices
3. Computational approaches to common financial problems
4. Real-world examples and applications

The goal is to build both theoretical understanding and practical skills you can apply immediately in your work.

Prerequisites

Basic experience with financial modeling is not strictly required, but familiarity with basic concepts will help anchor your understanding.

Advanced financial math (e.g., stochastic calculus) is *not* required. Indeed, this book is not oriented to the advanced technicalities of Wall Street “quants” and is instead directed at the multitudes of financial practitioners focused on producing results that are not measured in the microseconds of high-frequency trading.

Prior programming experience is not required; see Chapter 5, which introduces the basic syntax and concepts while Chapter 21 covers setting up your environment to follow along. For readers with background in programming, we recommend skimming Chapter 5.

The Contents of This Book

The book is organized into eight parts, each addressing key aspects of computational thinking and financial modeling. **Part I** introduces foundational concepts, explaining why programming matters for financial professionals, and why Julia is a particularly well-suited programming language for financial modeling applications.

Parts II, III, and IV establish the theoretical foundations—covering effective financial modeling practices, programming abstractions, and techniques for building performant models. These sections bridge theory with practical implementation, exploring topics like model design, functional programming, data types, and parallelization strategies.

Part V connects interdisciplinary concepts with practical applications, demonstrating how software engineering practices, computer science principles, statistical methods, and visualization techniques enhance financial modeling.

Since the rest of the book is intended to be generally agnostic about the choice of programming language, **Part VI** focuses on Julia tooling and advice. It provides detailed guidance on developing in Julia, from writing and troubleshooting code to optimization. This is the section that really leans into Julia-specific ideas and workflows.

Part VII showcases applied financial modeling techniques through real-world examples, including stochastic mortality projections, scenario generation, sensitivity analysis, and portfolio optimization.

While Julia is used for the examples throughout the book, the concepts presented are largely language-agnostic. The principles of computational thinking and financial modeling remain applicable regardless of implementation language. Readers are encouraged to follow along with the examples on their own computers. The entire book is available at <https://ModernFinancialModeling.com>.

A Crash-Course

This text is written with the intention to explain and show many core and related concepts that are useful for a practitioner utilizing code-based modeling and workflows. If you are looking for a crash-course in getting up to speed in order to contribute to a functional project, here is a condensed syllabus for getting the most out of this book:

Table 0.1.: A suggested short course reading list for chapters from this book.

How to Read	Chapter Number	Chapter Title	Why
Skim and re-reference	5	Elements of Programming	Programming syntax and core components provided by most languages
Read in entirety	6	Functional Abstractions	Patterns of functional abstractions that are repeatedly useful in financial modeling
Read in entirety	7	Data and Types	Utilizing types of data to improve efficiency and architecture of models
Read in entirety	12	Applying Software Engineering Practices	Best practices for code-based workflows, including version control, testing, documentation, and code distribution.
Skim and re-reference	21-24	Developing in Julia	Writing, troubleshooting, distributing, and optimizing Julia specific code. Skim lightly if using a language other than Julia; you'll learn what kinds of tools to look for in other ecosystems.

Notes on formatting

When a concept is defined for the first time, the term will be **bold**. Code, or references to pieces of code, will be formatted in inline code style like `1+1` or in separate code blocks:

"This is a code block that doesn't show any results"

"This is a code block that does show output"

"This is a code block that does show output"

When we show inline commands to be sent to Pkg mode in the REPL (see Section 21.9), such as `add DataFrames`, we will try to make it clear in the context. If using Pkg mode in standalone codeblocks, it will be presented showing the full prompt, such as:

(@v1.12) pkg> add DataFrames

There will be various callout blocks which indicate tips or warnings. These should be self-evident but we wanted to point to a particular callout which is intended to convey advice that stems from practical modeling experience of the authors:

 Financial Modeling Pro Tip

This box indicates a side note that's particularly applicable to improving your financial modeling.

Colophon

The HTML and PDF versions of the book were rendered using Quarto and Quarto's open source dependencies like Pandoc and LaTeX.

The HTML version of this book uses Lato for the body font and JuliaMono for the monospace font.

The PDF version of this book uses TeX Gyre Pagella for the body font and JuliaMono for the monospace font.

The cover was designed by Alec Loudenback using Affinity Designer with the graphic used with permission by GitHub user cormullion.

This book was rendered on February 25, 2026. The system used to generate the code and benchmarks was:

```
versioninfo()

Julia Version 1.12.5
Commit 5fe89b8ddc1 (2026-02-09 16:05 UTC)
Build Info:
    Official https://julialang.org release
Platform Info:
    OS: macOS (arm64-apple-darwin24.0.0)
    CPU: 14 × Apple M4 Max
    WORD_SIZE: 64
    LLVM: libLLVM-18.1.7 (ORCJIT, apple-m4)
    GC: Built with stock GC
Threads: 10 default, 1 interactive, 10 GC (on 10 virtual cores)
Environment:
    JULIA_NUM_THREADS = auto
    JULIA_PROJECT = @.
    JULIA_LOAD_PATH = @:@stdlib
```


1. Why Program?

CHAPTER AUTHORED BY: ALEC LOUDENBACK

"Humans are allergic to change. They love to say, 'We've always done it this way.' I try to fight that. That's why I have a clock on my wall that runs counterclockwise." - Grace Hopper (1987)

1.1. Chapter Overview

Why should a financial professional learn to program? Because it will improve your capabilities, your enjoyment of the work, and—not incidentally—your career prospects.

1.2. Introduction

The financial sector is undergoing a profound transformation. In an era defined by big data, (pseudo) artificial intelligence, and rapid technological advancement, the traditional boundaries of finance are expanding and blurring. From Wall Street to Main Street, from global investment banks to local credit unions, technology is reshaping how financial services are delivered, how risks are managed, and how decisions are made.

This digital revolution is not just changing the tools we use; it's fundamentally altering the skills required to succeed in finance. In the past, a strong foundation in mathematics, economics, and financial theory was sufficient for most roles in the industry. Today, these skills, while still crucial, are increasingly being augmented (and, in some cases, superseded) by technological proficiency.

Programming skills sit at the center of this shift. In the early computer era in finance, the differentiating skill was the ability to use digital computing, data processing, and calculation engines to automate, analyze, and report on the business. These skills required low-level programming, and the success of those early programs is evident in their legacy: many of them are still running in the 2020s!

At some point, due to regulatory pressures, attempts at organizational efficiencies, or management decision making, the skill of programming became highly specialized and most financial professionals (investment analysts, actuaries, accountants, etc.) became relegated to being “business users”, utilizing either Microsoft Excel or a proprietary third-party software to accomplish their responsibilities. The reasons for this were not totally wrong, even in retrospect.

At some point, with an increasingly complex stack of software sitting between the developer and the hardware, combined with the proliferation of computer security risks, it made some sense that many financial developers were pushed out of the programming trade. Instead, specialized, separate business and IT units were developed. Of course, this led to many inefficiencies and the pendulum is now swinging back the other way.

What's changed that's enabling financial professionals to re-engage with the powerful tools that programming provides? Some reasons include:

1. Why Program?

1. **Code management tools.** GitHub and other version control systems provide best-in-class ways of managing codebase changes and collaboration. Tools exist to scan repositories for leaking secrets, security vulnerabilities, and dependency management.
2. **Increasingly accessible development.** Originally, very few layers of complexity existed between the written code and running it on the mainframe. Over time, drivers, operating systems, networking, dependencies, and compilers made development more complex. Today, languages, libraries, code editors, and deployment tools have smoothed many of these frictions.
3. **Competitive Pressures.** An increasingly commoditized financial product with evermore competition has led to a need to improve efficiency of manufacturing and selling financial products. Having a business developer is a lot more efficient than a business user who needs to get an IT developer to implement something. Further, pressures from outside the financial sector abound: It's easier to teach a tech developer enough to be successful in a finance role than it is to teach a finance professional development skills.
4. **Regulatory and Risk Demands.** Pressures that previously motivated the move to proprietary software for modeling included regulatory reporting, internal risk metrics, and management performance evaluation. However, companies are realizing that their unique products, risk frameworks, preferred management measurements, and employee potential mean that having a bespoke internal model is seen as a key capability. Many regulatory frameworks also encourage the use of a bespoke model, which is a particularly attractive option especially for those who view the given regulatory framework as inappropriately reflecting their own business and risk profile.

Whether you're an investment banker modeling complex derivatives, an actuary calculating insurance risks, a financial planner optimizing client portfolios, or a risk manager stress-testing scenarios, the ability to code is becoming as fundamental as the ability to use a spreadsheet was a generation ago. To remain competitive, adaptable, and effective in the evolving landscape of finance, professionals must embrace programming as a core skill.

Note

One subset of business analysts that *did not* start to migrate away from development as a strategic part of their value were "quants" or quantitative analysts who heavily utilized programming skills to develop unique products, trading strategies, modeling frameworks, and risk engines. This book is not really for that set of people and is instead geared towards the mass of financial professionals who want to get some of the benefits of the tools that the quants have been using for years. Quants may find value here in adapting some of their existing knowledge with the concepts and capabilities that Julia enables.

As we delve into this topic, keep in mind that learning to code is not about replacing traditional financial acumen—it's about augmenting and enhancing it. It's about equipping yourself with the tools to tackle the complex, data-driven challenges of modern finance. In short, it's about future-proofing your career in an industry that is increasingly defined by its ability to innovate and adapt to technological change.

1.2.1. Market Forces

Today, there is a trend toward technological value-creation that is evident across many traditional sectors. Tesla claims that it's a technology company; Amazon is the #1 product retailer because of its vehement focus on internal information sharing¹; airlines are so dependent on their systems that the skies become quieter on the rare occasion that their computers give way. Why are companies involved in *things* (cars, shopping) and *physical services* (flights) more focused on improving their

¹Have you had your Bezos moment? What you can learn from Amazon.

1.3. Why Programming Matters Now

technological operations than insurance companies, *whose very focus is information-based? The market has rewarded those who have prioritized their internal technological solutions.*

Commoditized investing services and challenging yield environments have eroded the comparative advantage of “managing money.” Spread compression and the explosion of consumer-oriented investment services mean that competition now focuses on efficiently managing an asset or policy’s entire lifecycle (digitally), performing more real-time analysis of experience and risk, and handling growing product and regulatory complexity.

These are problems that have technological solutions and are waiting for insurance company adoption.

Companies that treat data like coordinates on a grid (spreadsheets) *will get left behind.* Two main hurdles have prevented technology companies from breaking into insurance and traditional finance:

1. High regulatory barriers to entry, and
2. Difficulty in selling complex insurance products without traditional distribution.

Once those two walls are breached, traditional finance companies without a strong technology core will struggle to keep up. The key to thriving is not just adding “developers” to an organization; it will require **getting domain experts like financial modelers to be an integral part of the technology transformation.**

1.3. Why Programming Matters Now

Programming is becoming as fundamental for financial professionals as spreadsheet skills were a generation ago. Here’s why:

1. **Enhanced Analysis Capabilities:** Programming allows for more complex analyses, handling of larger datasets, and application of advanced statistical and machine learning techniques.
2. **Automation and Efficiency:** Repetitive tasks can be automated, freeing up time for more value-added activities.
3. **Customization:** Bespoke solutions can be developed to address unique business needs, risk frameworks, and regulatory requirements.
4. **Data Handling:** As data volumes grow, programming provides tools to efficiently process, analyze, and derive insights from vast amounts of information.
5. **Integration:** Programming skills enable better integration across different systems and data sources, providing a more holistic view of financial operations.
6. **Competitive Edge:** In an increasingly technology-driven industry, programming skills can be a significant differentiator.

It’s now commonly accepted that to gather insights from your data, you need to know how to code. Modeling and valuation needs, too, are often better suited to customized solutions. Let’s not stop at data science when learning how to solve problems with a computer.

1. Why Program?

1.4. The Spectrum of Programming in Finance

It's important to note that becoming proficient in programming doesn't mean you need to become a full-time software developer. There's a spectrum of programming skills that can benefit financial professionals:

1. **Basic Scripting:** Automating repetitive tasks in Excel or other tools.
2. **Data Analysis:** Using languages like Python or R for statistical analysis and visualization.
3. **Model Building:** Developing financial models or risk assessment tools.
4. **Full-Scale Application Development:** Creating more complex applications for internal use or client-facing solutions.

1.5. Artificial Intelligence (AI)

One tantalizing path to contemplate is avoiding *really* learning how to code. Between artificial intelligence (AI) solutions being developed and low-code offerings, is there really a need to learn the fundamentals of coding? We argue emphatically that there is. Coding is not about mechanically typing out lines in an editor — it is a discipline for thinking clearly about complex problems. The act of translating a financial model into code forces you to make every assumption explicit, every edge case visible, and every dependency traceable. AI can't do that *for* you; it can only do it *instead of* you, and the distinction matters.

This is more than philosophy — it's supported by experimental evidence. In a study by Anthropic, 52 mostly junior software engineers with Python experience were given coding tasks. Half got AI assistance, half didn't (Shen and Tamkin 2026). Afterward, both groups took a quiz without AI. The AI-assisted group scored 17% lower — roughly two letter grades — with the biggest gap in debugging skills. Meanwhile, the AI users completed tasks only about two minutes faster on average, a difference that wasn't statistically significant. In other words: measurable skill loss for negligible speed gain.

The most revealing part is that *how* people used AI mattered enormously. The researchers identified six distinct interaction patterns. Users who engaged in "AI delegation" (copy-pasting without reading) or "iterative debugging" (repeatedly pasting errors back) had the lowest quiz scores. Those who adopted "conceptual inquiry" (asking how concepts work before coding) or "hybrid explanation" (asking AI to generate and explain code) maintained learning outcomes comparable to manual coders. The implication is clear: AI is a powerful *learning accelerant* when used to deepen understanding, but a *skill eroder* when used to bypass it.

This has direct implications for financial modeling teams. Developers who relied on AI encountered fewer errors during the task itself but were less able to identify and resolve issues independently afterward, while those without AI encountered more errors during the task, which appeared to strengthen their understanding. In a field where a missed edge case in a valuation model can move millions of dollars, the ability to debug, reason about, and defend your own code is not optional.

A somewhat sobering implication for agentic AI: the researchers think that while agentic AI might improve productivity on short run tasks, it may reduce the development of critical skills in the long run. If AI is doing the work, users may not be learning the underlying concepts and problem-solving skills that are essential for growth and adaptability in their field.

The current generation of AI is also fundamentally limited in ways that matter for modeling. As Yann LeCun, Meta's Chief AI Scientist, notes, current Large Language Models (LLMs) lack a fundamental grasp of logic and causality. They "cannot reason in any reasonable definition of the term and cannot plan... hierarchically," making them powerful assistants for syntax and boilerplate but unreliable architects for the complex, logical structures required in financial

modeling (Murphy and Criddle 2024). An LLM can generate a Monte Carlo loop, but it cannot tell you whether the model's assumptions are appropriate for the risk you're trying to measure.

The bottom line: AI will play an important and growing role *supporting* modelers — lowering syntactical hurdles (“in VBA I would do it like this, but in Julia how do I do X, Y, or Z?”), generating boilerplate, and providing algorithmic scaffolding. What it will not replace is the core value a modeler brings: skepticism, creative thinking, understanding of company and market dynamics, and the capability to grasp the broader architecture and conceptual aspects of a model. Learning to program well makes you a *better* user of AI tools, not a redundant one.

Julia and AI Code Generation

Counterintuitively, Julia — despite having a far smaller training corpus than Python — consistently performs as well or better in LLM code generation benchmarks. In a study evaluating ChatGPT 4 across 19 programming languages, Julia achieved the highest success rate at 81.5%, while C++ had the lowest at 7.3% (Buscemi 2023). The MultiPL-T project, which fine-tuned open-source Code LLMs on low-resource languages including Julia, found that these models outperformed other open Code LLMs on the MultiPL-E benchmark.

Why does a language with less training data produce better AI-generated code? The leading explanation is *syntactic consistency*. Julia has a uniform standard library API, one canonical way to do most things, and a corpus skewed toward expert-written packages rather than introductory exercises. Chris Rackauckas (MIT, author of DifferentialEquations.jl) observed that LLM-generated Python code for his diffeqpy library had dramatically more errors than the equivalent Julia code — particularly in the same spots where new students struggle: API inconsistencies, multiple ways to express basic operations, and fragmented ecosystems. In Julia, `rand` works the same way whether you’re sampling a scalar, a distribution, or a custom type. That consistency helps humans *and* machines.

The practical takeaway: if you’re going to use AI-assisted coding (and you should), Julia’s design means you’ll spend less time fixing the AI’s output and more time on the modeling problem itself.

1.6. Low Code

A similarly fraught path is using low-code solutions. Low-code is inherently limiting and locks you into a particular vendor’s proprietary ecosystem. If you know enough about what you’re trying to do to state it clearly in plain English, you’re most of the way to programming in a full coding solution (AI can help bridge this gap). As soon as you hit a limitation (“I’d like to use XYZ optimization algorithm at each timestep”), you’re reliant on the vendor to implement that option. Further, you’re outsourcing important inner-workings of the model to someone else and not building that expertise yourself or in-house.

1.7. The 10x Modeler

Increasingly complex business needs will highlight a large productivity difference between a financial modeler who can code and one who can’t—simply because the former can react, create, synthesize, and model faster. From transforming administration extracts to summarizing valuation output to analyzing data in ways spreadsheets simply can’t handle, you can become a “**10x Modeler**”².

²The term ‘10x developer’ originates from the technology sector to describe an engineer considered to be an order of magnitude more productive than their peers. This book adapts the concept to the domain of financial modeling.

1. Why Program?

Note

In the technology sector, a 10x developer is a term for a software engineer who is an order of magnitude more productive, creative, or capable than a typical peer. Here, we extend the notion to developers of financial models.

Working within a vendor's graphical user interface (GUI) makes you a consumer of their modeling tool. Writing your own model in code makes you an architect of the solution. This is the difference between surface-level familiarity and full command over the analysis and concepts involved—with the flexibility to do what your software can't.

Your current software might be able to perform the first layer of analysis, but be at a loss when you want to take it a step further. Tasks like visualizations, sensitivity analysis, summary statistics, stochastic analysis, or process automation, when done programmatically, are often just a few lines of additional code over and above the primary model.

Don't abandon commercial tools prematurely. Complement them with code to break out of constraints and integrate their outputs into broader pipelines. But the ability to supplement and break out of the modeling box has become an increasingly important part of most professionals' work, and this trend is accelerating.

Code-based solutions can also leverage the entire technology sector's progress to solve problems that are *hard* otherwise: scalability, data workflows, integration across functional areas, version control, model change governance, reproducibility, and more.

Thirty to forty years ago, there were no vendor-supplied modeling solutions—you had no choice but to build models internally. That shifted with the advent of vendor-supplied solutions. Today, it's never been better for companies to leverage open and inner source to support custom modeling, risk analysis and monitoring, and reporting workflows.

Note

Open source refers to software whose source code is freely available for anyone to view, modify, and distribute. It promotes collaboration, transparency, and innovation by allowing developers worldwide to contribute to and improve the codebase. Open source projects often benefit from diverse perspectives and rapid development cycles, resulting in robust and widely-adopted solutions.

Inner source applies open source principles within a single organization. It encourages internal collaboration, code sharing, and transparency across different teams or departments. By adopting inner source practices, companies can reduce duplication of effort, improve code quality, and foster a culture of knowledge sharing. This approach can lead to more efficient development processes and better utilization of internal resources.

It is said that you cannot fully conceptualize something unless your language has a word for it. Similar to spoken language, you may find that breaking out of spreadsheet coordinates (and even a dataframe-centric view of the world) reveals different questions to ask and enables innovative ways to solve problems. In this way, you reward your intellect while building more meaningful and relevant models and analysis.

1.8. Risk Governance

Code-based workflows are highly conducive to risk governance frameworks as well. If a modern software project has all of the following benefits, then why not a modern insurance product and

associated processes?

- Access control, reviews, and approvals
- Versioning, environment pinning, and reproducibility
- Automated testing and continuous validation
- Transparent design and documentation
- Fewer manual interventions and lower user error
- Monitoring, logging, and audit trails
- Clear interfaces to other processes (e.g., APIs, schedulers)
- Collaboration at scale via modern dev tooling

These aspects of business processes are what technology companies *excel* at. There is a litany of highly robust, battle-tested tools used in the information services sectors. This book will introduce much of this to the financial professional (specifically Chapter 12, Chapter 21, and Chapter 23).

1.9. Managing and Leading the Transformation

For managers, literacy in modern analytical tooling is a spectrum. Even senior leaders benefit from hands-on familiarity with how models are designed, tested, and deployed.

The skills that make good programmers—decomposing problems, designing abstractions, iterating—are also core leadership skills. They enable you to articulate a target operating model rather than simply manage current constraints. When you’re skilled at pulling apart a problem or process into constituent parts and designing optimal solutions... that’s a core attribute of leadership as well as the most essential skill in programming. This perspective also enables a vision of where the organization *should be* instead of where it is now.

The skillset described herein is as important an aspect of career development as mathematical ability, project collaboration, or financial acumen.

1.10. Outlook

Competitiveness increasingly depends on modernization. This will not come solely from large black-box packages but from domain experts who can encode firm-specific expertise into transparent, testable, and integrated systems—faster and more robustly than the competition.

1. Why Program?

2. Why use Julia?

CHAPTER AUTHORED BY: ALEC LOUDENBACK

We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled. - Bezanson, Karpinski, Shaw, Edelman (the original creators of Julia)

2.1. Chapter Overview

Why Julia? Because it's fast, expressive, and designed for the kind of work financial modelers actually do.

2.2. Introduction

Julia is a relatively new¹, productive, and fast programming language. This is evident in its pragmatic, productivity-focused design choices, pleasant syntax, rich ecosystem, thriving communities, and its ability to be both very general purpose and power cutting-edge computing.

With Julia: math-heavy code looks like math; it's easy to pick up, and quick to prototype. Packages are well-integrated, with excellent visualization libraries and pragmatic design choices.

Julia's popularity continues to grow across many fields, with a growing body of online references, tutorials, videos, and print media to learn from.

Large financial services organizations have already started realizing gains: BlackRock's Aladdin portfolio modeling, the Federal Reserve's economic simulations, and Aviva's Solvency II-compliant modeling². The last of these has a great talk on YouTube by Aviva's Tim Thornham that showcases an on-the-ground view of what difference the right choice of technology and programming language can make. Moving from their vendor-supplied modeling solution was **1000x faster, took 1/10 the amount of code, and was implemented 10x faster**.

The language is great for data science, but also for modeling, ETL, visualizations, package management, machine learning, string manipulation, web-backends, and many other use cases.

Julia is well suited for financial modeling work: easy to read and write and very performant.

¹Python first appeared in 1991. R is an implementation of S, which was created in 1976, though when to place the start of an independent R project varies (1993, 1995, and 2000 are alternate dates). The history of these languages is long and substantial changes have occurred since these dates.

²Aviva Case Study

2. Why use Julia?

💡 Tip

The **two language problem** is a term describing processes and teams that separate “domain expertise” coding from “production” coding. This isn’t always a “problem”, but the “two language problem” describes the scenario where this arises not out of intention but out of the necessity of dealing with limitations of the programming languages used. The most common combination is when the domain experts utilize Python, while the quants or developers write C++. This arises because the productive, high level language hits a barrier in terms of speed, efficiency, and robustness. Then, as a necessary step to achieve the end goals of the business, the domain experts hand off the logic to be re-implemented into the lower level language. Not only does this effectively limit the architecture, it essentially defines a required staffing model which may introduce a lot of cost and redundancy in expertise. Julia solves this to a large extent, allowing for a high level, productive language to be very fast.

A similar, related dichotomy is the **two culture problem** wherein domain experts (e.g. financial analysts) exist in a different sphere from developers. This manifests in many ways, such as restricting the tools that each group is permitted to use (e.g. Excel for domain experts, codebases and Git for developers). This is less of a technical problem and more of a social one. However, Julia is also one of the better languages in this regard, because much of the associated tooling is made as straightforward as possible (e.g. packaging, distribution, workflows, etc.). See Chapter 12 and Chapter 21 through Chapter 24 for more on this.

2.3. Julia and This Book

Julia is introduced only insofar as a basic understanding is necessary to illustrate certain concepts. Julia is ideal here because it’s generally straightforward and concise, letting the presented idea have the spotlight (as opposed to language boilerplate or obtuse keywords). The point of this structure is to introduce a wide variety of computer science concepts to the financial professional, *not* to introduce Julia as a programming language (there are many other resources that do that just fine).

This chapter motivates why we choose Julia for teaching and for work. In Chapter 5, we introduce core language concepts and syntax. After that, the content focuses on illustrating key concepts, with Julia taking a secondary role as backdrop. It’s not until Chapter 21 that Julia regains the spotlight for particulars that matter mainly to heavy Julia users.

2.4. Expressiveness and Syntax

Expressiveness is the *manner in which* and *scope of* ideas and concepts that can be represented in a programming language. **Syntax** refers to how the code *looks* on the screen and its readability.

In a language with high expressiveness and pleasant syntax, you:

- Go from idea in your head to final product faster.
- Encapsulate concepts naturally and write concise functions.
- Compose functions and data naturally.
- Focus on the end-goal instead of fighting the tools.

Expressiveness can be hard to explain, but perhaps two short examples will illustrate.

2.4.1. Example 1: Retention Analysis

This is a really simple example relating Cessions, Policies, and Lifes to do simple retention analysis. Retention is a measure of how much risk an insurance company holds on a policy after its own reinsurance risk transfer (ceded amounts of coverage are called “cessions”).

First, let’s define our data:

```
# Define our data structures
struct Life
    policies
end

struct Policy
    face
    cessions
end

struct Cession
    ceded
end
```

Now to calculate amounts retained. First, let’s say what retention means for a Policy:

```
# define retention
function retained(pol::Policy)
    pol.face - sum(cession.ceded for cession in pol.cessions)
end

retained (generic function with 1 method)
```

And then what retention means for a Life:

```
function retained(l::Life)
    sum(retained(policy) for policy in l.policies)
end

retained (generic function with 2 methods)
```

It’s almost exactly how you’d specify it in English. No joins, no boilerplate, no fiddling with complicated syntax. You express ideas the way you think of them—not as a series of dataframe joins or row/column coordinates on a spreadsheet.

We defined `retained` and adapted it to mean related but different things depending on the context. We didn’t have to define `retained_life(...)` and `retained_pol(...)` because Julia can `dispatch` based on what you give it (this is a more powerful, generalized version of method dispatch commonly used in object-oriented programming; see Chapter 7 for more).

Let’s use the above code in practice.

```
# create two policies with two and one cessions respectively
pol_1 = Policy(1000, [Cession(100), Cession(500)])
pol_2 = Policy(2500, [Cession(1000)])
```

2. Why use Julia?

```
# create a life, which has the two policies
life = Life([pol_1, pol_2])

Life(Policy(Policy(1000, Cession[Cession(100), Cession(500)]), Policy(2500,
    ↴ Cession[Cession(1000)])))

    retained(pol_1)

400

    retained(life)

1900
```

Finally, something called “broadcasting”, which automatically vectorizes any function you write over the given collection(s), no need to write loops or create if statements to handle a single vs repeated case:

```
retained.(life.policies) # retained amount for each policy

2-element Vector{Int64}:
 400
 1500
```

2.4.2. Example 2: Random Sampling

As another motivating example showcasing multiple dispatch, here's random sampling in Julia, R, and Python.

We generate 100 of each:

- Uniform random numbers
- Standard normal random numbers
- Bernoulli random numbers
- Random samples from a given set

Table 2.1.: A comparison of random outcome generation in Julia, R, and Python.

Julia	R	Python
<pre>using Distributions rand(100) rand(Normal(), 100) rand(Bernoulli(0.5), 100) rand(["Preferred", "Standard"], 100)</pre>	<pre>runif(100) rnorm(100) rbinom(100, 1, 0.5) sample(c("Preferred", "Standard"), 100, replace=TRUE)</pre>	<pre>import scipy.stats as sps import numpy as np sps.uniform.rvs(size=100) sps.norm.rvs(size=100) sps.bernoulli.rvs(p=.5, size=100) np.random.choice(["Preferred", "Standard"], size=100)</pre>

2. Why use Julia?

In Julia, rand dispatches on the argument's type: arrays, distributions, and collections all work with the same function, illustrating multiple dispatch. By understanding the different types of things passed to `rand()`, it maintains the same syntax across a variety of different scenarios. We could define `rand(Cession)` and have it generate a random Cession like we used above.

2.5. The Speed

As stated in the journal Nature, "Come for the Syntax, Stay for the Speed".

Earlier we described Aviva's Solvency II compliance modeling, which ran 1000x faster than the prior vendor solution: what does it mean to be 1000x faster at something? It's the difference between something taking 10 seconds instead of 3 hours — or 1 hour instead of 42 days.

With this difference in speed, you would be able to complete existing processes much faster, or extend the analysis further. This speed could allow you to do new things: a stochastic analysis of life-level claims, machine learning with your experience data, or perform much more frequent valuation.

Here's a real example, comparing the runtime to calculate the price of a vanilla European call option using the Black-Scholes-Merton formula, as well as the associated code for each. Here's the mathematical formula we are using:

$$\text{Call}(S_t, t) = N(d_1)S_t - N(d_2)Ke^{-r(T-t)}$$
$$d_1 = \frac{1}{\sigma\sqrt{T-t}} \left[\ln\left(\frac{S_t}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t) \right]$$
$$d_2 = d_1 - \sigma\sqrt{T-t}$$

```
using Distributions

function d1(S,K,τ,r,σ)
    (log(S/K) + (r + σ^2/2) * τ) / (σ * √(τ))
end

function d2(S,K,τ,r,σ)
    d1(S,K,τ,r,σ) - σ * √(τ)
end

function Call(S,K,τ,r,σ)
    N(x) = cdf(Normal(),x)
    d1 = d1(S,K,τ,r,σ)
    d2 = d2(S,K,τ,r,σ)
    return N(d1)*S - N(d2) * K * exp(-r*τ)
end

S,K,τ,σ,r = 300, 250, 1, 0.15, 0.03

Call(S,K,τ,r,σ) # 58.81976813699322

from scipy import stats
import math

def d1(S,K,τ,r,σ):
```

```

    return (math.log(S/K) + (r + sigma**2/2) * tau) / (sigma * math.sqrt(tau))

def d2(S,K,tau,r,sigma):
    return d1(S,K,tau,r,sigma) - sigma * math.sqrt(tau)

def Call(S,K,tau,r,sigma):
    N = lambda x: stats.norm().cdf(x)
    d_1 = d1(S,K,tau,r,sigma)
    d_2 = d2(S,K,tau,r,sigma)
    return N(d_1)*S - N(d_2) * K * math.exp(-r*tau)

S = 300
K = 250
tau = 1
sigma = 0.15
r = 0.03

Call(S,K,tau,r,sigma) # 58.81976813699322

d1<- function(S,K,t,r,sig) {
  ans <- (log(S/K) + (r + sig^2/2)*t) / (sig*sqrt(t))
  return(ans)
}

d2 <- function(S,K,t,r,sig) {
  return(d1(S,K,t,r,sig) - sig*sqrt(t))
}

Call <- function(S,K,t,r,sig) {
  d_1 <- d1(S,K,t,r,sig)
  d_2 <- d2(S,K,t,r,sig)
  return(S*pnorm(d_1) - K*exp(-r*t)*pnorm(d_2))
}
S <- 300
K <- 250
t <- 1
r <- 0.03
sig <- 0.15

Call(S,K,t,r,sig) # 58.81977

```

We find in Table 2.2 that, despite the syntactic similarity, Julia is much faster than the other two. Exact numbers vary by hardware and implementation choices; the key point is that you can write high-level Julia and still get C-like performance.

2. Why use Julia?

Table 2.2.: On our test machine, a scalar BSM pricing function implemented in idiomatic Julia ran in tens of nanoseconds per call after JIT warmup. Comparable R code ran in a few microseconds, and a pure-Python version (without vectorized NumPy) ran orders of magnitude slower. Python's `timeit` library does not calculate the median, so it is not available.

Language	Median (<i>nanoseconds</i>)	Mean (<i>nanoseconds</i>)	Relative Mean
Python	N/A	817000.0	19926.0
R	3649.0	3855.2	92.7
Julia	41.0	41.6	1.0

2.5.1. Development Speed

Speed is not just great for improvement in production processes. During development, it's really helpful too. When building something, the faster feedback loop allows for more productive development. The build, test, fix, iteration cycle goes faster this way.

Admittedly, most workflows don't see a 1000x speedup, but 10x to 1000x is a very common range of speed differences versus R, Python, or MATLAB.

i Note

Sometimes you will see less of a speed difference; R and Python have already circumvented this and written much core code in low-level languages. This is what's called the "two-language" problem where the language productive to write in isn't very fast. For example, more than half of R packages use C/C++/Fortran and core packages in Python like Pandas, PyTorch, NumPy, SciPy, etc. do this too.

Within the bounds of the optimized R/Python libraries, you can leverage this work. Extending it can be difficult. What if you have a custom retention management system running on millions of policies every night? In technical terms, libraries like NumPy are not able to handle custom data types, and instead limit use to pre-built types within the library. In contrast, all types in Julia are effectively equal, even ones that you might create yourself.

Julia packages you are using are almost always written in pure Julia: you can see what's going on, learn from them, or even contribute a package of your own!

2.6. More of Julia's benefits

Julia is easy to write, learn, and be productive in:

- It's free and open-source
 - Very permissive licenses, facilitating the use in commercial environments (same with most packages)
- Large and growing set of available packages
- Write how you like because it's multi-paradigm: vector-izable (R), object-oriented (Python), functional (Lisp), or detail-oriented (C)
- Built-in package manager, documentation, and testing library
- Jupyter Notebook support (it's in the name! **Julia-Python-R**)
- Many small, nice things that add up:

- Unicode characters like α or β
- Nice display of arrays
- Simple anonymous function syntax
- Wide range of text editor support
- First-class support for missing values across the entire language
- Literate programming support (like R-Markdown)
- Built-in Dates package that makes working with dates pleasant
- Ability to directly call and use R and Python code/packages with the PythonCall.jl and RCall packages
- Error messages are helpful and tell you *what line* the error came from, not just the type of error
- Debugger functionality so you can step through your code line by line

For power-users, advanced features are easily accessible: parallel programming, broadcasting, types, interfaces, metaprogramming, and more.

These are some of the things that make Julia one of the world's most loved languages on the StackOverflow Developer Survey.

In addition to the liberal licensing mentioned above, there are professional products from organizations like JuliaHub that provide hands-on support, training, IT governance solutions, behind-the-firewall package management, and deployment/scaling assistance.

2.7. Tradeoffs when Using Julia

2.7.1. Just-In-Time Compilation

Julia is fast because it's compiled, unlike R and Python where (loosely speaking) the computer reads one line at a time. Julia compiles code "just-in-time" (JIT): right before you use a function for the first time, it takes a moment to pre-process the code for the machine³. Subsequent calls don't need re-compilation and are very fast.

A hypothetical example: running 10,000 stochastic projections where Julia needs to precompile but then runs each 10x faster:

- Julia runs in 2 minutes: the first projection takes 1 second to compile and run; the remaining 9,999 projections only take 10 ms.
- Python runs in 17 minutes: 100 ms for each computation.

Typically, compilation is very fast (milliseconds), but in the most complicated cases it can be several seconds. The most common example is the "time-to-first-plot" issue: super-flexible plotting libraries have a lot to precompile. It can take several seconds to display the first plot after starting Julia, but then it's remarkably quick and easy to create an animation of your model results (which requires repeated evaluation of the plot). This is a solvable problem receiving attention from core developers and is improving with new Julia releases.

For users working with a lot of data or complex calculations (like actuaries!), the runtime speedup is worth a few seconds at the start.

³Julia can also precompile code ahead of time to avoid the latency involved with running a function for the first time each time a Julia session is started.

2. Why use Julia?

2.7.2. Static Binaries

Static binaries are self-contained executable programs that run specific code. They compile down to small binaries that accomplish just their pre-programmed tasks. Another use case is creating shared libraries callable from other languages. You can create self-contained artifacts via `PackageCompiler.jl` that bundle the Julia runtime; truly minimal static binaries are an active area of research.

Julia's dynamic nature means it needs to include supporting infrastructure to run general code, similar to how Python code needs the Python runtime bundled.

Development is happening at the language level to allow Julia to compile to a smaller, more static set of features for memory-constrained environments.

2.8. Package Ecosystem

Julia's bundled package manager helps with using packages as dependencies in your project.

For each project, you can track the exact set of dependencies and replicate the code on another machine or at another time. In R or Python, dependency management is notoriously difficult and it's one of the things the Julia creators wanted to fix from the start.

There are thousands of publicly available packages already published. It's also straightforward to share privately, such as proprietary packages hosted internally behind a firewall.

Another powerful aspect of the package ecosystem is that due to the language design, packages can be combined/extended in ways that are difficult for other common languages. This means that Julia packages are often interoperable without any additional coordination.

For example, packages that operate on data tables work together without issue in Julia. In R/Python, many features tend to come bundled in a giant singular package like Python's Pandas, which has Input/Output, date manipulation, plotting, resampling, and more. There's a new Consortium for Python Data API Standards which seeks to harmonize the different packages in Python to make them more consistent (R's Tidyverse plays a similar role in coordinating their subset of the package ecosystem).

In Julia, packages tend to be more plug-and-play. For example, every time you load a CSV you might not want a dataframe (maybe you want a matrix or a plot instead). To load data into a dataframe, the Julia practice is to use both the CSV and DataFrames packages, separating concerns and composing dependencies as necessary. Some users may prefer the Python/R approach of less modular but more all-inclusive packages.

2.9. Tools in Your Toolbox

Looking at other great tools like R and Python, it's difficult to summarize a single reason to switch to Julia—but hopefully we've piqued your interest and can now turn to important concepts.

That said, Julia shouldn't be the only tool in your toolkit. SQL will remain an important way to interact with databases. R and Python aren't going anywhere and will always offer different perspectives!

Being a productive financial professional means being proficient in the language of computers so you can build and implement great things. The choice of tools and paradigms shapes your focus. Productivity is one aspect, expressiveness another, speed one more. Trying out different tools is probably the best way to find what works for you. Julia's multiple-dispatch design lets you write generic models while still hitting performance targets. Combined with GPU acceleration

and packages like CUDA.jl, Optimization.jl, ModelingToolkit.jl, and DifferentialEquations.jl, you can solve large stochastic, PDE, or optimal control problems without switching languages. This is why shops like BlackRock, Aviva, and the Federal Reserve increasingly turn to Julia for portfolio analytics, ALM, and macroeconomic modeling.

2.9.1. Interoperability

Julia isn't an island: PythonCall.jl lets you call Python packages seamlessly; RCall.jl does the same for R. This interoperability allows gradual migration. You can keep mission-critical models written in other ecosystems while gradually migrating performance-sensitive components. Libraries like Arrow.jl, Parquet.jl, and ODBC.jl connect Julia with data warehouses and BI tools. In practice, you can move incrementally: wrap legacy C++ or Python risk engines, pilot new analytics in pure Julia, and phase out old code only when you're satisfied with the results.

2. Why use Julia?

Part II.

Foundations: Effective Financial Modeling

“The map is not the territory.” – Alfred Korzybski (1933)

Welcome to the heart of our exploration of financial modeling. Over the next two chapters, we'll lay out not only the conceptual foundations that shape how and why we model financial phenomena, but also the practical approaches and data management techniques that make these models work in real-world settings. This chapter does not cover the technical foundations of financial modeling—instead we focus on the “how” and “why” of modeling in a professional setting.

First, we'll focus on why financial models matter and what sets them apart from other forms of modeling. This discussion moves from the philosophical—how simplified representations of reality can still provide powerful insights—down to the practical concerns of “small world” assumptions and “large world” complexities. You'll see how models can exist purely for predictive accuracy, even when they don't fully capture underlying causal mechanisms, and why this can be both a strength and a danger. Along the way, we'll unpack what makes modelers effective and explore how curiosity and rigor create more resilient analytical tools.

Then we'll turn to the practicalities: selecting the right modeling approach, handling messy data, and navigating trade-offs like complexity vs. interpretability. You'll learn about building processes that prepare, transform, and feed data into models in ways that keep results transparent and reliable. We'll also explore techniques for balancing simulation, machine learning, and statistical models based on your project's requirements.

Think of these chapters as two sides of the same coin: the first clarifies *why* and *what* we need to model, the second equips you with *how*. By the end, you'll have broad conceptual principles for designing a meaningful model and the hands-on skills to bring that model to life.

3. Elements of Financial Modeling

CHAPTER AUTHORED BY: ALEC LOUDENBACK, YUN-TIEN LEE

"Truth ... is much too complicated to allow anything but approximations" - John von Neumann

3.1. Chapter Overview

What is a financial model? What are they for? We explore key attributes of models, discuss different approaches and their trade-offs, and get into some philosophy about what models are and how they capture important dynamics about the world.

3.2. What is a Model?

A **model** is a simplified representation of reality designed to help us understand, analyze, and make predictions about complex systems. In finance, models distill the intricate web of market behaviors, economic factors, and financial instruments into tractable mathematical and computational components. We build models for a variety of reasons, listed in Table 3.1.

Table 3.1.: The REDCAPE model use framework, from "The Model Thinker" by Scott Page.

Use	Description
Reason	To identify conditions and deduce logical implications.
Explain	To provide (testable) explanations for empirical phenomena.
Design	To choose features of institutions, policies, and rules.
Communicate	To relate knowledge and understandings.
Act	To guide policy choices and strategic actions.
Predict	To make numerical and categorical predictions of future and unknown phenomena.
Explore	To investigate possibilities and hypotheticals.

For example, say we want to simulate returns for stocks in our retirement portfolio. It would be impossible to build a model capturing all the individual people working jobs and making decisions, weather events damaging property, political machinations, etc. Instead, we try to capture certain fundamental characteristics. It's common, for instance, to model equity returns as cumulative pluses and minuses from random movements where those movements have certain theoretical or historical characteristics.

Whether we're using this model of equity returns to estimate available retirement income or replicate an exotic option price, a key aspect of the model is the **assumptions** used therein. For the retirement income scenario, we might *assume* a healthy 8% return on stocks and conclude that such a return will be sufficient to retire at age 53. Alternatively, we may assume that future

3. Elements of Financial Modeling

returns follow a stochastic path with a certain distribution of volatility and drift. These two assumption sets produce **output**—results from our model that must be inspected, questioned, and understood in the context of the “small world” of the model’s mechanistic workings. To be effective practitioners, we must be able to contextualize “small world” results within the “large world” that exists around us.

3.2.1. Small World vs. Large World

The distinction between the modeled “small world” and the real-life “large world” is fundamental to understanding model limitations:

- **Small World:** The constrained, well-defined environment within your model where all rules and relationships are explicitly specified.
- **Large World:** The complex, real-world environment where your model must operate, including factors not captured in your assumptions.

Say our model discounts a fixed set of future cashflows using the current US Treasury rate curve. If I run the model today, then re-run it tomorrow with the same future cashflows and the present value has increased by 5%, I might ask: “why has the result changed so much in such a short period?”

In the “small,” mechanistic world of the model, I can see that discount rates have fallen substantially. The small world answer: inputs changed, producing a mechanical change in output. The large world answer: perhaps the Federal Reserve lowered the Federal Funds Rate to prevent the economy from entering a deflationary recession.

We can’t completely explain the relation between our model and the real world (otherwise we could capture that relationship in the model!). An effective practitioner always tries to look up from the immediate work and take stock of how the world at large *is* or *is not* reflected in the model.

💡 Financial Modeling Pro Tip

When a model output swings materially between runs, first check the Small World mechanics (e.g., input rates changed) before hypothesizing Large World causes (e.g., policy shifts). Then reconcile both views to avoid over- or under-reacting.

3.3. What is a *Financial Model*?

Financial models are those used extensively to ascertain better understanding of complex contracts, perform scenario analysis, and inform market participants’ decisions related to perceived value (and therefore price). It can’t be quantified directly, but it is likely not an exaggeration that many billions of dollars transact each day as a result of decisions made from the output of financial models.

Most financial models can be characterized with a focus on the first or both of:

1. Attempting to project patterns of cashflows or obligations at future timepoints
2. Reducing the projected obligations into a current value (present value)

Examples of this:

- Projecting a retiree’s savings through time (1), and determining how much they should be saving today for their retirement goal (2)

- Projecting the obligation of an exotic option across different potential paths (1), and determining the premium for that option (2)
- Determining a range of outcomes (1) for risk modeling purposes, and translating those outcomes into a risk or capitalization level (2)

Models are sometimes taken a step further, such as transforming the underlying **economic view** into an accounting or regulatory view (such as representing associated debits and credits, capital requirements, or associated intangible, capitalized balances). In other words, many models go further than simply modeling anticipated cashflows by building representations of entities (companies or portfolios) which are associated with the cashflows.

3.3.1. Difference from Data Science

We should also distinguish a financial model from a purely statistical model, where inputs and outputs are often known and the intention is to estimate relationships between variables (e.g., linear regression). That said, a financial model may have statistical components, and many aspects of modeling are shared between the two. In this book, we focus on the practice of financial modeling in the ways that it is distinct from statistics and data science. To delineate the practices:

- **Statistics, or Statistical Modeling** is the practice of applying procedures and probabilistic reasoning to data in order to determine relationships between things or to predict outcomes.
- **Data Science** includes statistical modeling but also incorporates the art and science of good data hygiene, data pipelines and pre-processing, and more programming than a pure statistician usually uses.

Financial Modeling is similar in the goal of modeling complex relationships or making predictions (a modeled price of an asset is simply a prediction of what its value is), however, it differs from data science in a few ways:

- Financial modeling often encodes theory-driven structures and constraints, then calibrates parameters to data, rather than inferring all relationships purely from data.
- Financial modeling generally contains more unique ‘objects’ than a statistical model. The latter may have derived numerical relationships between data, however a financial model would have things like the concept of a company or portfolio, or sets of individually identifiable assets, or distinct actors in a system.
- Financial modeling often involves a lot more simulation and hypothesizing, while data science is focused on drawing conclusions from what data has already been observed.

Nonetheless, there is substantial overlap in practice. For example, the assumptions in a financial model (volatility, economic conditions, etc.) may be derived statistically from observed data. Given the overlap in topics, statistical content is sometimes covered in this book (including from a ground-up view of modern Bayesian approaches in Chapter 14).

3.4. Key Considerations for a Model

When creating a model, whether a data model, a conceptual model, or any other type, certain key considerations are generally important to include to ensure it is effective and useful. Some essential considerations include:

Consideration	Description
Objective	Clearly define what the model aims to achieve.
Boundaries	Specify the limits and constraints of the model to avoid scope creep.

3. Elements of Financial Modeling

Consideration	Description
Variables	Identify and define all variables involved in the model.
Parameters	Include constants or coefficients that influence the variables.
Dependencies	Describe how variables interact with each other.
Relationships	Detail the connections between different components of the model.
Inputs	Specify the data or resources required for the model to function.
Outputs	Define what results or predictions the model produces.
Underlying Assumptions	Document any assumptions made during the model's development to clarify its limitations and validity.
Validation Criteria	Outline how the model's accuracy and reliability are tested.
Performance Metrics	Define the metrics used to evaluate the model's performance.
Scalability	Ensure the model can handle increased data or complexity if needed.
Adaptability	Allow for adjustments or updates as new information or requirements arise.
Documentation	Provide comprehensive documentation explaining how the model works, including algorithms, data sources, and methods.
Transparency	Make the model's workings understandable to stakeholders or users.
User Interface	Design an intuitive interface if the model is interactive.
Ease of Use	Ensure that the model is user-friendly and accessible to its intended audience.
Ethics	Address any ethical concerns related to the model's application or impact.
Regulations	Ensure compliance with relevant laws and regulations.

Including these attributes helps create a robust, reliable, and practical model that effectively serves its intended purpose.

3.5. Predictive versus Explanatory Models

Given a set of inputs, our model generates an output and we're generally interested in its accuracy. This section explores the distinction between predictive models and explanatory models. *The model need not have a realistic mechanism for how the world works.* We may be primarily interested in accurately calculating an output value without the model having any scientific, explanatory power of how different parts of the real-world system interact.

3.5.1. A Historical Example

A historical parallel can be found in the shift from the Ptolemaic to the Copernican model of the solar system. While the Ptolemaic model was predictively accurate for its time, the Copernican model offered a more fundamentally correct explanation of the underlying mechanics¹.

The existing Ptolemaic model used a geocentric view of the solar system in which the planets and sun orbited the Earth in perfect circles with an epicycle used to explain retrograde motion (as seen in Figure 3.1). Retrograde motion is the term used to describe the apparent, temporarily reversed motion of a planet as viewed from Earth when the Earth is overtaking the other planet in orbit around the sun. Despite a fundamentally flawed underlying theory, the geocentric model was accurate enough to match the observational data for the position of the planets in the sky.

¹Prof. Richard Fitzpatrick has excellent coverage of the associated mathematics and implications in "A Modern Almagest": <https://farside.ph.utexas.edu/books/Syntaxis/Almagest/Almagest.html>

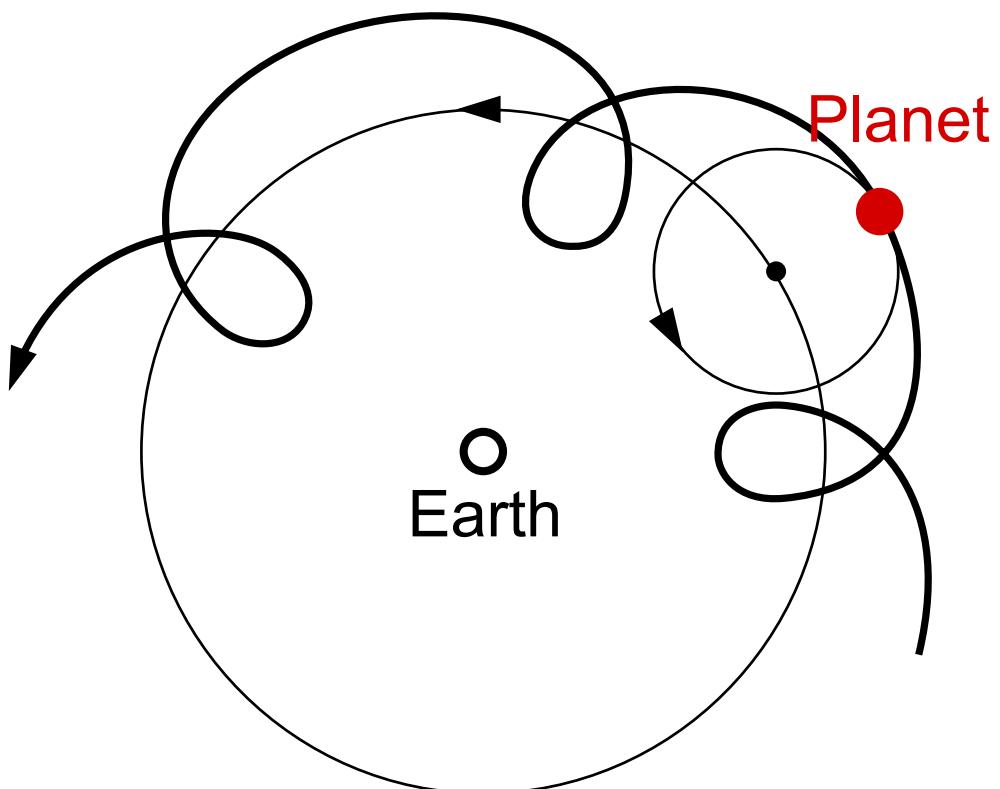


Figure 3.1.: In the Ptolemaic solar model, the retrograde motion of the planets was explained by adding an epicycle to the circular orbit around the earth.

3. Elements of Financial Modeling

Famously, Copernicus came along and said that the sun, not the Earth, should be at the center (a heliocentric model). Earth revolves around the sun! Today, we know this to be a much better description of reality than one in which the Earth arrogantly sits at the center of the universe. However, the model was actually slightly *less* accurate in predicting the apparent position of the planets (to the limits of observational precision at the time)! Why would this be?

First, the Copernican proposal still used perfectly circular orbits with an epicycle adjustment, which we know today to be inaccurate (versus an elliptical orbit consistent with the theory of gravity). *Despite being more scientifically correct, it was still not the complete picture.*

Second, the geocentric model was already very accurate because it was essentially a Taylor series approximation which described, to sufficient observational accuracy, the apparent position of the planet relative to the Earth. *The heliocentric model was effectively a re-parameterization of the orbital approximation.*

Third, we have considered a limited criterion for which we are evaluating the model for accuracy, namely apparent position of the planets. *It's not until we contemplate other observational data that the Copernican model would demonstrate greater modeling accuracy:* apparent brightness of the planets as they undergo retrograde motion and angular relationship of the planets to the sun.

For modelers today, this demonstrates a few things to keep in mind:

1. Predictive models need not have a scientific, causal structure to make accurate predictions.
2. It is difficult to capture the complete scientific inter-relationships of a system and much care and thought need to be given to what aspects are included in our model.
3. We should look at, or seek out, additional data that is related to our model because we may accurately overfit to one outcome while achieving an increasingly poor fit to other related variables.

Striving to better understand the world is a *good thing to do* but trying to include more components into the model is not always going to advance a given goal.

3.5.2. Examples in the Financial Context

3.5.2.1. Home Prices

American home prices exhibit strong seasonality and peak around April each year. We may find that a simple oscillating term captures price variability *better* than imperfectly modeling the true market dynamics: supply and demand curves varying by personal factors (job bonus timing, school calendars), local factors (new homes built, company relocation), and national factors (monetary policy, tax incentives for home-ownership). A simple sinusoidal periodic component could predict this stable pattern well. One could spend months building a more scientific model and not achieve as good a fit, *even though the latter tries to be more conceptually accurate.*

3.5.2.2. Replicating Portfolio

Another financial modeling example: in attempting to value a portfolio of insurance contracts, a **replicating portfolio** of hypothetical assets is sometimes constructed² to match a block of complex insurance contracts. The point is to create a basket of assets that can be valued more quickly (minutes to hours) in response to changing market conditions than it would take to run the actuarial model (hours to days). The basket of assets has no ability to explain *why* the projected cashflows are what they are—but retains strong predictive accuracy.

²See, e.g., SOA Investment Symposium March 2010. *Replicating Portfolios in the Insurance Industry* (Curt Burmeister, Mike Dorsel, Patricia Matson)



Figure 3.2.: House prices (seasonally unadjusted) often peak around April each year. A simple seasonal component can capture much of this variation without modeling full market microstructure, for example $p_t = \alpha + \beta \sin(2\pi t/12 + \phi) + \varepsilon_t$, where t is in months.

3.5.3. Predictive vs Explanatory Models in Practice

Section 32.2.5 describes practical considerations for evaluating models in both predictive and explanatory contexts.

3.6. Types of Models

Different modeling approaches come with their own trade-offs. Common approaches include statistical models such as linear regression and logistic regression; machine learning techniques like decision trees, random forests, and neural networks; probabilistic models including Bayesian networks; simulation models such as Monte Carlo simulation and agent-based models; and empirical models including time series forecasting and econometric models. A key concern with machine learning is overfitting.

3. Elements of Financial Modeling

Model Type	Description	Examples	Trade-offs	Assumptions
Statistical Models	Use mathematical relationships to describe data.	Linear regression, logistic regression	Simplicity vs. Accuracy: Often simpler and more interpretable but may not capture complex relationships.	Typically rely on assumptions like linearity and normality that may not always hold true.
Machine Learning Models	Learn patterns from data using algorithms.	Decision trees, random forests, neural networks	Complexity vs. Interpretability: Capture complex patterns but are often less interpretable. Overfitting: Risk of overfitting, requiring careful validation and tuning. Data Requirements: Require large datasets.	Performance can degrade with limited or noisy data.
Probabilistic Models	Use probability distributions to model uncertainty and relationships.	Bayesian networks, probabilistic graphical models	Flexibility vs. Computational Complexity: Handle uncertainty and complex relationships but require sophisticated computations.	May require assumptions about the nature of probability distributions and dependencies.
Simulation Models	Use computational models to simulate complex systems or scenarios.	Monte Carlo simulations, agent-based models	Accuracy vs. Computational Expense: Can be computationally expensive and time-consuming. Detail vs. Generalization: High-fidelity simulations may be overkill for simpler problems.	May require simplifications or assumptions for computational feasibility.
Theoretical Models	Based on theoretical principles to explain phenomena.	Economic models, physical models	Precision vs. Practicality: Provide foundational understanding but may rely on idealizations. Applicability: Highly accurate in specific contexts but less so in broader situations.	Often rely on idealizations or simplifications.
Empirical Models	Use historical data to predict future outcomes.	Time series forecasting, econometric models	Data Dependence vs. Predictive Power: Rely heavily on historical data and may not perform well if patterns change. Context Sensitivity: Accurate for specific data but may not generalize well.	Performance is heavily dependent on the quality and relevance of historical data.

Model Type	Description	Examples	Trade-offs	Assumptions
Hybrid Models	Combine different modeling approaches to leverage their strengths.	Combining statistical and machine learning approaches	Complexity vs. Versatility: Aim to leverage strengths of different approaches but can be complex to design. Integration Challenges: May present challenges in integration and consistency.	May inherit assumptions from combined models.

3. Elements of Financial Modeling

Summary of Common Trade-offs:

- Complexity vs. Simplicity: More complex models can capture more nuanced details but are harder to understand and manage.
- Accuracy vs. Interpretability: High-accuracy models may be less interpretable, making it harder to understand their decision-making process.
- Data Requirements: Some models require large amounts of data or very specific types of data, which can be a limitation in practice.
- Computational Resources: More sophisticated models or simulations can require significant computational power, which may not always be feasible.

Understanding these trade-offs helps select the most appropriate modeling approach for your specific problem and available resources.

4. The Practice of Financial Modeling

CHAPTER AUTHORED BY: ALEC LOUDENBACK, YUN-TIEN LEE

*"In theory there is no difference between theory and practice. In practice there is." – Yogi Berra
(often attributed)*

4.1. Chapter Overview

Having covered what models are and what they accomplish, we turn to the *craft* of modeling: what distinguishes a good model from a bad one, and what makes an astute practitioner. We'll also cover some "nuts and bolts" topics like data handling and governance practices.

4.2. What makes a good model?

The answer is: *it depends*.

4.2.1. Achieving original purpose

A model is built for a specific set of reasons and therefore we must evaluate a model in terms of achieving that goal. We should not critique a model for falling short in ways it was never intended to be used for.

Consider a model created for scenario analysis to value all assets in a portfolio to within half a percent of a more accurate, but much more computationally expensive model. That's a perfectly good model—for its intended use. But if someone later tries to add a never-before-seen asset class or repurpose it to order trades (a use case requiring much higher accuracy), they're extending beyond the original design scope and shouldn't be surprised when predictive accuracy suffers.

We've seen this pattern play out many times: a quick-and-dirty prototype built for an analyst's personal use gradually gets promoted to "the production model" without anyone rethinking the original design decisions. The model doesn't get worse—it just gets asked to do things it was never built for.

4.2.2. Usability

How easy is it for someone to use? Does it require pages of documentation, weeks of specialized training, and an on-call help desk? All else equal, the heavier the support and training required, the lower the model's usability. That said, you may sometimes want to create a highly capable, complex model that requires significant experience and expertise. Think of the cockpit of a small Cessna versus a fighter jet: the former is simpler and takes less training to master, but is also more limited.

4. The Practice of Financial Modeling

Figure 4.1 illustrates this concept and shows that if your goal is very high capability that you may need to expect to develop training materials and support the more complex model. On this view, a better model is one that is able to have a shorter amount of time and experience to achieve the same level of capability.

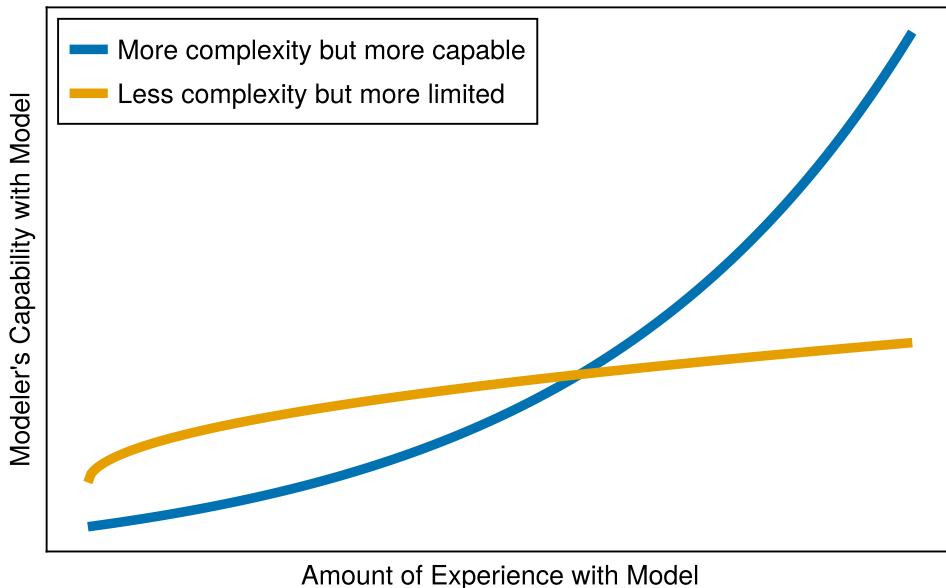


Figure 4.1.: Tradeoff between complexity and capability

4.2.3. Performance

Financial models are generally not used for their awe-inspiring beauty—users are results-oriented. All else equal, faster models are also better models. Beyond direct computational costs like server runtime, shorter model runtime means you can iterate faster, test new ideas on the fly, and stay focused on the problem at hand.

Many readers may be familiar with the cadence of (1) try running the model overnight, (2) see results failed in the morning, (3) spend the day developing, (4) repeat step 1. It is preferred if this cycle can be measured in minutes instead of hours or days.

Of course, requirements must be considered here too: needs for high frequency trading, daily portfolio rebalancing, and quarterly financial reporting models have different requirements when it comes to performance.

4.2.4. Separation of Model Logic and Data

When data is intertwined with business logic it can be difficult to understand, maintain, or adapt a model. Spreadsheets are a common example of where data exists commingled with business logic. An alternative which separates data sources from the computations provides for better model service in the future.

4.2.5. Organization of Model Components and Architecture

If model components or data inputs are spread out in a disorganized way, it can lead to usability and maintenance issues. As an example, oftentimes it's incredibly difficult to ascertain a model's operation if inputs are spread out across locations on many spreadsheet tabs. Or if related calculations are performed in multiple locations, or if it's not clear where the line is drawn between calculations performed in the worksheets or in macros.

If logical components or related data are broken out into discrete parts of a model, it becomes easier to reason about model behavior or make modifications. Compartmentalization is an important principle which allows a larger model to remain composed of simpler components where the whole model is greater than the sum of the pieces.

4.2.6. Abstraction of Modeled Systems

At different times we are interested in different levels on the **ladder of abstraction**: sometimes we are interested in the small details, but other times we are interested in understanding the behavior of systems at a higher level.

Say we are an insurance company with a portfolio of fixed income assets supporting long term insurance liabilities. We might delineate different levels of abstraction like so:

Table 4.1.: An example of the different levels of abstraction when thinking about modeling an insurance company's assets and liabilities.

Item	
More Abstract	Sensitivity of an entire company's solvency position
	Sensitivity of a portfolio of assets
	Behavior over time of an individual contract
More granular	Mechanics of an individual bond or insurance policy

At different times, we are often interested in different aspects of a problem. In general, you start to be able to obtain more insights and a greater understanding of the system when you move up the ladder of abstraction. Sometimes a problem can only be unwound when you move down the ladder and focus on the details in an analytical take-apart-the-pieces way of thinking. But often, the more abstracted view is more useful for understanding the system and making decisions about it.

In fact, a lot of designing a model is essentially trying to figure out where to put the right abstractions. What is the right level of detail to model this in and what is the right level of detail to expose to other systems?

Let us also distinguish between **vertical abstraction**, as described above, and **horizontal abstraction** which will refer to encapsulating different properties, or mechanics of components of the model at the same level of vertical abstraction. For example, both asset and liability mechanics sit at the most granular level in Table 4.1, but it may make sense in our model to separate the data and behavior from each other. If we were to do that, that would be an example of creating horizontal abstraction in service of our overall modeling goals.

This book will introduce powerful, programmatic ways to handle this through things like packages, modules, namespaces, and functions.

4.3. What makes a good modeler?

A model is nothing without its operator, and a skilled practitioner is worth their weight in gold. What elements separate a good modeler from a mediocre modeler?

4.3.1. Domain Expertise

An expert knowledgeable in relevant domains is crucial. Imagine if someone said, “Let’s emulate an architect by having a construction worker and an artist work together.” It’s all too common for businesses to pair a business expert with an IT person and hope for the finished product to be as good as one crafted by someone skilled in all the right domains.

Unfortunately, this means that there’s generally no easy way out of learning enough about finance, actuarial science, computers, and/or programming in order to be an effective modeler.

Also, a word of warning for the financial analysts out there: the computer scientists may find it easier to learn applied financial modeling than the other way around since the tools, techniques, and language of problem solving is already a more general and flexible skill-set. There are more technologists starting banks than there are financiers starting technology companies.

4.3.2. Model Theory

An essential part of financial modeling involves building up the modeler’s expertise. Then, we must characterize that knowledge more explicitly.

The modeler’s knowledge should be regarded as a theory, in the same sense as Ryle’s “Concept of Mind” (Ryle 1949).

A person with **Model Theory** in this sense could be described as one who:

- knows how to operate the model in a wide variety of use cases
- is able to envision the implementation of novel features and how those features relate to existing architecture of the model
- explains, justifies, and answers to queries about the model and its results¹.
- is keenly aware of the small world versus large world distinction (Section 3.2.1)

A financial model is rarely left in a final state. Regulatory changes, additional mechanics, sensitivity testing, market dynamics, new products, and new systems to interact with force a model to undergo change and development through its entire life. Like a living thing, it must have nurturing caregivers. This metaphor sounds extended, but Naur’s point is that unless the model also lives in the heads of its developers, it cannot successfully be maintained through time:

“The conclusion seems inescapable that at least with certain kinds of large programs, the continued adaptation, modification, and correction of errors in them, is essentially dependent on a certain kind of knowledge possessed by a group of programmers who are closely and continuously connected with them.” - Peter Naur, Programming as Theory Building, page 395.

Assume that we need to adapt the model to fit a new product. One possessing a high degree of model theory includes:

- the ability to describe the trade-offs between alternate approaches that would accomplish the desired change

¹The idea of “model theory” is adapted from Peter Naur’s 1985 essay, “Programming as Theory Building”.

- relate the proposed change to the design of the current system and any challenges that will arise as a result of prior design decisions
- provide a quantitative estimation for the impact the change will have: runtime, risk metrics, valuation changes, etc.
- the ability to analogize system behavior to themselves and to others
- Describe key limitations that the model has and where it is most divorced from the reality it seeks to represent.

Abstractions and analogies of the system are a critical aspect of model theory, as the human mind cannot retain perfectly precise detail about how the system works in each sub-component. The ability to, at some times, collapse and compartmentalize parts of the model to limit the mental overload while at others recall important implementation details requires training - and is enhanced by learning concepts like those which will be covered in this book.

An example of how the right abstractions (and language describing those abstractions) can be helpful in simplifying the mental load:

Instead of:

The valuation process starts by reading an extract into three tabs of the spreadsheet. A macro loops through the list of policies on the first tab and in column C it gives the name of the applicable statutory valuation ruleset. The ruleset is defined as the combination of (1) the logic in the macro in the "Valuation" VBA module with, (2) the underlying rate tables from the tabs named XXX to ZZZ, along with (3) the additional policy level detail on the second tab. The valuation projection is then run with the current policy values taken from the third tab of the spreadsheet and the resulting reserve (equal to the actuarial present value of claims) is saved and recorded in column J of the first tab. Finally, a pivot table is used to sum up the reserves by different groups.

We could instead design the process so that the following could be said instead:

Policy extracts are parsed into a Policy datatype which contains a subtype ValuationKind indicating the applicable statutory ruleset to apply. From there, we map the valuation function over the set of Policies and perform an additive reduce to determine the total reserve.

There are terminologies and concepts in the second example which we will develop over the course of this section of the book - we don't want to dwell on the details right now. However, we do want to emphasize that the process itself being able to be condensed down to descriptions that are much more meaningful to the understanding of the model is a key differentiator for a code-based model instead of spreadsheets. It is not exaggerating that we could develop a handful of compartmentalized logics such that our primary valuation process described above could look like this in real code:

```
policies = parse(Policy,CSV.File("extract.csv"))
reserve = mapreduce(value,+ ,policies)
```

We've abstracted the mechanistic workings of the model into concise and meaningful symbols that not only perform the desired calculations but also make it obvious to an informed but unfamiliar reader what it's doing.

`parse` , `mapreduce` , `+` , `value` , `Policy` are all imbued with meaning - the first three would be understood by any computer scientist by the nature of their training (and is training that this book covers). The last two are unique to our model and have "real world" meaning that our domain expert modeler would understand which analogizes very directly to the way we would suggest implementing the details of `value` or `Policy`. The benefit of this, again, is to provide tools and concepts which let us more easily develop model theory.

4. The Practice of Financial Modeling

4.3.3. Curiosity

No model, no matter how sophisticated, ever delivers a “final” answer. If anything, a good financial model sparks as many new questions as it answers.

Take the gnawing feeling you get when a model’s output seems “off” but you can’t quite put your finger on why. The untrained eye might chalk it up to randomness or let it slide, but genuine curiosity won’t settle for a hand-wavy excuse. That itch to resolve every weird edge case or apparent contradiction—to ask “what if?” and “why not?”—is what propels a practitioner beyond rote calculation into actual discovery.

Here’s an example from practice: we once inherited a model that produced a small negative value in a particular reserve calculation every 12 months, right around year-end. Everyone had been ignoring it for years because it was tiny and “probably just a rounding thing.” But someone finally got annoyed enough to dig in. Turned out there was an off-by-one error in how the model handled December versus January transitions—and fixing it revealed a larger issue with how certain liabilities were being recognized across calendar years. The “trivial” anomaly pointed to a real problem.

Curiosity in practice looks like:

- If two approaches give wildly different answers for the same scenario, don’t sweep that under the rug. Dig until you’ve either found the bug or learned a new subtlety about your assumptions.
- If changing an input slightly produces an outsized effect somewhere else, figure out the feedback mechanism causing it. That nonlinearity is telling you something.
- Challenge what “everybody knows.” You’d be surprised how many standard assumptions in financial modeling are half-remembered lore rather than principled choices. Ask: where did this formula actually come from? Does it still apply?
- When the model spits out something bizarre, treat it like an opportunity rather than a headache. Sometimes the oddball result teaches you more about the system than any routine validation could.

The best modelers I’ve worked with aren’t necessarily the flashiest coders or the most fluent in finance. They’re just relentless in their quest to not leave loose ends untied.

4.3.4. Rigor

If curiosity is the fuel, rigor is the steering wheel. All of that wandering through the thickets of “why?” needs a reliable process to keep from becoming noise or hand-waving. Rigor is what separates “I think it works” from “Here’s why it works, and here are its limits.”

When developing a model, assumptions and parameters need to be explicit, the methodology should align with established theory (or consciously depart from it for good reason), and you should think carefully about how the model will actually be used. Professional actuarial societies, for example, maintain a long list of Actuarial Standards of Practice (“ASOPs”), some of which apply directly to modeling and data handling. But regardless of whether formal standards apply to your situation, the underlying principles are worth internalizing.

Document your thinking as you go. Write it out, whether in a code comment, a README, or your own notebook. If you can’t explain your logic and your parameters, you probably don’t understand them as well as you think. We’ve all had the experience of returning to a model six months later and staring at a formula wondering “why did I do it this way?” Documentation is a gift to your future self.

Demand evidence for your choices. Don’t just trust your gut or yesterday’s industry standard. Check your results against reality, not just an assumed “right answer.” This means test cases,

sensitivity checks, and “could we break this?” scenarios. If you’re using a Black-Scholes model, can you reproduce known option prices? If you’re projecting cash flows, do the first few periods match what actually happened?

Don’t hide the warts. Make uncertainty visible, not hypothetical. Annotate what’s based on thin data versus what’s on solid ground. Rigor means being honest about what you don’t know—or what the model simply can’t say. A model that clearly states “this parameter is estimated from only 18 months of data” is more trustworthy than one that presents everything with false precision.

Lean on first principles when it matters. Oftentimes there’s a ‘simpler’ way to model something—a heuristic that says this complex transaction “works like exotic option ABC.” But making explicit all components of an interaction can be illuminating. Model out each leg of a transaction for clarity and confirmation of your understanding, even if you later simplify for production use.

A bad model can be worse than no model at all. It gives false confidence, and people make decisions based on that false confidence. Through rigorous effort, you establish a minimum standard of quality that protects against the worst outcomes.

4.3.5. Clarity

A model is only as useful as it is understandable. This applies both to the model itself and to how you communicate about it.

Consider the term “duration.” To a fixed income analyst, it means interest rate sensitivity. To a project manager, it means how long something takes. To an actuary, it might mean policy duration—how long a contract has been in force. If you’re writing documentation or presenting results and you use “duration” without defining it, you’re inviting confusion. Either define your terms up front or pick less ambiguous words.

The same principle applies to assumptions. It’s not enough to report that a portfolio is worth \$X million. What discount rate did you use? What prepayment assumptions? What credit spreads? Spell out what you left in, what you left out, and why. The philosophy and scaffolding matter as much as the final number.

Visual communication helps enormously. A simple diagram showing how data flows through a model, or a chart comparing scenarios, often communicates more than paragraphs of text. When explaining a model to stakeholders, we’ve found that a single well-designed visual can short-circuit twenty minutes of confusion.

Adjust your explanations for your audience. Developers want to know about data structures and edge cases. Business stakeholders want to know what the number means and what decisions it supports. End users want to know which buttons to push. The same model requires different explanations for each group.

Finally, review your documentation periodically. Models evolve, and documentation gets stale. Ask yourself: if I woke up with amnesia, would the next steps seem obvious? Clarity is about making your future self—and your colleagues—thankful, not furious, that you were ever given keyboard access.

4.3.6. Humility

The world is complicated in ways we can sometimes describe but never fully anticipate. A humble modeler tries to understand what the model can and cannot claim—and communicates those limitations in good faith. “We have a lot of data for low-rate environments, but rapidly rising rate environments haven’t been observed in this dataset” is the kind of caveat that should accompany results, not get buried in footnotes.

4. The Practice of Financial Modeling

There's a useful distinction between two kinds of uncertainty:

Irreducible uncertainty (also called aleatoric uncertainty) refers to the inherent randomness in a system. No amount of additional data or better models will eliminate it. Future market fluctuations, individual policyholder behavior, natural disasters—these are genuinely unpredictable. The best we can do is characterize the range of possibilities.

Reducible uncertainty (epistemic uncertainty) stems from a lack of knowledge. In principle, it can be reduced through more data, better measurement, or improved models. Parameter estimation errors fall into this category: if you had more historical observations, you could pin down that mortality rate more precisely. Model specification errors are similar—maybe you've left out an important variable, and including it would improve predictions.

Table 4.2 describes these distinctions in more detail. You don't need to enumerate every type of uncertainty for every model, but knowing your enemy is the first step in fighting it.

The practical implication is that a humble modeler distinguishes between “we're uncertain because the world is random” and “we're uncertain because we don't have enough data.” The first can't be fixed; the second might be. Communicating this distinction to stakeholders avoids overconfidence in model predictions and keeps everyone open to new information. It also builds trust: people respect a modeler who says “here's what we know, here's what we don't, and here's why” far more than one who presents every output as gospel.

Table 4.2.: In attempting to model an uncertain world, we can be even more granular and specific in discussing sources of that uncertainty. This table summarizes commonly noted kinds of uncertainty that arise, and whether we can reduce the uncertainty by doing better (more data, better data, better models, etc.) or not.

Type of Uncertainty	Key Characteristics	Reducibility	Example
Aleatory (Process) Uncertainty	<ul style="list-style-type: none"> - Inherent randomness (aka “irreducible uncertainty”) - Cannot be eliminated, even with perfect knowledge 	Irreducible	Rolling dice or coin flips; outcome is inherently uncertain despite full knowledge of initial state
Epistemic (Parameter) Uncertainty	<ul style="list-style-type: none"> - Due to limited data/knowledge (aka “reducible uncertainty”) - Imperfect information or model parameters - Uncertainty about the correct model or framework - Often considered a special subset of epistemic uncertainty - “Unknown unknowns” - Probability distributions themselves are not well-defined or are fundamentally unquantifiable 	Reducible (more data / improved modeling)	Uncertainty in a model’s parameters (e.g., climate sensitivity) that can be refined with more research
Model Structure Uncertainty	<ul style="list-style-type: none"> - Systematic biases or random errors in measurement - Uncertainty in implementation/execution - Human error, mechanical failure, or miscommunication in processes 	Partially reducible (better theory/model selection)	Linear vs. nonlinear models in complex systems; risk of omitting key variables or mis-specified dynamics
Deep (Knightian) Uncertainty	<ul style="list-style-type: none"> - “Unknown unknowns” - Probability distributions themselves are not well-defined or are fundamentally unquantifiable 	Not quantifiable (cannot assign probabilities)	Impact of radically new technology on society
Measurement Uncertainty	<ul style="list-style-type: none"> - Errors in data collection or instrument readings - Systematic biases or random errors in measurement 	Partially reducible (improved measurement methods)	Instrument precision limits in experiments; calibration errors in sensor data
Operational Uncertainty	<ul style="list-style-type: none"> - Uncertainty in implementation/execution - Human error, mechanical failure, or miscommunication in processes 	Partially reducible (better training/processes)	Surgical errors, system failures, or incorrect handling of a financial trade order

4.3.7. Architecture

Any sufficiently complex project benefits from architectural thinking. Think of your model like a house: if you don't plan the plumbing, you'll have a mess down the line.

One of the most important architectural principles is separating data from logic. The model itself should not embed substantial data—no hard-coded rate tables, no discount curves pasted into cells, no assumption sets buried in VBA. Instead, dynamically load data from appropriate data stores and leave the “model” as the implementation of datatypes and algorithms (the “business logic”). This makes it possible to run the same model against different datasets, to audit what inputs produced what outputs, and to update assumptions without touching the model code.

Modular design means breaking complex models into reusable, independent components. A valuation routine shouldn't also handle data parsing and report generation. Each piece should do one thing well. When something breaks (and something always breaks), modular design lets you isolate the problem.

Stable interfaces matter when components need to communicate. If your asset model passes data to your liability model, define clearly what that data looks like. When you change the internals of one component, the other shouldn't need to know—as long as the interface contract is preserved.

Version control is non-negotiable for any serious model. We'll cover this later in the book, but the ability to track changes, revert mistakes, and understand the history of a model pays dividends every time something goes wrong (and something always goes wrong).

Performance becomes important as models scale. The difference between a model that runs in 30 seconds and one that runs in 3 hours is the difference between interactive exploration and overnight batch jobs. We'll discuss data structures and algorithms that help, but the architectural decisions you make early—what to precompute, what to cache, what to parallelize—often determine whether performance optimization is even possible later.

Don't underestimate the value of a well-organized model: it's how you scale from small prototypes to systems you can trust in production.

4.3.8. Planning

When tackling a large modeling problem, it helps to think through the project before writing code. The temptation to “just start building” is strong, especially when you're excited about the technical challenge. But time invested at the planning stage almost always pays dividends.

Start with objectives. What questions does this model need to answer? Who will use the outputs, and for what decisions? A model built for quarterly regulatory reporting has different requirements than one built for real-time trading decisions. If you don't know the purpose clearly, you'll make architectural choices you'll regret later.

Define the scope explicitly. What's in and what's out? It's easy for a project to expand indefinitely as stakeholders think of additional features. Write down the boundaries. “This model covers fixed income assets only. Equities are out of scope for version 1.” That sentence, agreed to early, saves arguments later.

Assess your data situation early. What data do you need? Where does it come from? How clean is it? We've seen projects derailed because someone assumed a data feed existed that didn't, or assumed a field contained one thing when it contained another. Get eyes on actual data samples before committing to a design.

Choose your methodology consciously. Is this a Monte Carlo simulation? A closed-form approximation? A machine learning model? The choice should follow from the problem requirements and the available data, not from what's fashionable or what the team knows best. Different

methodologies have different data requirements, runtime characteristics, and interpretability properties.

Identify your stakeholders and get them involved early. Nothing derails a project faster than finishing it and discovering the business wanted something different. Regular check-ins prevent this. They also surface changing requirements before you've built too much to pivot.

Plan for validation and testing. How will you know if the model is working correctly? What test cases will you use? What benchmarks will you compare against? If you can't answer these questions at the planning stage, you probably don't understand the problem well enough yet.

Think about maintenance from the start. Who owns this model after it's built? How will assumptions be updated? What happens when the regulatory environment changes? Models are rarely "done"—they require ongoing care. If you don't plan for maintenance, you'll end up with an orphan that nobody understands and everybody's afraid to touch.

It's easier to make changes to a well-planned project halfway through, because the necessary accommodations are more clearly defined. Without a plan, changes cascade unpredictably, and you end up rewriting more than you expected.

4.3.9. Essential Tools and Skills

An experienced modeler has a mental toolbox with many different approaches to draw on. Some problems call for a back-of-the-envelope calculation; others require a full Monte Carlo simulation. Some questions are best answered with a statistical model; others with a simple lookup table. The ability to recognize which tool fits which problem—and to switch approaches when the first one isn't working—is what separates a craftsman from someone who only knows how to use a hammer.

This book will introduce many of these approaches. Table 4.3 lists some of the categories, covering both technical skills and the softer skills that matter in practice.

Table 4.3.: A variety of skills have their place in the proficient financial modeler's toolbelt.

Category	Examples
Diverse Modeling Techniques	<ul style="list-style-type: none"> • Statistical methods (e.g. regression, time series analysis, machine learning) • Optimization techniques (e.g. linear, non-linear, black-box) • Simulation methods (e.g. Monte-Carlo, agent-based, seriatim)
Software Proficiency	<ul style="list-style-type: none"> • Programming languages • Database and data handling • Proprietary tools (e.g. Bloomberg)
Financial Theory	<ul style="list-style-type: none"> • Asset pricing • Portfolio theory • Risk Management frameworks
Quantitative techniques	<ul style="list-style-type: none"> • Numerical methods and algorithms • Bayesian inference • Stochastic calculus
Soft Skills	<ul style="list-style-type: none"> • Verbal and written communication • Stakeholder engagement • Project Management

4.4. Feeding The Model

The lifeblood of the model is its data. In practice, a model's fate is often sealed not by the sophistication of its algorithms, but by the quality of the data it consumes. We've seen beautifully architected models produce nonsense because someone fed them stale prices, or because a data field meant one thing in one system and something slightly different in another. Even the most elegant model is helpless in the face of bad inputs.

4.4.1. “Garbage In, Garbage Out”

Every experienced modeler has a story where a subtle data quirk led to a dramatic miscalculation—a column header shifted by one, a stale price feed, or a single outlier that quietly cascaded into a million-dollar mistake. The lesson: treat the data with every bit as much skepticism (and care) as you give the model itself.

i Example: The JPMorgan ‘London Whale’

In 2012, JPMorgan Chase suffered over \$6 billion in losses, partly due to errors in a Value-at-Risk (VaR) model. The model relied on data being manually copied and pasted into spreadsheets, a process that introduced errors. Furthermore, a key metric was calculated by taking the sum of two numbers instead of their average. This seemingly small data handling error magnified the model’s inaccuracy, demonstrating that even the most sophisticated institutions are vulnerable to the ‘Garbage In, Garbage Out’ principle.

4.4.2. A Modeler’s Data Instincts

Rather than thinking of data handling as a rigid checklist, approach it as a series of habits and questions.

Know your sources. Where did this data come from? Who collected it, and how? Is it raw, or has someone already “cleaned” it in ways you need to understand (or undo)? We once inherited a dataset where someone had helpfully replaced all the missing values with zeros—which made it impossible to distinguish “no data” from “actually zero.” Data provenance is not a formality; it’s the first step in understanding what can go wrong.

Trust, but verify. Never take a dataset at face value, even if it comes from a trusted system. Run summary statistics. Plot the distributions. Check for the bizarre and the mundane: are dates reasonable, units consistent, and identifiers unique? A quick histogram can show you if someone entered dollars when they should have entered millions of dollars.

Expect messiness. Real-world data is rarely pristine. Missing values, odd encodings, duplicated rows, and outliers are the norm, not the exception. The best modelers are part detective, part janitor: they track down wonky values, document their triage decisions, and know when to escalate a data quality concern upstream rather than quietly “fixing” it themselves.

Feature engineering is judgment, not magic. Choosing which fields to keep, combine, or discard is where domain expertise shines. Sometimes a new ratio or flag column, born from your understanding of the business, makes all the difference. But beware of “kitchen sink” modeling—too many features can obscure, rather than reveal, the truth. If you can’t explain why a feature should matter, think twice before including it.

Be wary of temporal traps. Mixing data from different time periods, or accidentally leaking future information into a model (a classic error), can invalidate results without any warning

sign. If you’re building a model to predict defaults, and you accidentally include a field that was populated *after* the default occurred, your model will look great in backtesting and fail completely in production. When in doubt, plot your data against time and look for jumps, gaps, or trends that defy explanation.

Keep data and logic separate. As mentioned earlier: don’t hard-code data into the model. Keep sources external, interfaces clean, and ingest paths well documented. If someone wants to rerun last year’s scenario, they shouldn’t have to guess which tab or variable held the original rates.

4.4.3. Data Is Never “Done”

Data handling is not a one-time hurdle to clear and move on from. Markets move, data feeds change, formats drift, and the vendor you relied on for reference data gets acquired and changes their API. A model that worked perfectly last quarter can silently break this quarter because an upstream system started encoding dates differently.

Build routines to check for “data drift”—changes in the statistical properties of your inputs that might indicate a problem. Have a plan for periodic validations and refreshes. Some practical tips:

- Maintain a simple data log or data dictionary—even if informal—so others can trace what each field means and where it came from.
- Automate the boring parts: validation scripts, input checks, and sanity tests pay off a hundredfold.
- Version your datasets, just as you do your code. Nothing is more frustrating than trying to reproduce a result only to discover “the input file changed.” See Section 12.5.4

Data is unruly, idiosyncratic, and absolutely central to every model’s fate. Treat it as a first-class concern, not an afterthought, and your models will be far sturdier for it. As a methodical guide, Table 4.4 lists key steps to follow when bringing data into the model.

Table 4.4.: Typical Steps in the Data-to-Model Process.

Step	Key Actions	Purpose / Notes
Data Collection	Identify sources Acquire data (e.g., APIs, databases, scraping)	Ensures data is relevant, reliable, and timely
Data Exploration & Understanding	Summary statistics Visualization Data profiling	Uncovers initial insights, errors, distributions, and relationships
Data Cleaning	Handle missing values Detect/treat outliers Data transformation/formatting	Improves data quality, reduces noise and bias
Data Preprocessing	Scale/normalize features Encode categorical variables Augment data (if needed) with other datasets	Prepare data so it fits the format and requirements of the model
Feature Engineering	Select important features Create new features (e.g., ratios, aggregates)	Enhance or create new variables that improve model performance

4. The Practice of Financial Modeling

Step	Key Actions	Purpose / Notes
Data Splitting	Divide into training, testing, (validation) sets Apply cross-validation or static/dynamic validations	Prevents overfitting and enables robust performance assessment
Data Storage & Management	Store in databases/data lakes Maintain version control	Supports reproducibility, scalability, and reliable access
Ethical Considerations	Evaluate bias and fairness Ensure privacy and regulatory compliance	Avoids perpetuating bias and protects sensitive information
Continuous Monitoring & Updating	Monitor model/data performance Detect data drift Retrain/update as needed	Maintains accuracy and relevance as data and conditions change

4.5. Model Management

4.5.1. Risk Governance

Risk governance is about preventing costly mistakes before they happen. The 2008 financial crisis, the London Whale incident, and countless smaller disasters have demonstrated what happens when models aren't properly overseen. Organizations that take this seriously typically have written policies delineating responsibilities: management or board-level committees set high-level objectives, while operational teams handle day-to-day processes.

A model inventory—a catalog of all models in use—sits at the heart of any governance framework. Each entry should detail the model's purpose, its key assumptions, and its current status (prototype, production, deprecated). Without this inventory, organizations don't actually know what models they're running or what their cumulative exposure to model error might be. "We have a model for that somewhere" is not a governance strategy.

Many firms adopt tiered risk classifications to decide how much scrutiny a model warrants. A simple calculator that helps an analyst estimate bond duration doesn't need the same validation rigor as an enterprise valuation engine that determines capital requirements. Classification schemes might range from "low impact" to "mission-critical," with validation and testing requirements scaling accordingly.

For highly critical models, this means extensive backtesting, benchmarking against alternative approaches, and sensitivity analyses that get escalated to senior management. It also means ongoing monitoring—not just initial validation—with scheduled reports about model health. The goal is to create a culture where potential failures get surfaced early and openly, rather than hidden away until a crisis forces them into the light.

4.5.2. Change Management

Change management: no model remains static for long. Assumptions evolve, new asset classes appear, regulations change, and software libraries update (sometimes breaking things in the process). A firm's change management process should standardize how modifications are proposed, evaluated, and documented.

A central repository or version control system is essential. Whenever the model or its associated data structures shift, the changes and their justifications should be recorded. This makes it possible

to track lineage (“when did we start using this assumption?”) and revert to a prior version if an update proves problematic. Later in this book, we’ll introduce version control systems and workflows that make this practical for code-based models.

Equally important is assessing ripple effects. Simplifying a routine or adjusting a discount rate assumption may seem minor in isolation, but can have broad implications when that routine or assumption is used across multiple components. “I just tweaked this one function” can cascade into unexpected changes in reports that stakeholders rely on. Up-front impact assessments help determine which historical results need recalculating and whether stakeholder communication or training is needed before deployment.

We’ll describe package and model version numbering schemes in Chapter 23. The basic idea is that version numbers communicate what kind of change occurred: a patch fix that shouldn’t affect results, a minor enhancement that adds functionality, or a major change that might break compatibility with prior versions. This convention, combined with good release notes, lets users understand what they’re getting when they upgrade.

Communication around changes should be systematic. Concise notes on new features, potential risks, and recommended practices should reach both internal users and (where relevant) regulators. Well-handled change management enables innovation without sacrificing reliability.

4.5.3. Data Controls

Sound data controls matter because flawed inputs will undermine even the sturdiest model architecture. “The numbers looked fine” is not a defense when an audit reveals that a key data feed was stale for three months.

Most organizations define data quality standards addressing accuracy, completeness, and timeliness. These standards help detect common pitfalls: inconsistent formatting (did the vendor switch from MM/DD/YYYY to DD/MM/YYYY?), delayed updates (are we using yesterday’s prices or last week’s?), and incorrect mappings (does this security identifier actually correspond to the security we think it does?). Automated checks at ingestion points catch many issues before they propagate—out-of-range values that might indicate corruption, suspicious spikes that suggest input error, or missing records that break downstream calculations.

Security and access protocols add another layer of protection. Role-based permissions minimize the risk of data tampering, accidental deletions, or unauthorized access to confidential information. Not everyone who needs to view model outputs needs write access to the underlying data.

Data versioning applies to financial datasets just as much as it applies to code. Keeping a record of each dataset’s evolution allows managers and auditors to pinpoint when and how anomalies first appeared. If someone asks “what inputs produced last quarter’s results?” you should be able to answer definitively, not approximately.

Where regulations like GDPR come into play, data controls must also reflect requirements about personal information, consent, and retention periods. Coordinating these efforts under a unified data governance approach ensures that model outputs stand on a solid factual foundation—and that you can demonstrate this to auditors and regulators when asked.

4.5.4. Peer and Technical Review

Even the most experienced modelers benefit from additional eyes on their work. Peer review and technical review are essential for quality assurance. We all have blind spots. A peer reviewer who wasn’t involved in building the model will ask questions that never occurred to the original developer—not because the developer was careless, but because familiarity breeds assumptions.

4. The Practice of Financial Modeling

Peer review can be informal (“hey, can you look at this before I send it?”) or systematically mandated (formal sign-off required before production deployment). Some organizations require independent reviewers who have not contributed to the original model; smaller teams may rely on a rotating schedule of internal experts. The key is cultivating a culture where questioning assumptions is welcomed rather than resented. “Why did you use this discount rate?” should be a normal question, not an accusation.

Technical review goes deeper, focusing on verification of the computations themselves. For complex spreadsheets, this might mean walking through formulas cell by cell. For code, it might mean reviewing logic, running test scenarios, and confirming that edge cases are handled. The goal is to verify that the model actually does what it’s supposed to do—not just that the output looks reasonable.

This process should generate documentation: who performed the review, what methods they used, which issues surfaced, and how they were resolved. If challenges are identified, revisions loop back into the change management system. The documentation then serves as an audit trail, demonstrating that due diligence was performed.

Conceptual soundness also merits review. Does the model align with economic theory? Are the assumptions consistent with domain-specific knowledge? A model can be technically correct but conceptually flawed—using the wrong framework for the problem at hand. Catching this requires reviewers who understand the business context, not just the code.

Peer and technical review, conducted seriously, reinforce consistent quality and catch errors before they reach production. Conducted perfunctorily, they’re just bureaucratic overhead. The difference lies in organizational culture.

4.6. Conclusion

We’ve covered a lot of ground in this chapter: what makes models good (or bad), what distinguishes skilled practitioners, how data flows into models, and how organizations govern and maintain their modeling efforts. These aren’t separate concerns—they’re all interrelated. A well-architected model is easier to validate. Clear documentation makes governance feasible. Curiosity leads to better data practices.

If there’s one theme running through all of this, it’s that financial modeling is fundamentally a human activity. The code and the math matter, but so do the judgment calls, the communication, and the institutional practices that surround them. A model that’s technically correct but incomprehensible is not much better than one with bugs. A model that’s well-documented but built on bad data is still dangerous.

The rest of this book will introduce the technical foundations—the programming concepts, the numerical methods, the domain-specific libraries—that make sophisticated financial modeling possible. But those tools only become useful in the hands of someone who thinks carefully about what they’re modeling and why. That’s the craft we’re trying to develop.

Part III.

Foundations: Programming and Abstractions

“Out of intense complexities intense simplicities emerge.” — Winston Churchill (1923)

Over the next several chapters, we build essential concepts that enable sophisticated financial models while maintaining clarity and control. We begin with core programming building blocks—the vocabulary and grammar of communicating with computers—then explore how to decompose complex problems through abstraction: functions and methods, structs and parametric types, and multiple dispatch as the organizing principle.

Abstraction is selective attention—focusing on what matters for a given purpose while hiding irrelevant detail. Just as financial statements aggregate transaction-level noise to reveal performance and risk, thoughtful abstraction lets us manage complexity by working at the right level of detail: a pricing function that hides payoff minutiae, a portfolio type that encapsulates assets and weights, or a generic risk routine that dispatches on instrument type.

The goal isn’t to make you a computer scientist, but to equip you with the mental models and practical techniques needed to effectively leverage programming in your financial work. Understanding these foundations will help you design clean, maintainable models that can evolve with your needs.

Think of this section as building your modeling toolkit, one concept at a time. Ideas are introduced progressively, with concrete examples to ground the theory in practical application. The concepts build on each other, so take time to ensure your understanding before moving forward.

5. Elements of Programming

CHAPTER AUTHORED BY: ALEC LOUDENBACK

"Programming is not about typing, it's about thinking." — Rich Hickey (2011)

5.1. Chapter Overview

Here we start building up computer science concepts by introducing tangible programming essentials: data types, variables, control flow, functions, and scope.

💡 On Your First Read-through

This chapter is intended to be an introductory reference for most of the basic building blocks upon which we will build abstractions in the chapters that follow. We want this chapter to essentially be an easy and mildly opinionated stepping-stone on your journey.

At some point, you will likely find yourself seeking more precise or thorough documentation and will directly search or read the documentation of a language or library itself. However, it may be intimidating or frustrating reading reference documentation due to the density and terminology while so many concepts are new. Let this chapter (and book, writ large) be a bridge for you.

If reading this book in a linear fashion and new to programming, we suggest skipping the following sections and returning when encountering the concept or term later in the book:

- Section 5.4.4 through Section 5.4.9 which covers advanced and custom data types
- After Section 5.5.3 which deals with advanced function usage and program organization via scope

🔥 Caution

This introductory chapter is intended to provide a survey of the important concepts and building blocks, not to be a complete reference. For full details on available functions, more complete definitions, and a more complete tour of all language features, see the Manual at docs.julialang.org.

5.2. Computer Science, Programming, and Coding

Computer Science is the study of computing and information. As a science, it is distinct from programming languages which are merely coarse implementations of specific computer science

5. Elements of Programming

concepts¹.

Programming (or “coding”) is the art and science of writing code in programming languages to have the computer perform desired tasks. While this may sound mechanistic, programming truly is one of the highest forms of abstract thinking. The design space of potential solutions is so large and potentially complex that much art and experience is needed to create a well-made program.

The language of computer science also provides a lexicon so that financial practitioners can discuss model architecture and characteristics of problems with precision and clarity. Simply having additional terminology and language to describe a concept illuminates aspects of the problem in new ways, opening oneself up to more innovative solutions.

In this light, the financial modeling we do can be considered a type of computer program. It takes abstract information (data) as input, performs calculations (an algorithm), and returns new data as output. We generally don’t need to consider things a software engineer might contemplate—graphical user interfaces, networking, or access restrictions. But the programming fundamentals are there: a good financial modeler must understand data types, algorithms, and some hardware details. Those fundamentals ultimately determine whether a nested solvency routine finishes before dawn or whether a pricing service can evaluate thousands of instruments per second.

We will build up the concepts over this and the following chapter:

- This chapter will provide a survey of important concepts in computer science that will prove useful for our financial modeling. First, we will talk about data types, boolean logic, and basic expressions. We’ll build on those to discuss algorithms (functions) which perform useful work and use control flow and recursion.
- In the following chapters about abstraction, we will step back and discuss higher level concepts: the “schools of thought” around organizing the relationship between data and functions (functional versus object-oriented programming), design patterns, computational complexity, and compilation.

Tip

There will be brief references to hardware considerations for completeness, but hardware knowledge is not necessary to understand most programming languages (including Julia). It’s impossible to completely avoid talking about hardware when you care about the performance of your code, so feel free to gloss over the reference to hardware details on the first read and come back later after Chapter 9.

It’s highly recommended that you follow along and have a Julia session open (e.g. a REPL or a notebook) when first going through this chapter. See the first part of Chapter 21 if you haven’t gotten that set up yet. Follow along with the examples as we go.

Tip

You can get some help in the REPL by typing a ? followed by the symbol you want help with, for example:

¹Said differently, computer science may contemplate ideas and abstractions more generally than a specific implementation, as in mathematics where a theorem may be proved ($a^2 + b^2 = c^2$) without resorting to specific numeric examples ($3^2 + 4^2 = 5^2$).

```
help?> sum
search: sum sum! summary cumsum cumsum! ...
sum(f, itr; [init])
```

Sum the results of calling function f on each element of itr.

... More text truncated...

5.3. Expressions and Control Flow

5.3.1. Naming Values with Variables

One of the first things it will be convenient to understand is the concept of variables. In virtually every programming language, we can assign values to make our program more organized and meaningful to the human reader. In the following example, we assign values to intermediate symbols to benefit us humans as we convert (silly!) American distance units:

```
feet_per_yard = 3
yards_per_mile = 1760

feet = 3000
miles = feet / feet_per_yard / yards_per_mile
```

0.5681818181818182

The above is technically the same thing as just computing $3000 / 3 / 1760$, however we've given the elements names meaningful to the human user.

Beyond readability, variables are a form of **abstraction** which allows us to think beyond specific instances of data and numbers to a more general representation. For example, the last line in the prior code example is a very generic computation of a unit conversion relationship and `feet` could be any number and the expression remains a valid calculation.

Tip

We will dive a little bit deeper into variables and assignment in Section 5.3.4, distinguishing between assignment and references.

5.3.2. Expressions

Within the code examples above, we can zoom in onto small pieces of code called **expressions**. Expressions are effectively the basic block of code that gets evaluated to produce a value. Here is an expression that adds two integers together that evaluate to a new integer (3 in this case):

```
1 + 2
```

3

A bigger program is built up of many of these smaller bits of code.

5. Elements of Programming

5.3.2.1. Compound Expression

There are two kinds of blocks where we can ensure that sub-expressions get evaluated in order and return the last expression as the overall return value: `begin` and `let` blocks.

```
c = begin
    a = 3
    b = 4
    a + b
end

a, b, c
```

```
(3, 4, 7)
```

Alternatively, you can chain together `;`s to create a compound expression:

```
z = (x = 1; y = 2; x + y)
```

```
3
```

Compound expressions allow you to group multiple operations together while still having the entire block evaluate to a single value, typically the last expression. This makes it easy to use complex logic anywhere a value is needed, like in function arguments or assignments.

`let` blocks define variables within its scope and cannot be accessed outside that scope. More on scope in Section 5.6.

```
c = let
    g = 3
    h = 4
    g + h
end

@show c
@show g

c = 7

UndefVarError: UndefVarError(:g, 0x000000000000974d, Main.Notebook)
UndefVarError: `g` not defined in `Main.Notebook`
Suggestion: check for spelling errors or missing imports.
Stacktrace:
 [1] top-level scope
   @ ~/prog/julia-fin-book/foundations-of-programming.qmd:135
```

5.3.2.2. Conditional Expressions

Conditionals are expressions that evaluate to a `boolean` `true` or `false`. This is the beginning of really being able to assemble complex logic to perform useful work. Here are a handful of expressions that would evaluate to `true`:

```
1 > 0
1 == 1 # check for equality
1.0 isa Float64
```

```
(5 > 0) & (-1 < 2) # "and" expression
(5 > 0) | (-1 > 2) # "or" expression
1 != 2
```

true

i Note

In Julia, the booleans have an integer equality: `true` is equal to 1 (`true == 1`) and `false` is equal to 0 (`false == 0`).

However, `true != 5`. Only 1 is equal to true via `==` (in some languages, any non-zero number is “truthy”). Note that `true === 1` is false because they are different types.

Conditionals can be used to assemble different logical paths for the program to follow and the general pattern is an `if` block:

```
if condition
    # do one thing
elseif condition
    # do something else
else
    # do something if none of the
    # other conditions are met
end
```

A complete example:

```
function buy_or_sell(my_value, market_price)
    if my_value > market_price
        "buy more"
    elseif my_value < market_price
        "sell"
    else
        "hold"
    end
end

buy_or_sell(10, 15), buy_or_sell(15, 10), buy_or_sell(10, 10)

("sell", "buy more", "hold")
```

i Advanced: Short Circuiting Operator

Julia has short-circuiting boolean operators that avoid evaluating the right-hand side when it isn't needed:

- `a && b` evaluates `b` only if `a` is `true`
- `a || b` evaluates `b` only if `a` is `false`

5. Elements of Programming

```
function bigger_than_and_print(a, b)
    println("comparing $a and $b")
    a > b
end

(5 > 0) && bigger_than_and_print(5, 7)  # prints, returns false
(5 < 0) && bigger_than_and_print(5, 7)  # skips RHS, returns false

(5 > 0) || bigger_than_and_print(5, 7)  # skips RHS, returns true
(5 < 0) || bigger_than_and_print(5, 7)  # prints, returns false

comparing 5 and 7
comparing 5 and 7

false
```

By contrast, `&` and `|` are non-short-circuit (bitwise/logical) and always evaluate both sides:

```
(5 > 0) & bigger_than_and_print(5, 7)  # prints, returns false
(5 < 0) | bigger_than_and_print(5, 7)  # prints, returns false

comparing 5 and 7
comparing 5 and 7

false
```

This matters for performance and safety:

```
false && error("boom")  # no error (RHS skipped)
false & error("boom")  # ERROR: boom

ErrorException:ErrorException("boom")
boom
Stacktrace:
 [1] error(s::String)
   @ Base ./error.jl:44
 [2] top-level scope
   @ ~/prog/julia-fin-book/foundations-of-programming.qmd:220
```

Tips:

- Use `&&` and `||` for control flow on `Bool` values
- With arrays, use broadcasting: `.&` and `.|` (not `&&/||`)
- With missing data, `&&/||` require `Bool` and will error; `&` and `|` propagate missing

5.3.3. Equality

The “Ship of Theseus problem”² is an example of how equality can be a philosophically complex concept. In computer science we have the advantage that while we may not be able to resolve what’s the “right” type of equality, we can be more precise about it.

Here is an example for which we can see the difference between two types of equality:

- **Egal** equality is when a program could not distinguish between two objects at all
- **Equal** equality is when the values of two objects are the same

If two things are egal, then they are also equal.

In the following example, s and t are equal but not egal:

```
s = [1, 2, 3]
t = [1, 2, 3]
s == t, s === t
```

```
(true, false)
```

One way to think about this is that while the values are equal, there is a way that one of the arrays could be made not equal to the other:

```
t[2] = 5
t
```

```
3-element Vector{Int64}:
```

```
1
5
3
```

Now t is no longer equal to s:

```
s == t
```

```
false
```

The reason this happens is that arrays are containers that can have their contents modified. Even though they originally had the same values, s and t are different containers, and *it just so happened* that the values they contained started out the same.

Some data can’t be modified, including some kinds of collections. Immutable types like the following tuple, with the same stored values, are egal because there is no way for us to make them different:

```
(2, 4) === (2, 4)
```

```
true
```

Using this terminology, we could now interpret the “Ship of Theseus” as that his ship is “equal” but not “egal”.

²The Ship of Theseus problem specifically refers to a legendary ancient Greek ship, owned by the hero Theseus. The paradox arises from the scenario where, over time, each wooden part of the ship is replaced with identical materials, leading to the question of whether the fully restored ship is still the same ship as the original. The Ship of Theseus problem is a thought experiment in philosophy that explores the nature of identity and change. It questions whether an object that has had all of its components replaced remains fundamentally the same object.

5.3.4. Assignment, References, and Mutability

When we say `x = 2` we are **assigning** the integer value of 2 to the variable `x`. This is an expression that lets us bind a value to a variable so that it can be referenced more concisely or in different parts of our code. When we re-assign the variable we are not mutating the value: `x = 3` does not change the 2.

When we have a mutable object (e.g. an `Array` or a `mutable struct`), we can mutate the value inside the referenced container. For example:

```
x = [1, 2, 3]
x[1] = 5
x
```

(1)

(2)

- (1) `x` refers to the array which currently contains the elements 1, 2, and 3.
- (2) We re-assign the first element of the array to be the value 5 instead of 1

```
3-element Vector{Int64}:
5
2
3
```

In the above example, `x` has not been reassigned. It is possible for two variables to refer to the same object:

```
x = [1, 2, 3]
y = x
x[1] = 6
y
```

(1)

- (1) `y` refers to the *same* underlying array as `x`

```
3-element Vector{Int64}:
6
2
3
```

Note that variables can be freely re-assigned. Marking a variable as `const` signals that it should not be re-assigned and enables compiler optimizations:

```
const TAU = π * 2 # <1>
```

- (1) Capitalizing constant variables is a convention in Julia.

In an interactive session, Julia will allow reassignment of a `const` with a warning, but in compiled code this is not permitted.

Warning

Note that if we declare a `const` variable that refers to a mutable container like an `array`, the container can still be mutated. It's the reference to the container that remains constant, not necessarily the elements within the container.

For example, while `MY_ARRAY` will point always to the same array, the array itself can get mutated

```

const MY_ARRAY = [1, 2]
MY_ARRAY[1] = 99
MY_ARRAY

2-element Vector{Int64}:
99
2

```

Being explicit about whether you copied a container or merely referenced it is crucial in model code: mutating a shared array of discount factors by accident can change results elsewhere in the valuation chain. When in doubt, create an explicit copy with `copy` or work with immutable structures.

5.3.5. Loops

Loops are ways for the program to move through a program and repeat expressions while we want it to. There are two primary loops: `for` and `while`.

for loops are loops that iterate over a defined range or set of values. Let's assume that we have the array `v = [6,7,8]`. Here are multiple examples of using a `for` loop in order to print each value to output (`println`):

```

v = [6,7,8]
# use fixed indices
for i in 1:3
    println(v[i])
end

# use the indices of the array
for i in eachindex(v)
    println(v[i])
end

# use the elements of the array
for x in v
    println(x)
end

# use the elements of the array
for x ∈ v           # ∈ is typed \in<tab>
    println(x)
end

```

while loops will run repeatedly until an expression is false. Here's some examples of printing each value of `v` again:

```

# index the array
i = 1
while i <= length(v)
    println(v[i])
    global i += 1

```

①

5. Elements of Programming

```
    end
```

- ① `global` is used to increment `i` by 1. `i` is defined outside the scope of the `while` loop (see Section 5.6).

```
# index the array
i = 1
while true
    println(v[i])
    if i >= length(v)
        break
    end
    global i += 1
end
```

①

- ① `break` is used to terminate the loop manually, since the condition that follows the `while` will never be false.

5.3.6. Performance of Loops

Loops are highly performant in Julia and often the fastest way to accomplish things. This contrasts with advice often given to Python or R users, where vectorized operations are heavily favored over loops for performance.

5.4. Data Types

Data types categorize information by intrinsic characteristics. We instinctively know that 13.24 is different than "this set of words"—types are how we formalize this distinction. Mathematically, it's like having different sets of objects to perform specialized operations on. Underneath this set-like abstraction are implementation details related to computer hardware. You probably know that computers only natively "speak" in binary zeros and ones. Data types are a primary way for a computer to know if it should interpret 01000010 as B or as 66³.

Each 0 or 1 within a computer is called a **bit** and eight bits in a row form a **byte** (such as 01000010). This is where we get terms like "gigabytes" or "kilobits per second" as a measure of the quantity or rate of bits something can handle⁴.

5.4.1. Numbers

Numbers are usually grouped into two categories: **integers** and **floating-point**⁵ numbers. Integers are like the mathematical set of integers while floating-point is a way of representing decimal numbers. Both have some limitations since computers can only natively represent a finite set of numbers due to the hardware (more on this in Chapter 9). Here are three integers that are input into the **REPL** (Read-Eval-Print-Loop)⁶ and the result is **printed** below the input:

³These binary representations correspond to B and 66 with the *ASCII character set* and 8-bit integer encodings respectively, discussed later in this chapter.

⁴Some distinctions you may encounter: in short-form, "kb" means kilobits while the upper-case "B" in "kB" means kilobytes. Also confusingly, sometimes the "k" can be binary or decimal - because computers speak in binary, a binary "k" means 1024 (equal to 2^{10}) instead of the usual decimal 1000. In most computer contexts, the binary (multiples of 1024) is more common.

⁵The term floating point refers to the fact that the number's radix (decimal) point can "float" between the significant digits of the number.

⁶That is, it *reads* the code input from the user, *evaluates* what code was given to it, *prints* the result of the input to the screen, and *loops* through the process again.

```
2
```

```
2
```

```
423
```

```
423
```

```
1929234
```

```
1929234
```

And three floating-point numbers:

```
0.2
```

```
0.2
```

```
-23.3421
```

```
-23.3421
```

```
14e3      # the same as 14,000.0
```

```
14000.0
```

On most systems, `0.2` will be interpreted as a 64-bit floating point type called `Float64` in Julia. Unlike integers, floating-point literals are always `Float64` regardless of system architecture. Given that there are a finite amount of bits attempting to represent a continuous, infinite set of numbers means that some numbers are not able to be represented with perfect precision. For example, if we ask for `0.2`, the closest representations in 64 and 32 bit are:

- `0.20000000298023223876953125` in 32-bit
- `0.200000000000000011102230246251565404236316680908203125` in 64-bit

This leads to special considerations that computers take when performing calculations on floating point maths, some of which will be covered in more detail in Chapter 9. For now, just note that floating point numbers have limited precision and even if we input `0.2`, your computations will use the above decimal representations even if it will print out a number with fewer digits shown:

```
x = 0.2
```

(1)

```
big(x)
```

(2)

- (1) Here, we **assign** the value `0.2` to a **variable** `x`. More on variables/assignments in Section 5.3.4.
 (2) `big(x)` is an arbitrary precision floating point number and by default prints the full precision that was embedded in our variable `x`, which was originally `Float64`.

```
0.200000000000000011102230246251565404236316680908203125
```

Note

Note the difference in what printed between the last example and when we input `0.2` earlier in the chapter. The former had the same (not-exactly equal to `0.2`) *value*, but it printed

5. Elements of Programming

an abbreviated set of digits as a nicety for the user, who usually doesn't want to look at floating point numbers with their full machine precision. The system has the full precision (`0.20...3125`) but is truncating the output.

In the last example, we've converted the normal `Float64` to a `BigFloat` which will not truncate the output when printing.

Integers are similarly represented as 32 or 64 bits (with `Int32` and `Int64`) and are limited to exact precision:

- `Int32` range: `-2,147,483,648` to `2,147,483,647`
- `Int64` range: `-9,223,372,036,854,775,808` to `9,223,372,036,854,775,807`

Additional range in the positive direction if one chooses to use "unsigned", non-negative numbers (`UInt32` and `UInt64`). Unlike floating point numbers, the integers have a type `Int` which will use the system bit architecture by default (that is, `Int(30)` will create a 64 bit integer on 64-bit systems and 32-bit on 32-bit systems).

Floating Point and Excel

Excel's numeric storage and routine is complex and not quite the same as most programming languages, which follow the Institute of Electrical and Electronics Engineer's standards (such as the IEEE 754 standard for double precision floating point numbers). Excel uses IEEE for the *computations* but results (and therefore the cells that comprise many calculations interim values) are stored with 15 significant digits of information. In some ways this is the worst of both worlds: having the sometimes unusual (but well-defined) behavior of floating point arithmetic *and* having additional modifications to various steps of a calculation. In general, you can assume that the programming language result (following the IEEE 754 standard) is a better result because there are aspects to the IEEE 754 defines techniques to minimize issues that arise in floating point math. Some of the issues (round-off or truncation) can be amplified instead of minimized with Excel.

In practice, this means that it can be difficult to exactly replicate a calculation in Excel in a programming language and vice-versa. It's best to try to validate a programming model versus Excel model using very small unit calculations (e.g. a single step or iteration of a routine) instead of an all-in result. You may need to define some tolerance threshold for comparison of a value that is the result of a long chain of calculation.

Currencies and Decimals

Due to the inherent imprecision of floating point numbers, they should **not** be used in storing financial transaction records! The trade-offs inherent in floating point math described above do not lend itself to accurate record-keeping. For example, in looking at summing up two products sold for \$19.99 and \$84.99, since floating point operations like `19.99 + 84.99` do not precisely equal `105.98`.

`BigFloat(19.99 + 84.99)`

`104.97999999999989768184605054557323455810546875`

See how the prior is slightly lower than `105.98` — if we were adding US Dollars here, our system would be off by fractions of a cent. Do that for millions of transactions in a day and you have a problem!

Generally, when doing *modeling* or even creating a *valuation model*, it's okay to use floating point math. As an example, if you are trying to determine the value of an exotic option, your model is likely just fine outputting a value like 101.987087 . If you go and sell this option, you'll have to settle for either 101.98 or 101.99 when booking it. In most contexts this imprecision is likely okay!

If you are implementing a transaction or trading system, ensure proper treatment of the types representing your monetary numbers. A full treatment is beyond the scope of this book, but for a good introduction to the subject, see <https://cs-syd.eu/posts/2022-08-22-how-to-deal-with-money-in-software>.

5.4.2. Type Hierarchy

We can describe a *hierarchy* of types. Both `Float64` and `Int64` are examples of `Real` numbers (here, `Real` is an **abstract** Julia type which corresponds to the mathematical set of real numbers commonly denoted with \mathbb{R}). Both `Float64` and `Int32` are `Real` numbers, so why not just define all numbers as a `Real` type? Because for performant calculations, the computer must know in advance how many bits each number is represented with.

We can use Julia to introspect its own type hierarchy:

```
function print_type_tree(T::Type, indent=0)
    println("  " ^ indent * string(T))
    for sub in subtypes(T)
        print_type_tree(sub, indent + 1)
    end
end
print_type_tree(Number)

Number
Base.MultiplicativeInverses.MultiplicativeInverse
Base.MultiplicativeInverses.SignedMultiplicativeInverse
Base.MultiplicativeInverses.UnsignedMultiplicativeInverse
Complex
Real
AbstractFloat
    BigFloat
    Core.BFloat16
    Float16
    Float32
    Float64
AbstractIrrational
    Irrational
Integer
    Bool
    Signed
        BigInt
        Int128
        Int16
        Int32
        Int64
        Int8
Unsigned
    UInt128
```

5. Elements of Programming

```
UInt16  
UInt32  
UInt64  
UInt8  
Rational
```

The integer and floating point types described in the prior section are known as **concrete types** because there are no possible sub types (child types). Further, a concrete type can be a **bit type** if the data type will always have the same number of bits in memory: a `Float32` will always be 32 bits in memory, for example. Contrast this with strings (described below) which can contain an arbitrary number of characters.

5.4.3. Collections

Collections are types for storing data that contains many elements. This section describes some of the most common and useful types of containers.

5.4.3.1. Arrays

Arrays are the most common way to represent a collection of similar data. For example, we can represent a set of integers as follows:

```
[1, 10, 300]
```

```
3-element Vector{Int64}:  
 1  
 10  
 300
```

And a floating point array:

```
[0.2, 1.3, 300.0]
```

```
3-element Vector{Float64}:  
 0.2  
 1.3  
 300.0
```

Note the above two arrays are different types of arrays. The first is `Vector{Int64}` and the second is `Vector{Float64}`. These are arrays of concrete types and so Julia will know that each element of an array is the same amount of bits, which will enable more efficient computations. With the following set of mixed numbers, Julia will **promote** the integers to floating point since the integers can be accurately represented⁷ in floating point.

```
[1, 1.3, 300.0, 21]
```

```
4-element Vector{Float64}:  
 1.0  
 1.3  
 300.0  
 21.0
```

⁷Accurate only to a limited precision, as described in Section 5.4.1.

However, if we explicitly ask Julia to use a `Real`-typed array, the type is now `Vector{Real}`. Recall that `Real` is an abstract type. Having heterogeneous types within the array is conceptually fine, but in practice limits performance. Again, this will be covered in more detail in Chapter 9.

In Julia, arrays can be multi-dimensional. Here are two three-dimensional arrays with length three in each dimension:

```
rand(3, 3, 3)
```

```
3x3x3 Array{Float64, 3}:
[:, :, 1] =
0.505707  0.846021  0.277581
0.0672428 0.573422  0.892956
0.417667  0.274474  0.990571

[:, :, 2] =
0.152057  0.165935  0.937167
0.130106  0.932558  0.0753073
0.712946  0.42471   0.840693

[:, :, 3] =
0.279693  0.994087  0.509565
0.516839  0.0356226 0.529319
0.965268  0.426369  0.116956
```

```
[x + y + z for x in 1:3, y in 11:13, z in 21:23]
```

```
3x3x3 Array{Int64, 3}:
[:, :, 1] =
33 34 35
34 35 36
35 36 37

[:, :, 2] =
34 35 36
35 36 37
36 37 38

[:, :, 3] =
35 36 37
36 37 38
37 38 39
```

The above example demonstrates **array comprehension** syntax, which is a convenient way to create arrays in Julia.

A two-dimensional array has the rows separated by semi-colons (;):

```
x = [1 2 3; 4 5 6]
```

```
2x3 Matrix{Int64}:
1 2 3
4 5 6
```

5. Elements of Programming

i Note

In Julia, a `Vector{Float64}` is simply a one-dimensional array of floating points and a `Matrix{Float64}` is a two-dimensional array. More precisely, they are **type aliases** of the more generic `Array{Float64,1}` and `Array{Float64,2}` names. Arrays with three or more dimensions don't have a type alias pre-defined.

5.4.3.2. Array indexing

Array elements are accessed with the integer position, starting at 1 for the first element⁸ ⁹:

```
v = [10, 20, 30, 40, 50]  
v[2]
```

20

We can also access a subset of the vector's contents by passing a range:

```
v[2:4]
```

3-element Vector{Int64}:

```
20  
30  
40
```

And we can generically reference the array's contents, such as:

```
v[begin+1:end-1]
```

3-element Vector{Int64}:

```
20  
30  
40
```

We can assign values into the array as well, as well as combine arrays and push new elements to the end:

```
v[2] = -1  
push!(v, 5)  
vcat(v, [1, 2, 3])
```

9-element Vector{Int64}:

```
10  
-1  
30  
40  
50  
5  
1  
2  
3
```

⁸Whether an index starts at 1 or 0 is sometimes debated. Zero-based indexing is natural in the context of low-level programming which deal with bits and positional *offsets* in computer memory. For higher level programming one-based indexing is more natural: in a set of data stored in an array, it is much more natural to reference the *first* (through n^{th}) datum instead of the *zeroth* (through $(n - 1)^{th}$) datum.

⁹Arrays in Julia can actually be indexed with an arbitrary starting point: see the package `OffsetArrays.jl`

5.4.3.3. Array Alignment

When you have an $M \times N$ matrix (M rows, N columns), a choice must be made as to which elements are next to each other in memory. Typical math convention and fundamental computer linear algebra libraries (dating back decades!) are column major and Julia follows that legacy. **Column major** means that elements going down the rows of a column are stored next to each other in memory. This is important to know so that (1) you remember that vectors are treated like a column vector when working with arrays (that is: a N element 1D vector is like a $N \times 1$ matrix), and (2) when iterating through an array, it will be faster for the computer to access elements next to each other column-wise. A 10×10 matrix is actually stored in memory as 100 elements coming in order, one after another in single file.

This 3x3 matrix is stored with the elements of columns next to each other, which we can see with `vec`:

```
mat = [1 2 3; 4 5 6; 7 8 9]

3x3 Matrix{Int64}:
1 2 3
4 5 6
7 8 9

vec(mat) # column major order

9-element Vector{Int64}:
1
4
7
2
5
8
3
6
9
```

5.4.3.4. Ranges

A **range** is a compact representation of a sequence of numbers. We actually used them above to index into arrays. They are expressed as `start:stop`

We don't have to actually store all of these numbers on the computer somewhere as in an `Array`. Instead, this is an object that *represents* the ordered set of numbers. So for example, we can "sum" up a ridiculous amount of numbers almost instantaneously:

This is possible due to two things:

1. not needing to actually store that many numbers in memory, and

5. Elements of Programming

2. Julia being smart enough to apply the triangular number formula¹⁰ when `sum` is given a consecutive range.

There are more general ways to construct ranges:

Step by another number instead of the default 1:

```
1:2:7
```

1:2:7

Specify the number of values within the range, inclusive of the first number¹¹:

```
# range(start, stop, length)
range(0, 10, 21)
```

0.0:0.5:10.0

5.4.3.5. Characters, Strings, and Symbols

Characters are represented in most programming languages as letters within quotation marks. In Julia, individual characters are represented using single quotes:

```
'a'
```

'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

Letters and other characters present more difficulties than numbers to represent within a computer (think of how many languages and alphabets exist!), and it essentially only works because the world at large has agreed to a given representation. Originally **ASCII** (American Standard Code for Information Interchange) was used to represent just 95 of the most common English characters ("a" through "z", zero through nine, etc.). Now, **UTF** (Unicode Transformation Format) can encode more than a million characters and symbols from many human languages.

Strings are a collection¹² of characters, and can be created in Julia with double quotes:

```
"hello world"
```

"hello world"

It's easy to ascertain how 'normal' characters can be inserted into a string, but what about things like new lines or tabs? They are represented by their own characters but are normally not printed in computer output. However, those otherwise invisible characters do exist. For example, here we will use a **string literal** (indicated by the """) to tell Julia to interpret the string as given, including the invisible new line created by hitting return on the keyboard between the two words:

¹⁰The triangular numbers (sum of integers from 1 to n) are:

$$T_n = \sum_{k=1}^n k = 1 + 2 + \dots + n = \frac{n^2 + n}{2} = \frac{n(n+1)}{2} = \binom{n+1}{2}$$

¹¹If the `step` keyword argument is defined: whether the last number is in the resulting range depends on if the step evenly divides the end of the range. In the example `1:2:7`, the `step` is 2.

¹²Under the hood, strings are essentially a vector of characters but there are complexities with character encoding that don't allow a lossless conversion to individual characters of uniform bit length. This is for historical compatibility reasons and to avoid making most documents' file sizes larger than it needs to be.

```
"""
hello
world
"""

"hello\nworld\n"
```

The output above shows the `\n` character contained within the string.

Symbols are a way of representing an identifier which cannot be seen as a collection of individual characters. `:helloworld` is distinct from `"helloworld"` - you can kind of think of the former as an un-executed bit of code - if we were to execute it (with `eval(:helloworld)`), we would get an error `UndefVarError: 'helloworld' not defined`. Symbols can *look* like strings but do not behave like them. For now, it is best to not worry about symbols but it is an important aspect of Julia which allows the language to represent aspects of itself as data. This allows for powerful self-reference and self-modification of code but this is a more advanced topic generally out of scope of this book.

5.4.3.6. Tuples

Tuples are a fixed, ordered collection of values, denoted by values inside parentheses and separated by a comma. They are best suited for small collections of related data. An example might be x-y coordinates in 2 dimensional space:

```
x = 3
y = 4
p1 = (x, y)
```

(3, 4)

A tuple's values can be accessed like arrays:

```
p1[1]
```

3

Tuples fill a middle ground between scalar types and arrays in more ways than one:

- Tuples have no problem having heterogeneous types in the different slots.
- Tuples are **immutable**, meaning that you cannot overwrite the value in memory (an error will be thrown if we try to do `p[1] = 5`).
- It's generally expected that within an array, you would be able to apply the same operation to all the elements (e.g. square each element) or do something like sum all of the elements together, which isn't generally the case for a tuple.
- Tuples are generally stack allocated instead of being heap allocated like arrays¹³, meaning that a lot of times they can be faster than arrays.

¹³What this means will be explained in Chapter 9.

5. Elements of Programming

5.4.3.6.1. Named Tuples

Named tuples provide a way to give each field within the tuple a specific name. For example, our x-y coordinate example above could become:

```
p2 = (x=3, y=4)  
(x = 3, y = 4)
```

The benefit is that we can give more meaning to each field and access the values in a nicer way. Previously, we used `location[1]` to access the x-value, but with the new definition we can access it by name:

```
p2.x  
3
```

5.4.3.7. Dictionaries

Dictionaries are a container which relates a **key** to an associated **value**. Kind of like how arrays relate an index to a value, but the difference is that a dictionary is (1) un-ordered and (2) the key doesn't have to be an integer.

Here's an example which relates a name to an age:

```
d = Dict(  
    "Joelle" => 10,  
    "Monica" => 84,  
    "Zaylee" => 39,  
)  
  
Dict{String, Int64} with 3 entries:  
  "Monica" => 84  
  "ZayLee" => 39  
  "Joelle" => 10
```

Then we can look up an age given a name:

```
d["Zaylee"]  
39
```

Dictionaries are super flexible data structures and can be used in many situations.

5.4.4. Parametric Types

We just saw how tuples can contain heterogeneous types of data inside a common container. **Parametric Types** are a way of allowing types themselves to be variable, with a wrapper type containing a to-be-specified inner-type.

Let's look at this a little bit closer by looking at the full type:

```
typeof(p1)
```

```
Tuple{Int64, Int64}
```

p1 is a Tuple{Int64, Int64} type, which means that its first and second elements are both Int64. Contrast this with:

```
typeof(("hello", 1.0))
```

```
Tuple{String, Float64}
```

These tuples are both of the form Tuple{T,U} where T and U are both types. Why does this matter? We and the compiler can distinguish between a Tuple{Int64,Int64} and a Tuple{String,Float64} which allows us to reason about things ("I can add the first element of tuple together only if both are numbers") and the compiler to optimize (sometimes it can know exactly how many bits in memory a tuple of a certain kind will need and be more efficient about memory use). Further, we will see how this can become a powerful force in writing appropriately abstracted code and more logically organize our entire program when we encounter "multiple dispatch" later on.

This is a very powerful technique - we've already seen the flexibility of having an Array type which can contain arbitrary inner types and dimensions. The full type signature for an Array looks like Array{InnerType,NumDimensions}.

```
let
    x = [1 2
          3 4]
    typeof(x)
end
```

```
Matrix{Int64} (alias for Array{Int64, 2})
```

5.4.5. Types for things not there

`nothing` represents that there's nothing to be returned - for example if there's no solution to an optimization problem or if a function just doesn't have any value to return (such as in the case with input/output like `println`).

`missing` is to represent a value that *should* be there but is not, as is all too common in real-world data. Julia natively supports `missing` and three-value logic, which is an extension of the two-value boolean (true/false) logic, to handle missing logical values:

Table 5.1.: NOT logic

NOT (!)	Value
<code>true</code>	<code>false</code>
<code>missing</code>	<code>missing</code>
<code>false</code>	<code>true</code>

Table 5.2.: AND logic

AND (&)	true	missing	false
true	true	missing	false
missing	missing	missing	false
false	false	false	false

Table 5.3.: OR Logic

OR ()	true	missing	false
true	true	true	true
missing	true	missing	missing
false	true	missing	false

Three value logic with true, missing, and false.

Tip

Missing and Nothing are the *types* while missing and nothing are the values here¹⁴. This is analogous to Float64 being a type and 2.0 being a value.

5.4.6. Union Types

When two types may arise in a context, **union types** are a way to represent that. For example, if we have a data feed and we know that it will produce *either* a Float64 or a Missing type then we can say that the value for this is Union{Float64, Missing}. This is much better for the compiler (and our performance!) than saying that the type of this is Any.

5.4.7. Creating User Defined Types

We've talked about some built-in types but so much additional capabilities come from being able to define our own types. For example, taking the x-y-coordinate example from above, we could do the following instead of defining a tuple:

```
struct BasicPoint
    x::Int64
    y::Int64
end

p3 = BasicPoint(3, 4)
```

BasicPoint(3, 4)

BasicPoint is a **composite type** because it is composed of elements of other types. Fields are accessed the same way as named tuples:

¹⁴Missing and Nothing are instances of **singleton type**, which means that there is only a single value that either type can take on.

p3.x, p3.y

(1)

- ① Note that here, Julia will return a tuple instead of a single value due to the comma separated expressions.

(3, 4)

structs in Julia are immutable like tuples above.

But wait, didn't tuples let us mix types too via parametric types? Yes, and we can do the same with our type!

```
struct Point{T}
    x::T
    y::T
end
```

(1)

- ① The {T} after the type's name allows for different Points to be created depending on what the type of the underlying x and y is.

Here's two new points which now have different types:

```
p4 = Point(1, 4)
p5 = Point(2.0, 3.0)

p4, p5
```

(Point{Int64}(1, 4), Point{Float64}(2.0, 3.0))

Note that the types are not equal because they have different type parameters!

```
typeof(p4), typeof(p5), typeof(p4) == typeof(p5)

(Point{Int64}, Point{Float64}, false)
```

Both p4 and p5 are instances of different concrete types Point{Int} and Point{Float64}. The expression X isa Y is true when X is a (sub)type of Y:

```
p4 isa Point, p5 isa Point

(true, true)
```

Note though, that x and y are both of the same type in each Point that we created. If instead we wanted to allow the coordinates to be of different types, then we could have defined Point as follows:

```
struct Point{T,U}
    x::T
    y::U
end
```

i Note

Can we define the structs above without indicating a (parametric) type? Yes!

5. Elements of Programming

```
struct PointUntyped
    x # no type here!
    y # no type declared here either!
end
```

But! x and y will both be allowed to be Any, which is the fallback type where Julia says that it doesn't know any more about the type until runtime (the time at which our program encounters the data when running). Observe that the type of x and y here is Any:

```
fieldtypes(PointUntyped)
(Any, Any)
```

This means that the compiler (and us!) can't reason about or optimize the code as effectively as when the types are explicit or parametric. This is an example of how Julia can provide a nice learning curve - don't worry about the types until you start to get more sophisticated about the program design or need to extract more performance from the code.

The above `structs` that we have defined are examples of **concrete types** which hold data. **Abstract types** don't directly hold data themselves but are used to define a hierarchy of types which we will later exploit (Chapter 8) to implement custom behavior depending on what type our data is.

Here's an example of (1) defining a set of related types that sits above our `Point2D`:

```
abstract type Coordinate end
abstract type CartesianCoordinate <: Coordinate end
abstract type PolarCoordinate <: Coordinate end

struct Point2D{T} <: CartesianCoordinate
    x::T
    y::T
end

struct Point3D{T} <: CartesianCoordinate
    x::T
    y::T
    z::T
end

struct Polar2D{T} <: PolarCoordinate
    r::T
    θ::T
end
```

💡 Unicode Characters

Julia has wonderful Unicode support, meaning that it's not a problem to include characters like θ . The character can be typed in Julia editors by entering `\theta` and then pressing the TAB key on the keyboard.

Unicode is helpful for following conventions that you may be used to in math. For example, the math formula $\text{circumference}(r) = 2 \times r \times \pi$ can be written in Julia with `circumference(r) = 2 * r * π`.

The name for the characters follows the same for LaTeX, so you can search the internet to find the appropriate name: for example search “theta LaTeX”. Further, you can use the REPL help mode to find out how to enter a character if you can copy and paste it from somewhere:

```
help?> θ
"θ" can be typed by \theta<tab>
```

To constrain the types that could be used within our coordinates above, such as if we wanted the fields to all be Real-valued, we could modify the struct definitions with the `<:Real` annotation:

```
struct Point2D{T<:Real} <: CartesianCoordinate
    # ...
end

struct Point3D{T<:Real} <: CartesianCoordinate
    # ...
end

struct Polar2D{T<:Real} <: PolarCoordinate
    # ...
end
```

5.4.8. Mutable structs

It is possible to define structs where the data can be modified - such a data field is said to be **mutable** because it can be changed or mutated. Here’s an example of what it would look like if we made `Point2D` mutable:

```
mutable struct Point2D{T}
    x::T
    y::T
end
```

You may find that this more naturally represents what you are trying to do. However, recall that an advantage of an immutable datatype is that costly memory doesn’t necessarily have to be allocated for it. So you may think that you’re being more efficient by re-using the same object... but it may not actually be faster. Again, more will be revealed in Chapter 9.

💡 Financial Modeling Pro Tip

For financial models, it is best practice to default to immutable structs. Immutability prevents accidental modification of data, making your model’s state easier to reason about and debug. This is especially critical in complex models with many interacting components. Use mutable struct only when you have a specific reason to modify data in-place.

5.4.9. Constructors

Constructors are functions that return a data type (functions will be covered more generally later in the chapter). When we declare a struct, an implicit function is defined that takes a tuple of arguments and returns the data type that was declared. In the following example, after we define

5. Elements of Programming

If you define a type `MyType` the struct, Julia creates a function (also called `MyType`) which takes two arguments and will return the datatype `MyType`:

```
struct MyDate
    year::Int
    month::Int
    day::Int
end

methods(MyDate)

# 2 methods for type constructor:
[1] Main.Notebook.MyDate(year::Int64, month::Int64, day::Int64)
    @ ~/prog/julia-fin-book/foundations-of-programming.qmd:974
[2] Main.Notebook.MyDate(year, month, day)
    @ ~/prog/julia-fin-book/foundations-of-programming.qmd:974
```

Implicit constructors are nice in that you don't have to define a default method and the language does it for you. Sometimes there's reasons to want to control how an object is created, either for convenience or to enforce certain restrictions.

We can use an inner constructor (i.e. inside the `struct` block) to enforce restrictions:

```
struct MyDate
    year::Int
    month::Int
    day::Int

    function MyDate(y,m,d)
        if !(m in 1:12)
            error("month is not between 1 and 12")
        elseif !(d in 1:31)
            error("day is not between 1 and 31")
        else
            return new(y,m,d)
        end
    end
end
```

And outer constructors are simply functions defined that have the same name as the data type, but are not defined inside the `struct` block. Extending the `MyDate` example, say we want to provide a default constructor for if no day is given such that the date returns the 1st of the month:

```
function MyDate(y,m)
    return MyDate(y,m,1)
end
```

5.5. Functions

Functions are a set of expressions that take inputs and return specified outputs.

5.5.1. Special Operators

Operators are the glue of expressions which combine values. We've already seen quite a few, but let's develop a little bit of terminology for these functions.

Unary operators are operators which only take a single argument. Examples include the `!` which negates a boolean value or `-` which negates a number:

```
!true, -5
```

```
(false, -5)
```

Binary operators take two arguments and are some of the most common functions we encounter, such as `+` or `-` or `>`:

```
1 + 2, 1 - 2, 1 > 2
```

```
(3, -1, false)
```

The above unary and binary operators are special kinds of functions which don't require the use of parenthesis. However, they can be written with parentheses for greater clarity:

```
!(true), -(5), +(1, 2), -(1, 2)
```

```
(false, -5, 3, -1)
```

In Julia, we distinguish between **functions** which define behavior that maps a set of inputs to outputs. But a single function can adapt its behavior to the arguments themselves. We have just seen the function `-` be used in two different ways: negation and subtraction depending on whether it had one or two arguments given to it. In this way there is a conceptual hierarchy of functions that complements the hierarchy we have discussed in relation to types:

- `-` is the overall function
- `-(x)` is a unary function which negates its values, `-(x,y)` subtracts `y` from `x`
- Specific methods are then created for each combination of concrete types: `-(x::Float64)` is a different method than `-(x::Int)`

Methods are specific compiled versions of the function for specific types. This is important because at a hardware level, operations for different types (e.g. integers versus floating point) differ considerably. By optimizing for the specific types Julia is able to achieve nearly ideal performance without the same sacrifices of other dynamic languages. We will develop more with respect to methods when we talk about dispatch in Chapter 8.

For example, `factorial` would be referred to as the *function*, while specific implementations are called *methods*. We can see all of the methods for any function with the `methods` function, like the following for `factorial` which has implementations taking into account the specialized needs for different types of arguments:

```
methods(factorial)
```

```
# 7 methods for generic function "factorial" from Base:
[1] factorial(n::UInt128)
    @ combinatorics.jl:33
[2] factorial(n::Int128)
    @ combinatorics.jl:32
[3] factorial(x::BigFloat)
    @ mpfr.jl:855
```

5. Elements of Programming

```
[4] factorial(n::BigInt)
    @ gmp.jl:704
[5] factorial(n::Union{Int16, Int32, Int8, UInt16, UInt32, UInt8})
    @ combinatorics.jl:40
[6] factorial(n::Union{Int64, UInt64})
    @ combinatorics.jl:34
[7] factorial(n::Integer)
    @ intfuncs.jl:1207
```

5.5.2. Defining Functions

Functions more generally are defined like so:

```
function functionname(arguments)
    # ... code that does things
end
```

Here's an example which returns the difference between the highest and lowest values in a collection:

```
function value_range(collection)

    hi = maximum(collection)
    lo = minimum(collection)
    return hi - lo
end
```

①

- ① `return` is optional but recommended to convey to readers of the program where you expect your function to terminate and return a value.

5.5.3. Defining Methods on Types

Here's another example of a function which calculates the distance between a point and the origin:

```
function distance(point)
    return sqrt(point.x^2 + point.y^2)
end
```

①

②

- ① A function block is declared with the name `distance` which takes a single argument called `point`
② We compute the distance formula for a point with `x` and `y` coordinates. The `return` value makes explicit what value the function will output.

`distance` (generic function with 1 method)

Note

An alternate, simpler function syntax for `distance` would be:

```
distance(point) = sqrt(point.x^2 + point.y^2)
```

However, we might at this point note a flaw in our function's definition if we think about the various Coordinates we defined earlier: our definition would currently only work for `Point2D`. For example, if we try a `Point3D` we will get the wrong answer:

```
distance(Point3D(1, 1, 1))
```

1.4142135623730951

The above value should be $\sqrt{3}$, or approximately 1.73205.

What we need to do is define a refined distance for each type, which we'll call `dist` to distinguish from the earlier definition.

```
"""
dist(point)

The euclidean distance of a point from the origin.
"""

dist(p::Point2D) = sqrt(p.x^2 + p.y^2)
dist(p::Point3D) = sqrt(p.x^2 + p.y^2 + p.z^2)
dist(p::Polar2D) = p.r
```

`dist` (generic function with 3 methods)

Now our result will be correct:

```
dist(Point3D(1, 1, 1))
```

1.7320508075688772

This is referred to as **dispatching** on the argument types. Julia will look up to find the most specific method of a function for the given argument types, and falling back to a generic implementation if one is defined.

In Chapter 8 we will see how dispatch (single and multiple) can provide very nice abstractions to simplify the design of a model.

Docstrings (Documentation Strings)

Notice the strings preceding the definition of `dist`. In Julia, putting a string ("...") or string literal (""""...""") right above the definition will allow Julia to recognize the string as documentation and provided it to the user in help mode (Section 21.4.1) and/or have a documentation tool create a webpage or PDF documentation resource.

Defining Methods for Parametric Types

We learned that `Float64 <: Real` in the type hierarchy. However, note that `Tuple{Float64}` is not a sub-type of `Tuple{Real}`. This is called being **invariant** in type theory... but for our purposes this just practically means that when we define a method we need to specify that we want it to apply to all subtypes.

For example, `myfunction(x::Tuple{Real})` would *not* be called if `x` was a `Tuple{Float64}` because it's not a sub-type of `Tuple{Real}`. To act the way we want, we would define the method with the signature of `myfunction(x::Tuple{<:Real})` or `myfunction(x::Tuple{T})` where `{T<:Real}`.

5. Elements of Programming

5.5.4. Keyword Arguments

Keyword arguments are arguments that are passed to a function using named identifiers rather than position. In the following example, `filepath` is a **positional argument** while the two arguments after the semicolon (`;`) are keyword arguments.

```
function read_data(filepath; normalizenames, hasheaderrow)
    # ... function would be defined here
end
```

The function would need to be called and have the two keyword arguments specified:

```
read_data("results.csv"; normalizenames=true, hasheaderrow=false)
```

5.5.5. Default Arguments

We are able to define default arguments for both positional and keyword arguments via an assignment expression in the function signature. For example, we can make it so that the user need not specify all the options for each call. Modifying the prior example so that typical CSVs work with less customization from the user:

```
function read_data(filepath;
    normalizenames = true,
    hasheader = false
)
```

This is a simplified example, but if you look at the documentation for most data import packages you'll see a lot of functionality defined via keyword arguments which have sensible defaults so that most of the time you need not worry about modifying them.

5.5.6. Anonymous Functions

Anonymous functions are functions that have no name and are used in contexts where the name does not matter. The syntax is `x → ...expression with x....`. As an example, say that we want to create a vector from another where each element is squared. `map` applies a function to each member of a given collection:

```
v = [4, 1, 5]
map(x -> x^2, v)
```

(1)

(1) The `x → x^2` is the anonymous function in this example.

```
3-element Vector{Int64}:
16
1
25
```

They are often used when constructing a new value from another value, or defining a function within optimization or solving routines.

5.5.7. First Class Nature

Functions in many languages including Julia are **first class** which means that functions can be assigned and moved around like data variables.

In this example, we have a general approach to calculate the error of a modeled result compared to a known truth. In this context, there are different ways to measure error of the modeled result and we can simplify the implementation of loss by keeping the different kinds of error defined separately. Then, we can assign a function to a variable and use it as an argument to another function:

```
function square_error(guess, correct)
    (correct - guess)^2
end

function abs_error(guess, correct)
    abs(correct - guess)
end

# obs meaning "observations"
function loss(modeled_obs,
            actual_obs,
            loss_function
        )
    sum(
        loss_function.(modeled_obs, actual_obs)
    )
end

let
    a = loss([1, 5, 11], [1, 4, 9], square_error)           ①
    b = loss([1, 5, 11], [1, 4, 9], abs_error)             ②
    a, b
end
```

- ① `loss_function` is a variable that will refer to a function instead of data.
- ② Using a `let` block here is good practice to not have temporary variables `a` and `b` scattered around our workspace.
- ③ Using a function as an argument to another function is an example of functions being treated as “first class”.

(5, 3)

5.5.8. Broadcasting

Looking at the prior definition of `dist`, what if we wanted to compute the squared distance from the origin for a set of points? If those points are stored in an array, we can **broadcast** functions to all members of a collection at the same time. This is accomplished using the **dot-syntax** as follows:

```
points = [Point2D(1, 2), Point2D(3, 4), Point2D(6, 7)]
dist.(points) .^ 2

3-element Vector{Float64}:
 5.000000000000001
```

5. Elements of Programming

```
25.0
```

```
85.0
```

Let's unpack that a bit more:

1. The `.` in `dist.(points)` tells Julia to apply the function `dist` to each element in `points`.
2. The `.` in `.^` tells Julia to square each value as well

Why broadcasting is useful:

1. Without needing any redefinition of functions we were able to transform the function `dist` and exponentiation (`^`) to work on a collection of data. This means that we can keep our code simpler and easier to reason about (operating on individual things is easier than adding logic to handle collections of things).
2. When multiple broadcasted operations are joined together, Julia can **fuse** the operations so that each operation is performed at the same time instead of each step sequentially. That is, if the operation were not fused, the computer would first calculate `dist` for each point, and then apply the square on the collection of distances. When it's fused, the operations are combined into a single pass over the data without creating an interim set of values.

Note

For readers coming from numpy-flavored Python or R, broadcasting is a way that can feel familiar to the array-oriented behavior of those two languages. Once you feel comfortable with Julia in general, you may find yourself relaxing and relying less on array-oriented design and instead picking whichever iteration paradigm feels most natural for the problem at hand: loops or broadcasting over arrays.

5.5.8.1. Broadcasting Rules

What happens if one of the collections is not the same size as the others? When broadcasting, singleton dimensions (i.e. the 1 in $1 \times N$, "1-by- N ", dimensions) will be expanded automatically when it makes sense. For example, if you have a single element and a one dimensional array, the single element will be expanded in the function call without using any additional memory (if that dimension matches one of the dimensions of the other array).

The rules with an $M \times N$ and a $P \times Q$ array:

- either (M and P) or (N and Q) need to be the same, *and*
- one of the non-matching dimensions needs to be 1

Some examples might clarify. This 1×1 element is being combined with a 4×1 , so there is a compatible dimension (N and Q match, M is 1):

```
2 .^ [0, 1, 2, 3]
```

```
4-element Vector{Int64}:
1
2
4
8
```

Here, this 1×3 works with the 2×3 (N and Q match, M is 1)

```
[1 2 3] .+ [1 2 3; 4 5 6]
```

2x3 Matrix{Int64}:

2	4	6
5	7	9

This 3x1 isn't compatible with this 2x3 array (neither M and P nor N and Q match)

```
[1, 2, 3] .+ [1 2 3; 4 5 6]
```

This 2x2 isn't compatible with the 2x3 (M and P match, but neither N nor Q is 1):

```
[1 2; 3 4] .+ [1 2 3; 4 5 6]
```

5.5.8.2. Not Broadcasting

What if you do not want the array to be used element-wise when broadcasting? Then you can wrap the array in a `Ref`, which is used in broadcasting to make the array be treated like a scalar. In the example below, `in(needle, haystack)` searches a collection (`haystack`) for an item (`needle`) and returns true or false if the item is in the collection:

```
in(4, [1 2 3; 4 5 6])
```

true

What if we had an array of things ("needles") that we wanted to search for? By default, broadcasting would effectively split the array up into collections of individual elements to search:

```
in.([1, 9], [1 2 3; 4 5 6])
```

2x3 BitMatrix:

1	0	0
0	0	0

Effectively, broadcasting paired up each element of the vector with each row of the matrix:

```
in(1, [1, 2, 3]) # first element checked against first row
in(9, [4, 5, 6]) # second element checked against second row
```

If we were expecting Julia to return `[true, false]` (that the first needle is in the haystack but the second needle is not), then we need to tell Julia not to broadcast along the second array with `Ref`:

```
in.([1, 9], Ref([1 2 3; 4 5 6]))
```

2-element BitVector:

1
0

5. Elements of Programming

5.5.9. Passing by Sharing

We often want to share data between scopes, such as between modules or by passing data into a function's scope. Arguments to a function in Julia are **passed-by-sharing** which means that an outside variable can be mutated from within a function. We can modify the array in the outer scope (scope discussed later in this chapter) from within the function. In this example, we modify the array that is assigned to `v` by doubling each element:

```
v = [1, 2, 3]

function double!(v)
    for i in eachindex(v)
        v[i] = 2 * v[i]
    end
end

double!(v)

v
```

3-element Vector{Int64}:

2
4
6

Tip

Convention in Julia is that a function that modifies its arguments has a `!` in its name and we follow this convention in `double!` above. Another example would be the built-in function `sort!` which will sort an array in-place without allocating a new array to store the sorted values.

We won't discuss all potential ways that programming languages can behave in this regard, but an alternative that one may have seen before (e.g. in Matlab) is pass-by-value where a modification to an argument only modifies the value within the scope. Here's how to replicate that in Julia by copying the value before handing it to a function. This time, `v` is not modified because we only passed a copy of the array and not the array itself:

```
v = [1, 2, 3]
double!(copy(v))
v

3-element Vector{Int64}:
1
2
3
```

5.5.10. The Function Type

In Julia, every function is an object with its own unique type. `Function` is the abstract supertype of all functions. You can see this by inspecting the type of a function:

```
typeof(+)
```

```
typeof(+) (singleton type of function +, subtype of Function)
```

The output, `typeof(+)`, indicates that the function `+` has its own special type. This specific type is a subtype of the abstract `Function` type:

```
typeof(+) <: Function
```

```
true
```

This is true for any function, including ones you define:

```
function my_func(x)
    x + 1
end

typeof(my_func) <: Function
```

```
true
```

5.6. Scope

In projects of even modest complexity, it can be challenging to come up with unique identifiers for different functions or variables. **Scope** refers to the bounds for which an identifier is available. We will often talk about the **local scope** that's inside some expression that creates a narrowly-defined scope (such as a function or `let` or `module` block) or the **global scope** which is the top level scope that contains everything else inside of it. Here are a few examples to demonstrate scope.

```
i = 1
let
    j = 3
    i + j
end
```

(1)
(2)
(3)

- (1) `i` is defined in the global scope and would be available to other inner scopes.
- (2) The `let ... end` block creates a local scope which inherits the defined global scope definitions.
- (3) `j` is only defined in the local scope created by the `let` block.

4

In fact, if we try to use `j` outside of the scope defined above we will get an error¹⁵.

```
j
```

```
UndefVarError: UndefVarError(:j, 0x0000000000009799, Main.Notebook)
UndefVarError: `j` not defined in `Main.Notebook`
Suggestion: check for spelling errors or missing imports.
Stacktrace:
 [1] eval(m::Module, e::Any)
   @ Core ./boot.jl:489
 [2] (::QuartoNotebookWorker.var"#10#11"{Module, Expr})()
```

¹⁵Note that this error **stack trace** is unfortunately much messier due to it running in a Quarto notebook environment. When running scripts or in a REPL, your stack traces should look much simpler and easier to follow the error's lineage.

5. Elements of Programming

```
... (83 more lines omitted)
```

Tip

let blocks are a great way to organize your code in bite-sized chunks or to be able to re-use common variable names without worrying about conflict. Here's an example of using let blocks to:

1. Perform intermediate calculations without fear of returning a partially modified variable
2. Re-use common variable names

```
bonds = let
    df = CSV.read("bonds.csv", DataFrame)
    df.issuer = lookup_issuer(df.CUSIP)
    df
end

mortgages = let
    df = CSV.read("mortgages.csv", DataFrame)
    df.issuer = lookup_issuer(df.CUSIP)
    df
end
```

If we were running this interactively (e.g. step-by-step in VS Code, the REPL, or notebooks) then these two code blocks will run completely and will run separately. The short, descriptive name df is reused, but there's no chance of conflict. We also can't easily run the block of code (let ... end) and get a partially evaluated result (e.g. getting the dataframe before it has been appropriately modified to add the issuer column).

Here is an example with functions:

```
x = 2
base = 10
foo() = base^x
foo(x) = base^x
foo(x, base) = base^x
foo(), foo(4), foo(4, 4)
```

- ① Both base and x are inherited from the global scope.
- ② x is based on the local scope from the function's arguments and base is inherited from the global scope.
- ③ Both base and x are defined in the local scope via the function's arguments.

(100, 10000, 256)

In Julia, it's always best to explicitly pass arguments to functions rather than relying on them coming from an inherited scope. This is more straight-forward and easier to reason about and it also allows Julia to optimize the function to run faster because all relevant variables coming from outside the function are defined at the function's entry point (the arguments).

5.6.1. Modules and Namespaces

Modules are ways to encapsulate related functionality together. Another benefit is that the variables inside the module don't "pollute" the **namespace** of your current scope. Here's an example:

```
module Shape (1)

    struct Triangle{T}
        base::T
        height::T
    end

    function area(t::Triangle) (2)
        return t.base * t.height / 2
    end
end

t = Shape.Triangle(4, 2) (3)
area = Shape.area(t) (4)
```

- (1) `module` defines an encapsulated block of code which is anchored to the namespace `Shape`
- (2) Here, `area` is a *function* defined within the `Shape` module.
- (3) Outside of `Shape` module, we can access the definitions inside via the `Module.identifier` syntax.
- (4) Here, `area` is a *variable* in our global scope that *does not* conflict with the `area` defined within the `Shape` module. If `Shape.area` were not within a module then when we said `area = ...` we would have reassigned `area` to no longer refer to the function and instead would refer to the `area` of our triangle.

4.0

i Note

Summarizing related terminology:

- A **module** is a block of code such as `module MySimulation ... end`
- A **package** is a module that has a specific set of files and associated metadata. Essentially, it's a module with a `Project.toml` file that has a name and unique identifier listed, and a file in a `src/` directory called `MySimulation.jl`
 - **Library** is just another name for a package, and the most common context this comes up is when talking about the packages that are bundled with Julia itself called the **standard library** (`stdlib`).

5. Elements of Programming

6. Functional Abstractions

CHAPTER AUTHORED BY: ALEC LOUDENBACK

The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise. - Edsger Dijkstra (1972)

6.1. Chapter Overview

We demonstrate different approaches to a problem, gradually introducing more reusable or general techniques. These allow you to construct sophisticated models while maintaining consistency and simplicity. We cover imperative programming, functional programming, and recursion.

6.2. Introduction

This chapter centers around a simple task: calculate the present value of a single fixed, risk-free, coupon-paying bond under two different interest rate environments. The focus is on describing different approaches, not adding complexity (no credit spreads, settlement timing, etc.).

Mathematically, the problem is to determine the Present Value, where:

$$\text{Present Value} = \sum \text{Cashflow}_t \times \text{Discount Factor}_t$$

Where

$$\text{Discount Factor}_t = \prod^t \frac{1}{1 + \text{Discount Rate}_i}$$

```
| cf_bond = [10, 10, 10, 10, 110];  
| rate = [0.05, 0.06, 0.05, 0.04, 0.05];
```

①

- ① The rates are the one year forward rates for time 0, 1, 2, etc.

We will focus on this first discount vector, and introduce more scenarios later in the chapter.

We will repeatedly solve the same problem before extending it to more examples. It may feel repetitive, but the goal is to highlight variations in approach rather than the problem itself.

6.3. Imperative Style

One of the most familiar programming styles is **imperative** (or **procedural**), where we provide explicit, step-by-step instructions to the computer. The programmer defines the data involved and how it moves through the program one step at a time. It commonly uses loops to perform tasks repeatedly or across a set of data. The program's **state** (assignment and logic of variables) is defined and managed by the programmer explicitly.

Here's an imperative style of calculating the present value of the bond.

```
let
    pv = 0.0
    discount = 1.0
    for i in 1:length(cf_bond)
        discount = discount / (1 + rate[i])
        pv = pv + discount * cf_bond[i]
    end
    pv
end
```

- ① Declare variables to keep track of the discount rate and running total for the present value `pv`
- ② Loop over the length of the cashflow vector.
- ③ At each step of the loop, look up (via index `i`), update the discount factor to account for the prevailing rate, and add the discounted cashflow to the running total present value.

121.48888490821489

This style is simple, digestible, and clear. If we were performing the calculation by hand, it would likely follow a pattern very similar to this. Look up the first cashflow and discount rate, compute a discount factor, and subtotal the value. Repeat for the next set of values.

6.3.1. Iterators

Note that in the prior code example we defined an index variable `i` and had to manually define the range over which it would operate (1 through the `length` of the bond's cashflow vector). A couple of reasons this could be sub-optimal:

1. Julia arrays are 1-indexed by default. Some custom arrays (e.g., via `OffsetArrays.jl`) can use different starting indices: using `eachindex` avoids hard-coding index ranges and works for arrays with arbitrary axes.
2. We manually perform the lookup of the values within each iteration.

We can solve the first one (partially) by letting Julia return an iterable set of values corresponding to the indices of the `cf_bond` vector. This is an example of an **iterator** which is an object upon which we can repeatedly ask for the next value until it tells us to stop.

By using `eachindex` we can get the indices of the vector since Julia already knows what they are:

```
eachindex(cf_bond) # an efficient index iterator
Base.OneTo(5)
```

Lazy Programming

Lazy evaluation: the result, `Base.OneTo(5)`, returns an efficient iterator over valid indices (e.g., `Base.OneTo(5)` for a `Vector`). Iterators represent sequences without allocating per-element containers. You can traverse them directly, or materialize them if needed with `collect`.

An analogy is that we can write the “set of all numbers from 1 to 100” without writing out each of the 100 numbers, but we are referring to the same thing. In practice, this means we can iterate over scenario paths, coupon dates, or bucketized tenors without allocating temporary arrays, which keeps the garbage collector quiet during valuation runs.

An example of operating on a lazy iterator, is that we could find the largest index:

```
maximum(eachindex(cf_bond)) # operates without materializing all the
                             ↳ indices
5
```

The point is if we have an object that *represents* a set, we need not actually enumerate each element of the set to interact with it.

We can fully instantiate an iterator with `collect`

```
collect(eachindex(cf_bond)) # materialize indices as a Vector
5-element Vector{Int64}:
1
2
3
4
5
```

Laziness is generally a good thing in programming because sometimes it can be computationally or memory expensive to fully instantiate the collection of interest (this will be discussed further in Chapter 9).

And when used in context:

```
let
    pv = 0.0
    discount = 1.0

    for i in eachindex(cf_bond)
        discount = discount / (1 + rate[i])
        pv = pv + discount * cf_bond[i]
    end
    pv
end
```

121.48888490821489

Here Julia gave us the index associated with the bond cashflows, but we are still looking up the values (why not just ask for the values instead of their index?) as well as assuming that the indices are the same for the discount rates.

We can get the value and the associated index with `enumerate`:

6. Functional Abstractions

```
collect(enumerate(cf_bond))  
  
5-element Vector{Tuple{Int64, Int64}}:  
(1, 10)  
(2, 10)  
(3, 10)  
(4, 10)  
(5, 110)
```

This would allow us to skip the step of needing to look up the bond's cashflows. However, we can go even further by just asking for the values associated with both collections. With `zip` (named because it's sort of like zipping up two collections together), we get an iterator that provides the values of the underlying collections:

```
collect(zip(cf_bond, rate))  
  
5-element Vector{Tuple{Int64, Float64}}:  
(10, 0.05)  
(10, 0.06)  
(10, 0.05)  
(10, 0.04)  
(110, 0.05)
```

This provides the simplest implementation of the imperative approaches:

```
let  
    pv = 0.0  
    discount = 1.0  
  
    for (cf, r) in zip(cf_bond, rate)  
        discount = discount / (1 + r)  
        pv = pv + discount * cf  
    end  
    pv  
end
```

121.48888490821489

The primary downsides to iterative approaches are:

1. Needing to keep track of state is fine in simple cases, but can quickly become difficult to reason about and error prone as the number and complexity of variables grows.
2. Program flow is explicitly stated, leaving fewer places that the compiler can automatically optimize or parallelize.

💡 Tip

In the imperative style, we mentioned needing to explicitly handle program state. In general, it's advisable to minimize as many temporary state variables as possible - more mutability tends to produce more complex, difficult to maintain code. An example of maintaining state in the examples above is keeping track of the current index as well as interim `pv` and `discount` variables.

Modifying values is often avoidable by restructuring the logic, using functional techniques, or finding the right abstractions. However, sometimes for performance reasons, clarity, or expediency you may find modifying state to be the preferred option and that's okay.

6.4. Functional Techniques and Terminology

Functional programming is a paradigm which attempts to minimize state via composing functions together.

Table 6.1 introduces a set of core functional methods to familiarize yourself with. Note that anonymous functions (Section 5.5.6) are used frequently to define intermediary steps.

Table 6.1: Important Functional Methods.

Function	Description	Example
<code>map(f, v)</code>	Apply function <i>f</i> to each element of the collection <i>v</i> .	<pre>map(x->x^2, [1, 3, 5]) # [1, 9, 25]</pre>
<code>reduce(op, v)</code>	Apply binary <i>op</i> to pairs of values, reducing the dimension of the collection <i>v</i> .	<pre>reduce(*, [1, 3, 5]) # 15</pre> <p>Has a couple of important, optional keyword arguments to note (which also apply to other variants of <code>reduce</code> below):</p> <ul style="list-style-type: none"> • <code>init</code> defines the identity element (e.g. the initial value of <code>+</code> and <code>*</code> is 0 and 1 respectively) • <code>dims</code> defines which dimension to reduce across (if the dimension of <i>v</i> is more than one).
<code>mapreduce(f, op, v)</code>	Maps <i>f</i> over collection <i>v</i> and returns a reduced result using <i>op</i> .	<pre>mapreduce(x->x^2, *, [1, 3, 5]) # 225</pre>
<code>foldl(op, v), or foldlr(op, v)</code>	Like <code>reduce</code> , but applies <i>op</i> from left to right (<code>foldl</code>) or right to left (<code>foldr</code>). Also has <code>mapfoldl</code> and <code>mapfoldr</code> versions.	<pre>foldl(*, [1, 3, 5]) # 15</pre>

Function	Description	Example
<code>accumulate(op, v)</code>	Apply op along v, creating a vector with the cumulative result.	<pre>accumulate(+, [1, 3, 5]) # [1, 4, 9]</pre>
<code>filter(f, v)</code>	Apply f along v and return a copy of v with elements where f is true.	<pre>filter(>=(3), [1, 3, 5]) # [3, 5]</pre>

6. Functional Abstractions

This paradigm is very powerful in a few ways:

1. It provides a language for talking about what a computation is doing. Instead of “looping over a collection called `portfolio` and calling a `value` function” we can more concisely refer to this as `mapreduce(value, +, portfolio)`.
2. Often you are forced to think about the design of the program more deeply, recognizing the core calculations and data used within the model.
3. The compiler is free to apply more optimizations. For example, with `reduce`, the compiler could optimize the calculation since the operation is assumed to be associative.
4. The lack of mutable state.

Let’s build a version of the present value calculation using the functional building blocks described above. In practice these patterns show up when aggregating P&L across books, summing hedges across currencies, or transforming raw trades into risk factors. We will work up to the bond example by discussing the core building blocks—`map`, `accumulate`, `reduce`—and culminate in combining them with `mapreduce`.

6.4.1. `map`

`map` is so named for the mathematical concept of mapping an input to an output. Here, it’s effectively the same thing. We take a collection and use the given function to calculate an output. The size of the output equals the size of the input.

First, we will use `map` to compute the one-period discount factors:

```
map(x -> 1 / (1 + x), rate)
```

```
5-element Vector{Float64}:
0.9523809523809523
0.9433962264150942
0.9523809523809523
0.9615384615384615
0.9523809523809523
```

`map` transforms the `rate` collection by applying the anonymous function `x → 1 / (1 + x)`, which is the single period discount factor used in discount curve construction. This operation is conveyed visually in Figure 6.1.

Tip

`map` is an absolute workhorse of a function and the authors recommend using it liberally within your code. We find ourselves using `map` frequently, usually avoiding defining an explicit loop (unless we are modifying some existing collection).

`map` would likely be a better tool for a loop like this:

```
output = []
for x in collection
    result = # ... do stuff ...
    push!(output,result)
end
output
```

Instead, `map` simplifies this to:

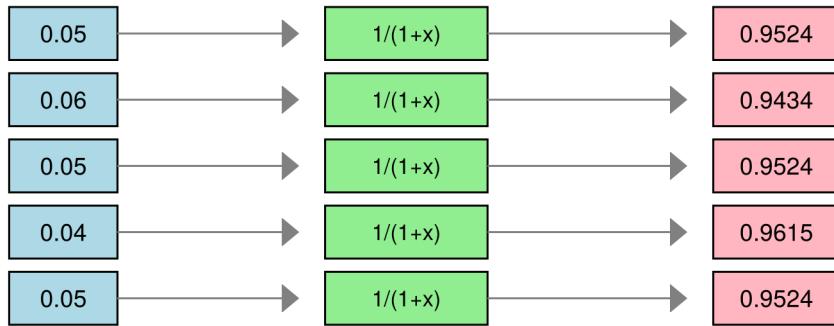


Figure 6.1.: A diagram showing that `map` creates a new collection mirroring the old one, after applying the given function to each element in the original collection.

```
map(collection) do x
    # ... do stuff
end
```

Not only does this have the advantage of being clearer, more concise, and less work, it also lets Julia infer the output type of your computation so you don't have to worry about the type of output.

6.4.2. accumulate

`accumulate` takes an operation and a collection and returns a collection where each element is the cumulative result of applying the operation from the first element to the current one. For example, to calculate the cumulative product of the one-period discount factors:

```
let
    rates =
        accumulate(*, map(x -> 1 / (1 + x), rate))
end
```

```
5-element Vector{Float64}:
0.9523809523809523
0.898472596585804
0.8556881872245752
0.822777103100553
```

6. Functional Abstractions

0.7835972410481457

This results in a vector of the cumulative discount factors for each point in time corresponding to the given cashflows.

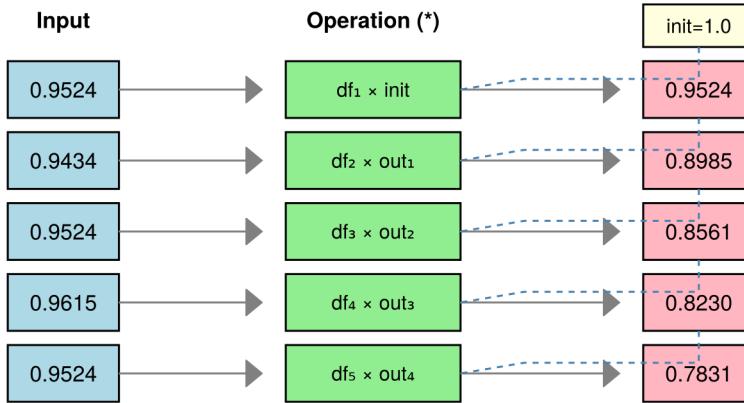


Figure 6.2.: A diagram showing that `accumulate` creates a new collection where each element is the cumulative result of applying the given operation to all previous elements.

Note

For `accumulate` and `reduce`, an important, optional value is the `init` (an optional keyword argument), which is the initial value to start the accumulation or reduction. For common operations this **identity element** is already predefined. For example, for `+` the identity is `0` while for `*` it is `1`. The identity element `e` is the one where for a given binary operation \odot , that $x \odot e = x$.

Another example is string concatenation. In Julia, two strings are concatenated with `*` (like in mathematics, $a * b$ is also written as ab). The identity element for strings where the binary operation $\odot = *$ is `" "`. For example:

```
accumulate(*, ["a", "b", "c"], init="")
```

```
3-element Vector{String}:
 "a"
 "ab"
 "abc"
```

*This is a taste of a branch of mathematics known as Category Theory, a very rich subject but largely beyond the immediate scope of this book. The category theoretical term for sets of elements that work with the binary operator and identity elements as described above is a **monoid**. There will not be a quiz on this trivia.*

6.4.3. reduce

reduce takes an operation and a collection and applies the operation repeatedly — combining elements pairwise, carrying forward the intermediate result — until there is only a single value left.

For example, we start with the calculation of the vector of discounted cashflows

```
dfs = accumulate(*, map(x -> 1 / (1 + x), rate))
discounted_cfs = map(*, cf_bond, dfs)
```

```
5-element Vector{Float64}:
9.523809523809524
8.98472596585804
8.556881872245752
8.22777103100553
86.19569651529602
```

Then we can sum them with reduce:

```
reduce(+, discounted_cfs)
```

```
121.48888490821487
```

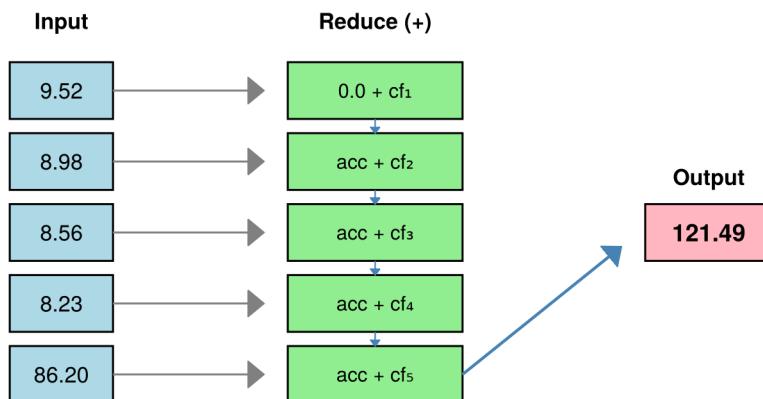


Figure 6.3.: A diagram showing how `reduce` applies the given operation to pairs of elements, ultimately reducing the collection to a single value.

6. Functional Abstractions

6.4.4. mapreduce

We can combine `map`, `accumulate` and `reduce` to concisely calculate the present value in a functional style using `mapreduce`. This calculates the discount factors, applies them to the cashflows with `map`, and sums the result with a reduction:

```
dfs = accumulate(*, map(x -> 1 / (1 + x), rate))  
      mapreduce(*, +, cf_bond, dfs)
```

(1)

(2)

- ① Multiplicatively accumulate a discount factor derived from the rate vector.
- ② Multiply the discount factor and bond cashflows (map the multiplication), then sum the result (additive reduce).

121.48888490821487

Contrast this example with the earlier imperative styles:

- This functional approach is more concise.
- The functions used are more descriptive and obvious (once familiar with them, of course!).
- There is no state that the user/programmer keeps track of.
- The compiler is able to potentially optimize the code, as it can deduce that certain operations are associative.

This completes the example of using a functional approach to determine the present value of bond cashflows.

6.4.5. filter

For completeness, we will also cover `filter` even though it's not necessary for the bond cashflow example.

`filter` does what you might think - filter a collection based on some criterion that can be determined as true or false.

For example filtering out even numbers using the `isodd` function:

```
filter(isodd, 1:6)
```

```
3-element Vector{Int64}:  
1  
3  
5
```

Or filtering out things that don't match a criterion:

```
filter(x -> x != 5, 1:6)
```

```
5-element Vector{Int64}:  
1  
2  
3  
4  
6
```

While we didn't need `filter` to calculate a bond's present value in the example above, one can imagine how you may want to filter dates that a bond might pay a cashflow, say last day of a quarter:

```

using Dates
let d = Date(2024, 01, 01)
    filter(d -> lastdayofquarter(d) == d, d:Day(1):lastdayofyear(d))
end

4-element Vector{Date}:
2024-03-31
2024-06-30
2024-09-30
2024-12-31

```

6.4.6. More Tips on Functional Styles

6.4.6.1. do Syntax for Function Arguments

In more complex situations such as with multiple collections or multi-line logic, there is a clearer syntax that is often used. `do` is a reserved keyword in Julia that creates an anonymous function and passes its arguments to a function like `map`. For example, this (terrible) code which decides if a number is prime. The anonymous function requires a `begin` block since the logic of the function is extended into multiple lines.

```

map(x -> begin
    if x == 1
        "not prime"
    elseif x == 2
        "prime"
    elseif x == 3
        "prime"
    elseif x > 4
        "probably not prime"
    end
end,
[1, 2, 3, 10]
)

```

```

4-element Vector{String}:
"not prime"
"prime"
"prime"
"probably not prime"

```

This can be written more cleanly with the `do` syntax:

```

map([1, 2, 3, 10]) do x
    if x == 1
        "not prime"
    elseif x == 2
        "prime"
    elseif x == 3
        "prime"
    elseif x > 4
        "probably not prime"
    end

```

6. Functional Abstractions

```
| end  
  
4-element Vector{String}:  
"not prime"  
"prime"  
"prime"  
"probably not prime"
```

6.4.6.2. Multiple Collections

`map` and the other functional operators discussed in this section can take multiple arguments. This is convenient if you have multiple arguments to a function:

```
| discounts = [0.9, 0.81, 0.73]  
cashflows = [10, 10, 10]  
  
map((d, c) -> d * c, discounts, cashflows)
```

```
3-element Vector{Float64}:  
9.0  
8.10000000000001  
7.3
```

Or an example with the `do` syntax:

```
| map(discounts, cashflows) do d, c  
    d * c  
end  
  
3-element Vector{Float64}:  
9.0  
8.10000000000001  
7.3
```

6.4.6.3. Using More Functions

At the risk of sounding obvious, an easy way to make the program more “functional” is to simply use more functions. Do this one thing and it will improve the model’s organization, maintainability, and reduce bugs!

Take the example from earlier:

```
| pv = 0.0  
discount = 1.0  
  
for (cf,r) in zip(cf_bond, rate)  
    discount = discount / (1 + r)  
    pv = pv + discount * cf  
end  
pv
```

We can easily turn this code into a function so that it can operate on data beyond the single pair of `cf_bond` and `rate` previously defined:

```

function pv(rates,cashflows)
    pv = 0.0
    discount = 1.0

    for (r,cf) in zip(rates, cashflows)
        discount = discount / (1 + r)
        pv = pv + discount * cf
    end
    pv
end

```

①

- ① Here, `cf_bond` and `rate` would refer to whatever was passed as arguments to the function instead of any globally defined values.

Now we could use this definition of `pv` on other instances of `rates` and `cashflows`.

6.4.6.4. Mixing Functional And Imperative Styles

One of the best things about Julia is how natural it can be to mix different styles. Sometimes the best approach is a mix of both. That's one of the benefits of Julia: use the style that's most natural to the problem.

Flexibility and the Lisp Curse

Lisp ("list processing") is another, much older language than Julia (created in the 1950s!). One of its claims to fame is how flexible and powerful the tools are within the language to build upon. There's a couple aspects of this curse that we wish to describe because we can learn from it while Julia is still a relatively young language.

Part of the "curse" is that: because there's so much freedom in what can be expressed in the language, there's not an obvious "best" way of doing things. This can lead to decision paralysis where you are trying to over-analyze what's the best way to write part of your code. Our advice: *don't worry about it!* A working implementation of something is better than an over-optimized idea.

The other part of the "curse" is that it is relatively easy to implement so many things from the building blocks that Julia provides and compose them together to do what you want. This has a downside because the general approach to packages is smaller, standalone pieces that you compose as needed. For example, consider Python's Pandas library, upon which Python's data science community was built. It came bundled with a CSV reader, Excel reader, Database reader, DataFrame type, visualization library, and statistical functions. In Julia, each of those are separate packages that specialize for the respective topics. This is advantageous in that they can progress independently from one another, you don't have to include functionality that you don't need, and you can mix and match libraries depending on your preference.

6.5. Array-Oriented Styles

Another paradigm is **array-oriented**, which is a style that relies heavily on putting similar data into arrays and operating on the entire array at the same time (as opposed to going element-by-element).

Array-oriented programming is practiced in two main contexts:

6. Functional Abstractions

1. GPU programming
2. Python numerical computing

The former because GPUs want large blocks of similar data to operate in parallel. The latter is because native Python is too slow for many modeling problems so libraries like NumPy, SciPy, and tensor libraries utilize C++ (or similar) libraries for users to call out to.

Array-oriented programming is not always natural for financial and actuarial applications. Differences in behavior or timing of underlying cashflows can make a set of otherwise similar products difficult to capture in nicely gridded arrays. Nonetheless, certain applications (scenario generation, some valuation routines) fit very naturally into this paradigm. Furthermore, for those that work well it's often a great way to extract additional performance due to the parallelization offered via CPU or GPU array programming.

Table 6.2 shows the bond present value example in this style.

Table 6.2.: The two code examples demonstrate the same logic using Julia and NumPy (Python's most popular array package). Julia's broadcasting facilitates an array-oriented style, similar to the approach that would be used with Python's NumPy.

Julia	Python (NumPy)
<pre>cf_bond = [10, 10, 10, 10, 110] rate = [0.05, 0.06, 0.05, 0.04, 0.05] discount_factors = cumprod(1 ./ (1 .+ rate)) result = sum(cf_bond .* discount_factors)</pre>	<pre>import numpy as np cf_bond = np.array([10, 10, 10, 10, 110]) rate = np.array([0.05, 0.06, 0.05, 0.04, 0.05]) discount_factors = np.cumprod(1 / (1 + rate)) result = np.sum(cf_bond * discount_factors)</pre>

6. Functional Abstractions

The downsides to this style are:

1. Sometimes it is unnatural because of non-uniformity of the data we are working with. For example if the length of the cashflows were shorter than the discount rates, we would have to perform intermediate steps to shorten or lengthen arrays in order to get them to be the same size.
2. A good bit of runtime performance is lost because the computer needs to allocate and fill many intermediate arrays (note how in Table 6.2, the `discount_factors` needs to instantiate an entirely new vector even though it's only temporarily used). See more on allocations in Chapter 9.

6.6. Recursion

A **recursive function** is a pattern where current steps are defined in a way that depends on previous steps. Typically, an explicit starting condition is also required to be specified.

The Fibonacci sequence is a classic example of a recursive algorithm, with the starting conditions of n specified for the first two steps:

$$F(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F(n - 1) + F(n - 2), & \text{if } n > 1 \end{cases}$$

In code, this translates into a function definition that refers to itself:

```
function fibonacci(n)
    if n == 0
        return 0
    elseif n == 1
        return 1
    else
        return fibonacci(n-1) + fibonacci(n-2)
    end
end
```

How could a recursive pattern be defined for valuing our bond? A possible pattern is defining the present value to be the discounted value of:

- the current period's cashflow, *plus*
- the accumulated cashflows up to that point in time

Here's how that might be defined:

```
function
    ↵ pv_recursive(rates,cashflows,accumulated_value=0.0,discount_factor=1.0) ①
    if isempty(cashflows)
        return accumulated_value
    else
        discount_factor = discount_factor / (1+first(rates)) ②
        av = first(cashflows) * discount_factor + accumulated_value ③
        remaining_rates = rates[begin+1:end]
        remaining_cfs = cashflows[begin+1:end]
        return pv_recursive(remaining_rates,remaining_cfs, av,discount_factor) ④
```

```
    end
end
```

- ① Add a terminating condition, that if we have no more cashflows then return the accumulated value.
- ② Decrement the discount factor as we step forward in time.
- ③ Take the prior accumulated value and add the first value in the given cashflows.
- ④ Pass the remaining subset of the cashflow vector, the running total, and the current discount factor to the next call of the recursive function.

```
pv_recursive (generic function with 3 methods)
```

And an example of its use:

```
| pv_recursive(rate,cf_bond)
```

```
121.48888490821489
```

The recursive pattern often works very nicely for simpler examples. However, more complex logic and conditionals can make this approach unwieldy. Nonetheless, attempting to distill the desired functionality into a single function can be a beneficial thought exercise.

6. Functional Abstractions

7. Data and Types

CHAPTER AUTHORED BY: ALEC LOUDENBACK, YUN-TIEN LEE

I am only one, but I am one. I can't do everything, but I can do something. The something I ought to do, I can do. And by the grace of God, I will - Edward Everett Hale (1902)

7.1. Chapter Overview

Using Julia's type system lets us express financial instruments and valuation logic cleanly and efficiently. In this chapter we will:

- model assets as concrete types within a simple hierarchy,
- define valuation functions using multiple dispatch,
- show how to aggregate portfolio value succinctly,
- compare this data-oriented approach with common object-oriented patterns.

7.2. Using Types to Value a Portfolio

We'll assemble the tools and terminology to value a portfolio of assets by leveraging Julia's type system. Using the constructs from the prior chapter, we can describe portfolio valuation as mapping a value function over assets and summing the results. We aim to compute total portfolio value by applying a valuation function to each asset and summing the individual valuations:

```
mapreduce(value,+,portfolio)
```

The challenge: how do we design an all-purpose value function? Assets in `portfolio` may be heterogeneous, so we need to define valuation semantics for different asset types. To reach our end goal, we need to:

1. Define the different kinds of assets within our portfolio
2. How the assets are to be valued.

We will accomplish this by utilizing data types.

7.3. Benefits of Using Types

As a preview of why we want to utilize types in our program, there are a number of benefits:

1. **Separate concerns.** For example, deciding how to value an option need not know how we value a bond. The code and associated logic is kept distinct which is easier to reason about and to test.
2. **Re-use code.** When a set of types within a hierarchy all share the same logic, then we can define the method at the highest relevant level and avoid writing the method for each possible type. In our simple example we won't get as much benefit here since the hierarchy is simple and the set of types small.

7. Data and Types

3. **Extensibility through dispatch.** By defining types for our assets, we can use multiple dispatch to define specialized behavior for each type. This allows us to write generic code that works with any asset type, and the Julia compiler will automatically select the appropriate method based on the type of the asset at runtime. This is a powerful feature that enables extensibility and modularity in our code.
4. **Improve readability and clarity.** By defining types for our assets, we make our code more expressive and self-documenting. The types provide a clear indication of what kind of data we are working with, making it easier for other developers (or ourselves in the future) to understand and maintain the codebase.
5. **Enable type safety.** By specifying the expected types for function arguments and return values, we can catch type-related errors at compile time rather than at runtime. This helps prevent bugs and makes our code more robust.

With these benefits in mind, let's start by defining the types for our assets. We'll create an abstract type called Asset that will serve as the parent type for all our asset types. If you haven't read it already, Section 5.4.7 is a good reference for details on types at the language level (this section is focused on organization and building up the abstracted valuation process).

7.4. Defining Types for Portfolio Valuation

We will define two types of assets in this simplified universe:

- Cash
- Risk-free Bonds (coupon and zero-coupon varieties)

To do the valuation of these, we need some economic parameters as well: risk free rates for discounting.

Here's the outline of what follows to get an understanding of types, type hierarchy, and multiple dispatch.

1. Define the Cash and Bond types.
2. Define the most basic economic parameter set.
3. Define the value functions for Cash and Bonds.

```
## Data type definitions
abstract type AbstractAsset end (1)

struct Cash <: AbstractAsset
    balance::Float64
end (3)

abstract type AbstractBond <: AbstractAsset end (2)

struct CouponBond <: AbstractBond
    par::Float64
    coupon::Float64
    tenor::Int
end (3)

struct ZeroCouponBond <: AbstractBond
    par::Float64
    tenor::Int
end
```

- ① General convention is to name abstract types beginning with Abstract...
- ② There can exist an abstract type which is a subtype of another abstract type.
- ③ We define concrete data types (structs) with the fields necessary for valuing those assets.

Now to define the economic parameters:

```
struct EconomicAssumptions{T}
    riskfree::T
end
```

This is a parametric type because later on we will vary what objects we use for `riskfree`. For now, we will use simple scalar values, like in this potential scenario:

```
econ_baseline = EconomicAssumptions(0.05)

EconomicAssumptions{Float64}(0.05)
```

Now on to defining the valuation for `Cash` and `AbstractBonds`. `Cash` is always equal to its balance:

```
value(asset::Cash, econ::EconomicAssumptions) = asset.balance

value (generic function with 1 method)
```

Risk free bonds are the discounted present value of the riskless cashflows. We first define a method that generically operates on any fixed bond, all that's left to do is for different types of bonds to define how much cashflow occurs at the given point in time by defining `cashflow` for the associated type.

```
function value(asset::AbstractBond, r::Float64)                                ②
    discount_factor = 1.0
    pv = 0.0
    for t in 1:asset.tenor
        discount_factor /= (1 + r)
        pv += discount_factor * cashflow(asset, t)
    end
    return pv
end

function cashflow(bond::CouponBond, time)
    if time == bond.tenor
        (1 + bond.coupon) * bond.par
    else
        bond.coupon * bond.par
    end
end

function value(bond::ZeroCouponBond, r::Float64)                                ③
    return bond.par / (1 + r)^bond.tenor
end
```

① `x /= y`, `x += y`, etc. are shorthand ways to write `x = x / y` or `x = x + y`

② `value` is defined for `AbstractBonds` in general...

③ ... and then more specifically for `ZeroCouponBonds`. This will be explained when discussing "dispatch" below.

`value (generic function with 3 methods)`

7. Data and Types

7.4.1. Dispatch

When a function is called, the computer has to decide which method to use. In the example above, when we want to value a `ZeroCouponBond`, does the `value(asset::AbstractBond, r)` or `value(bond::ZeroCouponBond, r)` version get used?

Dispatch is the process of determining which method to use, and the rule is that *the most specific defined method gets used*. In this case, that means that even though our `ZeroCouponBond` is an `AbstractBond`, the routine that will be used is the most specific `value(bond::ZeroCouponBond, r)`.

Already, this is a powerful tool to simplify our code. Imagine the alternative of a long chain of conditional statements trying to find the right logic to use:

```
# don't do this!
function value(asset,r)
    if asset.type == "ZeroCouponBond"
        # special code for Zero coupon bonds
        #
    elseif asset.type == "ParBond"
        # special code for Par bonds
        #
    elseif asset.type == "AmortizingBond"
        # special code for Amortizing Bonds
        #
    else
        # here define the generic AbstractBond logic
    end
end
```

With dispatch, the compiler does this lookup for us, and more efficiently than enumerating a list of possible codepaths.

In contrast with the prior code example, we didn't have a long chain of `if` statements, and instead are letting the types themselves dictate which functions are relevant and will be called. We provided a generic `value` function for any `AbstractBond` which loops through time, and a specialized one for `ZeroCouponBond`.

We *could* have simply used the generic `AbstractBond` method for the `ZeroCouponBond` as well. To do so, we would only need to define its `cashflow` method:

```
# An alternative, but less efficient, implementation
function cashflow(bond::ZeroCouponBond, time)
    # A zero-coupon bond only pays its par value at the very end
    if time == bond.tenor
        return bond.par
    else
        return 0.0
    end
end
```

With this method, the generic `value` function would have worked correctly, looping from `t=1` to `t=tenor` and finding only a single non-zero cashflow to discount.

However, this is inefficient. We know there is a more direct, closed-form formula to value a zero-coupon bond: $PV = \frac{\text{Par}}{(1+r)^{\text{tenor}}}$. There's no need to loop through intermediate years where the cashflow is zero.

This is the power of dispatch. By defining a **more specific** method, `value(bond::ZeroCouponBond, r :: Float64)`, we are telling Julia: "When you have a `ZeroCouponBond`, use this highly efficient, direct formula. For any *other* kind of `AbstractBond`, you can fall back on the generic looping version." Dispatch ensures that the most specific, and in this case most performant, implementation is automatically chosen. This allows you to build a system that is both general and extensible, while also being highly optimized for the most common and simple cases.

7.4.1.1. Integrating Economic Assumptions

Despite the definitions above, the following will error because we haven't defined a method for `value` which takes as its second argument a type of `EconomicAssumptions`:

```
value(ZeroCouponBond(100.0, 5), econ_baseline)
```

Let's fix that by defining a method which takes the economic assumption type and just relays the relevant risk free rate to the `value` methods already defined (which take an `AbstractBond` and a scalar `r`).

```
value(bond::AbstractBond, econ::EconomicAssumptions) =
    ↳ value(bond, econ.riskfree)
```

`value` (generic function with 4 methods)

Now this following works:

```
value(ZeroCouponBond(100.0, 5), econ_baseline)
```

78.35261664684589

Here's an example of how this would be used:

```
portfolio = [
    Cash(50.0),
    CouponBond(100.0, 0.05, 5),
    ZeroCouponBond(100.0, 5),
]
map(asset-> value(asset, econ_baseline), portfolio)
```

3-element Vector{Float64}:

50.0
99.9999999999999
78.35261664684589

This is very close to the goal that we set out at the end of the section. We can complete it by reducing over the collection to sum up the value:

```
mapreduce(asset -> value(asset, econ_baseline), +, portfolio)
```

228.3526166468459

7. Data and Types

i Note

This code:

```
mapreduce(asset-> value(asset,econ_baseline), +, portfolio)
```

is more verbose than what we set out to do at the start (`mapreduce(value,+,portfolio)`) due to the two-argument `value` function requiring a second argument for the economic variables. This works well! However, there is a way to define it which avoids the anonymous function, which in some cases will end up needing to be compiled more frequently than you want it to. Sometimes we want a lightweight, okay-to-compile-on-the-fly function. Other times, we know it's something that will be passed around in compute-intensive parts of the code.

A technique in this situation is to define an object that "locks in" one of the arguments but behaves like the anonymous version. There is a pair of types in the `Base` module, `Fix1` and `Fix2`, which represent partially-applied versions of the two-argument function `f`, with the first or second argument fixed to the value "x".

That is, `Base.Fix1(f, x)` behaves like `y->f(x, y)` and `Base.Fix2(f, x)` behaves like `y->f(y, x)`.

In the context of our valuation model, this would look like:

```
val = Base.Fix2(value,econ_baseline)
mapreduce(val,+,portfolio)
```

228.3526166468459

7.4.1.2. Multiple Dispatch

A more general concept is that of **multiple dispatch**, where the types of *all arguments* are used to determine which method to use. This is a very general paradigm, and in many ways is more extensible than traditional object oriented approaches, (more on that in Section 7.5). What if instead of a scalar interest rate value we wanted to instead pass an object that represented a term structure of interest rates?

Extending the example, we can use a time-varying risk free rate instead of a constant. For fun, let's say that the risk free rate has a sinusoidal pattern:

```
econ_sin = EconomicAssumptions(t -> 0.05 + sin(t) / 100)

EconomicAssumptions{var"#8#9"}(var"#8#9"())
```

Now `value` will not work, because we've only defined how `value` works on bonds if the given rate is a `Float64` type:

```
value(ZeroCouponBond(100.0, 5), econ_sin)

MethodError: MethodError(value, (ZeroCouponBond(100.0, 5), var"#8#9"()),
    ↵ 0x000000000000977d)
MethodError: no method matching value(::ZeroCouponBond, ::var"#8#9")
The function 'value' exists, but no method is defined for this combination of
    ↵ argument types.
```

```

Closest candidates are:
  value(::ZeroCouponBond, ::Float64)
    @ Main.Notebook ~/prog/julia-fin-book/type-abstractions.qmd:130
  value(::AbstractBond, ::EconomicAssumptions)
    @ Main.Notebook ~/prog/julia-fin-book/type-abstractions.qmd:204
  value(::AbstractBond, ::Float64)
    @ Main.Notebook ~/prog/julia-fin-book/type-abstractions.qmd:112
...
Stacktrace:
[1] value(bond::ZeroCouponBond, econ::EconomicAssumptions{var"#8#9"})
    @ Main.Notebook ~/prog/julia-fin-book/type-abstractions.qmd:204
[2] top-level scope
    @ ~/prog/julia-fin-book/type-abstractions.qmd:266

```

We can extend our methods to account for this:

```

function value(bond::ZeroCouponBond, r::Function)           ①
    return bond.par / (1 + r(bond.tenor)) ^ bond.tenor
end                                         ②

```

- ① The `r :: Function` constraint says use this method if `r` is any subtype of the (abstract) `Function` type.
- ② `r` is a function that we call with the time to get the zero coupon bond (a.k.a. spot) rate for the given timepoint.

```
value (generic function with 5 methods)
```

Now it works:

```
value(ZeroCouponBond(100.0, 5), econ_sin)
```

```
82.03058910862806
```

The important thing to note here is that the compiler is using the most specific method of the function (`value(bond :: ZeroCouponBond, r :: Function)`). Both the types of the arguments are influencing the decision of which method to use. To complete the example for `CouponBond`, we would need to define a similar method that calls `r(t)` at each coupon date within the valuation loop.

7.5. Object-Oriented Design

Object-oriented (OO) type systems use the analogy that various parts of the system are their own objects that encapsulate both data and behavior. Object oriented design is often one of the first computer programming abstractions introduced because it is very relatable¹, however there are a number of flaws in over-relying on OO patterns. Julia does not natively have traditional OO classes and types, but much of OO design can be emulated in Julia except for data inheritance.

i Note

For readers without background in OO programming, the main features of OO languages are:

¹"Many people who have no idea how a computer works find the idea of object-oriented programming quite natural. In contrast, many people who have experience with computers initially think there is something strange about object oriented systems." - David Robson, "Object Oriented Software Systems" in Byte Magazine (1981).

7. Data and Types

- Hierarchical type structures, which include concrete and abstract (often called classes instead of types).
- Sub-classes inherit both behavior *and* data (in Julia, subtypes only inherit behavior, not data).
- Functions that depend on the type of the object need to be ascribed to a single class and then can dispatch more specifically on the given argument's type.

We bring up object-oriented design for comparison's sake, but think that ultimately choosing a data-driven or functional design is better for financial modeling. Of course, many robust, well-used financial models have been built this way, but in our experience the abstractions become unnatural. Additionally, maintenance becomes unwieldy beyond simple examples. We'll now discuss some of the aspects of OO design and why the overuse of OO is not preferred.

7.5.1. Assigning Behavior

Needing to assign methods to a single class can lead to awkward design limitations - when multiple objects are involved in a computation, why dictate that only one of them "controls" the logic?

The value function is a good example of this. If we had to assign `value` to one of the objects involved, should it be the economic parameter object or the asset objects? The choice is not obvious at all. Isn't it the market (economic parameters) that determines the value? But then if `value` were to be a method wholly owned by the economic parameters, how could it possibly define in advance the valuation semantics of all types of assets? What if one wanted to extend the valuation to a new asset class? Downstream users or developers would need to modify the economic types to handle new assets they wanted to value. However, because the economic types were owned by an upstream package, they can't be extended this way.

This is an issue with traditional OO designs and that resolves itself so elegantly with multiple dispatch.

7.5.1.1. Example: The Expression Problem

A fundamental limitation of OOP is what's called the **Expression Problem**. The challenge (or problem) is that with OOP languages it is difficult to extend both datatypes *and* behavior. In the example that follows, we define types of insurance products with associated methods.

Here's the setup: we are modeling insurance contracts and someone has provided a nice library which we will call `Insurance.jl` and `pyInsurance` for a Julia and Python package. The package defines datatypes for Term and Whole Life insurance products, as well as a lot of utilities related to calculating premiums and reserves (i.e. performing valuations). Defining the functionality is straightforward enough in both languages/approaches:

Insurance.jl

A hypothetical package available for your use.

```
abstract type InsuranceProduct end

struct TermLife <: InsuranceProduct
    term::Int
    face_amount::Float64
```

```

    age::Int
end

struct WholeLife <: InsuranceProduct
    face_amount::Float64
    age::Int
end

# Calculate premium
premium(p::TermLife) = ...
premium(p::WholeLife) = ...

# Calculate reserves
reserve(p::TermLife, t::Int) = ...
reserve(p::WholeLife, t::Int) = ...

```

pyInsurance

A hypothetical package available for your use.

```

from abc import ABC, abstractmethod
from dataclasses import dataclass

class InsuranceProduct(ABC):
    @abstractmethod
    def calculate_premium(self) -> float:
        pass

    @abstractmethod
    def calculate_reserve(self, t: int) -> float:
        pass

@dataclass
class TermLife(InsuranceProduct):
    term: int
    face_amount: float
    age: int

    # Term-specific calculations
    def calculate_premium(self) -> float:
        return ...

    def calculate_reserve(self, t: int) -> float:
        return ...

@dataclass
class WholeLife(InsuranceProduct):
    face_amount: float
    age: int

    # WholeLife-specific calculations
    def calculate_premium(self) -> float:
        return ...

```

7. Data and Types

```
def calculate_reserve(self, t: int) -> float:  
    return ...
```

Now, say that we want to utilize this package and extend the behavior. Specifically, we want to add a Deferred Annuity type and add functionality (for all products) related to determining a cash surrender value.

We run into limitations with Python version. We can extend a new representation (dataclass), but adding new functionality (e.g. `cash_value`) requires modifying other classes which you may not own and for which the method may not apply.

Julia (Multiple Dispatch)

```
struct DeferredAnnuity <: InsuranceProduct  
    premium::Float64  
    deferral_period::Int  
    age::Int  
end  
  
# Implement existing methods  
premium(p::DeferredAnnuity) = ...  
reserve(p::DeferredAnnuity, t::Int) = ...  
  
# Adding new function  
cash_value(p::WholeLife, t::Int) = ...  
cash_value(p::DeferredAnnuity, t::Int) = ...
```

Python (Object-Oriented)

```
@dataclass  
class DeferredAnnuity(InsuranceProduct):  
    premium: float  
    deferral_period: int  
    age: int  
  
    def calculate_premium(self) -> float:  
        # Annuity-specific premium calc  
        return ...  
  
    def calculate_reserve(self, t: int) -> float:  
        # Annuity-specific reserve calc  
        return ...
```

There are workarounds to handle this, which include those listed in Table 7.1.

Table 7.1.: Workarounds for Object Oriented Programming restrictions. The object oriented paradigm does not allow for extension of **both** representation (data types) and behavior (methods).

Workaround to OOP Expression Problem	Concerns with Workaround
<p>Monkey-Patching You can dynamically inject the method into the library's class definition at runtime.</p> <pre># WARNING: This is generally considered bad practice from ultimate_insurance_models import WholeLife def _calculate_cash_value_for_wholelife(self, t: int) -> ↪ float: ↪ return ...</pre>	<ul style="list-style-type: none"> Overwriting Conflicts: If two different parts of a program try to patch the same method, the last one to run will overwrite the others. This can disable expected functionality in a way that is difficult to predict. Harder to Debug: Patching makes the code's runtime behavior different from its source code. This complicates debugging because the written code no longer represents what is actually happening. Upgrade Instability: A patch may break when the underlying code it modifies is updated. This creates a maintenance burden, as patches need to be re-validated and potentially re-written with each library upgrade. <pre># At the start of your app, "patch" the library's class WholeLife.calculate_cash_value = ↪ _calculate_cash_value_for_wholelife wl_policy = WholeLife(face_amount=500000, age=40) wl_policy.calculate_cash_value(t=10)</pre>

Workaround to OOP Expression Problem	Concerns with Workaround
<p>Subclassing You can inherit from the parent class to create your own custom version.</p> <pre>from ultimate_insurance_models import WholeLife class MyExtendedWholeLife(WholeLife): def calculate_cash_value(self, t: int) -> float: # Your brilliant logic return self.face_amount * 0.1 * t # You have to make sure you ONLY create your extended # version my_policy = MyExtendedWholeLife(face_amount=50000000, age=40)</pre>	<ul style="list-style-type: none"> Incomplete Coverage: The new functionality only exists on your subclass. Any code that creates an instance of the original parent class will produce an object that lacks your new methods. Breaks Polymorphism: It forces you to check an object's specific type before using the new functionality (e.g., using <code>isinstance</code>). This defeats the purpose of having a common interface and makes the code more complex and less robust. Doesn't Affect Object Creation: Functions within the original library will continue to create and return instances of the original parent class. You cannot alter this behavior, meaning objects created by the library will not have your added methods.

In Julia, functions are defined separately from data types. This allows you to add new functions to existing types—even those from external libraries—without altering their original code. Your new functionality works on all instances of the original type, avoiding both the conflicts of patching and the type-checking required by subclassing. Julia’s multiple dispatch facilitates a more general, data-oriented approach.

7.5.2. Inheritance

As seen in the prior example, in most OO implementations this hierarchy comes with inheriting both data *and* behavior. This is different from Julia where subtypes inherit behavior but not data from the parent type.

Inheriting the data tends to introduce a tight coupling between the parent and the child classes in OO systems. This tight coupling can lead to several issues, particularly as systems grow in complexity. For example, changes in the parent class can inadvertently affect the behavior of all its child classes, which can be problematic if these changes are not carefully managed. This is often referred to as the “fragile base class problem,” where base classes are delicate and changes to them can have widespread, unintended consequences.

Another issue with inheritance in OO design is the temptation to use it for code reuse, which can lead to inappropriate hierarchies. Developers might create deep inheritance structures just to reuse code, leading to a scenario where classes are not logically related but are forced into a hierarchy. This can make the system harder to understand and maintain.

7.5.2.1. Composition over Inheritance

To mitigate some of the problems associated with inheritance, there is a growing preference for *composition*. Composition involves creating objects that contain instances of other objects to achieve complex behaviors. This approach is more flexible than inheritance as it allows for the creation of more modular and reusable code.

To see why, consider a concrete problem: we want to model financial instruments that share *some* but not *all* characteristics. A municipal bond is a fixed-income instrument (it has a coupon and tenor), but it’s also a listed security (it has a CUSIP identifier). An interest rate swap has fixed-income characteristics on both legs, but isn’t a listed security at all. A listed equity option has a CUSIP but no fixed-income characteristics.

With inheritance, we’d be forced to choose a single parent for each type:

```
# The inheritance approach runs into trouble
abstract type Asset end
abstract type FixedIncomeAsset <: Asset end
abstract type ListedAsset <: Asset end

# But what is a MunicipalBond? It's both fixed income AND listed.
# We can only pick one parent:
struct MunicipalBond <: FixedIncomeAsset # Can't also be <: ListedAsset
    #
end
```

There’s no natural hierarchy here. A municipal bond isn’t “more” of a fixed-income instrument than a listed security—it’s both. Inheritance forces us to pick one lineage, and we lose the ability to share behavior along the other dimension.

Composition solves this by building types out of reusable components:

7. Data and Types

```
struct CUSIP
    code::String
end

struct FixedIncome
    coupon::Float64
    tenor::Int
end

struct FloatingRate
    spread::Float64
    tenor::Int
end
```

Now we can compose these building blocks into the instruments we need:

```
struct MunicipalBond      # Has both CUSIP and fixed-income characteristics
    cusip::CUSIP
    fi::FixedIncome
end

struct Swap                # Two legs, no CUSIP
    fixed_leg::FixedIncome
    float_leg::FloatingRate
end

struct ListedOption        # Has CUSIP, but no fixed-income component
    cusip::CUSIP
    strike::Float64
    expiry::Int
end

struct PrivatePlacement    # Fixed income, but not listed
    fi::FixedIncome
    counterparty::String
end
```

Note

A CUSIP (Committee on Uniform Security Identification Procedures) number is a unique nine-character alphanumeric code assigned to securities in the United States and Canada, used to identify them in clearing and settlement.

The key insight is **delegation**: we define behavior on the component types, then delegate to those components from the composite types.

```
# Macaulay duration for a fixed-income component
function duration(fi::FixedIncome, rate)
    # Simplified duration calculation
    pv_weighted_time = 0.0
    pv_total = 0.0
    for t in 1:fi.tenor
        cf = t == fi.tenor ? (1 + fi.coupon) : fi.coupon
        pv_weighted_time += cf * t
        pv_total += cf
    end
    return pv_weighted_time / pv_total
```

```

    pv = cf / (1 + rate)^t
    pv_weighted_time += t * pv
    pv_total += pv
end
return pv_weighted_time / pv_total
end

# Delegate: any type with a .fi field can compute duration
duration(asset::MunicipalBond, rate) = duration(asset.fi, rate)
duration(asset::PrivatePlacement, rate) = duration(asset.fi, rate)

duration (generic function with 3 methods)

```

Now we can use these:

```
muni = MunicipalBond(CUSIP("123456789"), FixedIncome(0.05, 10))
duration(muni, 0.04)
```

8.190898824083233

```
private = PrivatePlacement(FixedIncome(0.06, 5), "Acme Corp")
duration(private, 0.04)
```

4.490230868454794

The `MunicipalBond` and `PrivatePlacement` types share duration logic through their common `FixedIncome` component—not through a shared parent class. We could similarly define `lookup_trades(c :: CUSIP)` and delegate to it from `MunicipalBond` and `ListedOption`, sharing that behavior along an orthogonal dimension.

When is inheritance appropriate? Inheritance works well for genuine “is-a” relationships where a single hierarchy makes sense. A `CouponBond` really *is* a kind of `Bond`—there’s no ambiguity about where it belongs. The earlier portfolio valuation example in this chapter uses inheritance appropriately: `ZeroCouponBond <: AbstractBond <: AbstractAsset` is a clean hierarchy.

The rule of thumb: use inheritance when types form a natural tree. Use composition when types share characteristics along multiple independent dimensions—which is common in financial modeling, where instruments often combine features (listed vs. private, fixed vs. floating, secured vs. unsecured) in various combinations.

💡 Working with Nested Immutable Data

One practical consideration with composition: if your composite types are immutable (the default for `struct`), updating a nested field requires creating a new instance. The `Accessors.jl` package provides convenient syntax for this:

```
using Accessors
# Create a new MunicipalBond with a different coupon
updated_muni = @set muni.fi.coupon = 0.055
```

This returns a new `MunicipalBond` with the nested coupon changed, leaving the original unchanged.

7.6. Data-Oriented Design

Data-Oriented Programming (DOP), especially in a computational field like financial modeling, is an approach that prioritizes the data itself—its structure, its layout in memory, and how it's processed in bulk. This stands in contrast to Object-Oriented Programming, which prioritizes encapsulating data within objects and interacting with that data through the object's methods.

DOP separates the data from the behavior:

1. **Data is transparent and inert.** We define structures (like the `Cash` and `CouponBond` structs in our example) that are simple, transparent containers for information. Their job is to hold data, not to have complex internal logic.
2. **Behavior is handled by functions.** Logic is implemented in generic functions (like our `value` function) that operate on this data.

The portfolio valuation model we have built in this chapter is an example of data-oriented design. We created a collection of data—the `portfolio` array. We then used Julia's functions (`mapreduce`, and our own `value` function) to transform that data into a final result. We can easily add a completely new operation, say `calculate_duration(asset, econ_assumptions)`, without ever modifying the original `struct` definitions for our assets.

This approach is pertinent for financial modeling for several key reasons:

- **Flexibility:** Financial models require many different views of the same data. Today we need to value a portfolio. Tomorrow, we might need to calculate its credit risk, liquidity risk, or run a stress test. With a data-oriented approach, each new requirement is simply a new set of functions that we write to operate on the same underlying data structures. In a strict OO world, we might be forced to add a `.calculate_credit_risk()` method to every single asset class, which can become unwieldy.
- **Performance:** Financial computations often involve applying a single operation to millions or billions of items (e.g., valuing every asset in a large portfolio, running a Monte Carlo simulation with millions of paths). DOP allows the data to be laid out in memory in a way that is highly efficient for modern CPUs or GPUs to process (e.g., in contiguous arrays). By processing data in bulk, we leverage how computer hardware is designed to work, leading to significant performance gains over designs that require calling methods on individual objects one by one.
- **Simplicity and Scalability:** As a system grows, data-oriented designs can be easier to reason about. The “state” of the system is just the data itself. The logic is contained in pure functions that transform data. This avoids the complex webs of object relationships, inheritance hierarchies, and hidden state that can make large OO systems difficult to maintain and debug.

While object-oriented design patterns can be useful, for the performance-critical and mathematically intensive world of financial modeling, a data-oriented approach often proves to be a more natural, scalable, and efficient choice. It aligns perfectly with the core task: the transformation of data (market and instrument parameters) into insight (value, risk, etc.).

8. Higher Levels of Abstraction

CHAPTER AUTHORED BY: ALEC LOUDENBACK

“Simple things should be simple, complex things should be possible.” - Alan Kay (1970s)

8.1. Chapter Overview

Abstraction matters as a technique in its own right. This chapter focuses on code organization, interfaces, and reusable design patterns.

8.2. Introduction

In programming and modeling, as in mathematics, abstraction permits the definition of interchangeable components and patterns that can be reused. Abstraction is a selective ignorance - focusing on the aspects of the problem that are relevant, and ignoring the others. The last two chapters described what we might call “micro-level” abstractions: specific functions and types.

In this chapter, we zoom out and examine some principles that guide good model development, manifesting in architectural concerns such as how different parts of the code are organized, what parts of the program are considered ‘public’ versus ‘private’, and patterns themselves.

Chapter 5 described a number of tools that we can utilize as interfaces within our model. We use these tools that are provided by our programming language *in service of* the conceptual abstraction described above.

- Functions let us implement behavior without troubling ourselves with the low-level details.
- Data types provide a hierarchical structure to provide meaning to things, and to group those things together into more meaningful structures.
- Modules allow us to group related data and functions into cohesive namespaces that can be shared across different parts of a model.

8.3. Principles for Abstraction

Table 8.1 lists some principles that arise when developing a particular abstraction. Not all abstractions serve all of these purposes but generally fit one or more of them. These principles provide guidance for creating abstractions that are modular, reusable, and maintainable. By following these principles, developers can create financial models that are easier to understand, extend, and adapt to changing requirements.

Table 8.1.: Finding abstractions generally means finding patterns that fit into one of these principles.

Principle	What	Why	Example
Separation of Concerns	Divide the system into distinct parts, each addressing a separate concern.	Promote modularity and reduce coupling between components.	Separating data retrieval, data processing, and output generation steps in a process.
Encapsulation	Hide the internal details of a component and expose only a clean, well-defined set of functionality (interface).	Prevent other parts of the program from modifying internal data, making the system easier to understand and maintain.	Defining a type or module with well-defined behavior and responsibility.
Composability	Design simple components that can be combined to create more complex behaviors, rather than a single component that attempts to handle all behavior.	Promote reuse and allow for the components to be combined creatively.	Separate details about economic conditions into different types than contracts/instruments.
Generalization	Identify common patterns and create generic components that can be specialized as needed. Often this means identifying common behaviors that arise repeatedly in a model.	Avoid duplication and make the system more expressive and extensible.	Defining a generic Instrument type that can be specialized for different asset classes.

8.3.1. Pragmatic Considerations for Model Design

8.3.1.1. Behavior-Oriented

This strategy groups together components within a model that behave similarly. So, in our example of bonds and interest-rate swaps fundamentally, they share many characteristics and are used in very similar ways within a model. Therefore, it might make sense to group them together when developing a model.

8.3.1.2. Domain Expertise

It may be that components of the model require sufficient expertise that different persons or groups are involved in the development. This may warrant separating a model's design so that different groups can focus on more specific aspects, regardless of similarities among components. For example, at a higher vertical level of abstraction, financial derivatives may fall under similar grouping, but sufficient differences exist for equity, credit, or foreign exchange derivatives that the model should separate those three asset classes for development purposes.

8.3.1.3. Composability versus All-in-One

For some model design goals, it may be warranted to bundle together more functionality instead of allowing users to compose functionality that comes from different packages. For example, perhaps a certain visualization of model results is particularly useful, not easy to create from scratch, and it is widely desired to see the model output visualized that way. Instead of relying on the user to install a separate visualization package and develop the visualization themselves, it could make sense to bundle visualization functionality with a model that is otherwise unconcerned with graphical capabilities.

In general, loosely coupled systems are preferred—you want to pick and choose which components you use while ensuring they work well together.

8.4. Interfaces

Interfaces are the boundaries between different encapsulated abstractions. The user-facing interface is what users of a package or model must actually think about—separate from the intermediate variables, logic, and complexity hidden within.

i Example of an interface

When looking up a ticker for a market quote, one need not be mindful of the underlying realtime databases, networking, rendering text to the screen, memory management, etc. The interface is “put in symbol, get out number”. By design, there are multiple layers of interfaces and abstractions used under the hood, but the financial modeler need only be actively concerned about the points that he or she comes in contact with, not the entire chain of complexity.

For a financial model, there might be interfaces for bonds or interest rate swaps, and separate interfaces for calculating risk metrics or visualizing results. Good design separates visualization concerns from contract mechanics and other domain logic. Open-source ecosystems (for visualization, data frames, file I/O, statistics, etc.) make these boundaries easier to see; the hard part is drawing similar boundaries inside your financial model.

8. Higher Levels of Abstraction

However, it's often difficult to find where to draw lines within financial models. For example, should bonds and interest-rate swaps be in separate packages? Or both part of a broader fixed income package? This is where much of the art and domain expertise of the financial professional comes to bear in modeling. There would be no way for a pure software engineer to think about the right design for the system without understanding how underlying components share similarities or differences and how those components interact.

8.4.1. Defining Good Interfaces

A well-designed interface should follow these principles:

1. **Be minimal and focused.** The interface should provide only the essential functionality needed, without unnecessary clutter. This makes the interface easier to understand and facilitates building the necessary complexity through digestible, composable components.
2. **Be consistent and intuitive.** The interface should use consistent naming conventions, parameter orders, and behaviors. It should match the user's mental model and expectations.
3. **Hide implementation details.** The interface should abstract away the internal complexity and expose only what the user needs to know. This separation of concerns allows the implementation to change without affecting users of the interface.
4. **Be documented and contractual.** The interface should clearly specify what inputs it expects and what outputs or behaviors it provides. It forms a contract between the implementation and the users.
5. **Be testable.** A good interface allows the functionality to be easily tested through the public interface, without needing to access internal details.

8.4.2. Interfaces: A Financial Modeling Case Study

As a case study, we will look at the `FinanceModels.jl` and related packages to discuss some of the background and design choices that went into the functionality. This suite was written by one of the authors and is publicly available as a set of installable Julia packages.

8.4.2.1. Background

In actuarial work, you constantly work with interest rate and bond yield curves—determining forward rates, estimating the shape of future curves, or discounting cashflows to estimate a present value. Determining things like “given a par yield curve, what’s the implied discount factor for a cashflow at time 10” or “what is the 10 year BBB public corporate rate implied by the current curve in five years’ time” is cumbersome at best in a spreadsheet.

For example, to determine the answer to the first one (“a discount factor for time 10”) actually requires quite a bit of detail and assumption to derive:

- Reference market data and a specification for how that market data should be interpreted. For example, if given the rate `0.05` for time 10, is it quoted as a continuously compounded rate or as an annual effective rate? Is that a par rate, a zero-coupon bond (spot) rate, or a one-year-forward rate from time 10?
- Smoothing, interpolation, or extrapolation for noisy or sparse data. Should the rates be bootstrapped or fit to a parametrically specified curve?

This is exactly the kind of complexity we want to hide when the goal is simply to value a stream of riskless life insurance payments, which might look like this:

```

risk_free_rates = [0.05, 0.055, 0.06]      # example par yields
tenors = [1.0, 5.0, 10.0]                   # years
yield_curve = Yields.Par(risk_free_rates, tenors)

cashflow_vector = [1_000_000.0, 3_000_000.0, 1_000.0]
present_value(yield_curve, cashflow_vector)

```

The variable and function names make the purpose and steps clear. Now imagine doing this in a spreadsheet: bootstrapping, interpolation, and discounting before you even get to the present value you actually wanted. What can be five lines of code (with the right abstractions) would take hundreds of cells in a spreadsheet. Abstractions like these are productivity multipliers for financial modelers.

8.4.2.2. Initial Versions

Two main abstractions emerged in the early versions.

8.4.2.2.1. Rates

Utilizing the benefit of the type system, it was decided that it would be most useful to represent rates not as simple floating point numbers (e.g. 0.05) but instead with dedicated types to distinguish between rate conventions. The abstract type `CompoundingFrequency` had two subtypes: `Continuous` and `Periodic` so that a 5% compounded continuously versus an effective per period rate would be distinguished via `Continuous(0.05)` versus `Periodic(0.05,1)`. The two could be converted between by extending the built-in `Base.convert` function.

This was useful because once rates were converted into `Rates` within the ecosystem, that data contained within itself characteristics that could distinguish how downstream functionality should treat the rates.

8.4.2.2.2. Yield Curves

At first, only bootstrapping was supported as a method to construct curve objects. This required that there was only one rate given per time period (no noisy data) and only supported linear, quadratic, and cubic splines.

Further, there was a specific constructor for different common types of instruments. From the old documentation:

- `Yields.Zero(rates,maturities)` using a vector of zero, or spot, rates
- `Yields.Forward(rates,maturities)` using a vector of one-period
- `Yields.Constant(rate)` takes a single constant rate for all times
- `Yields.Par(rates,maturities)` takes a series of yields for securities priced at par. Assumes that maturities <= 1 year do not pay coupons and that after one year, pays coupons with frequency equal to the `CompoundingFrequency` of the corresponding rate.
- `Yields.CMT(rates,maturities)` takes the most commonly presented rate data (e.g. `Treasury.gov`) and bootstraps the curve given the combination of bills and bonds.
- `Yields.OIS(rates,maturities)` takes the most commonly presented rate data for overnight swaps and bootstraps the curve.

This covered a lot of lightweight use-cases, but made a lot of implicit assumptions about how the given rates should be interpreted.

8. Higher Levels of Abstraction

8.4.2.3. The Birth of FinanceModels

Several insights led to a more flexible interface in later versions.

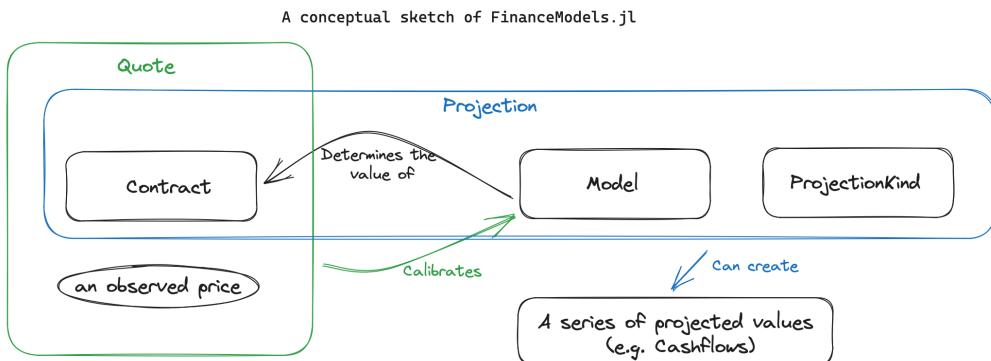


Figure 8.1.: A conceptual sketch of FinanceModels.jl components.

First, realizing that yield curves were just a particular kind of model - one that used interest rates to discount cashflows. But you can have different kinds of models - such as Black-Scholes option valuation or a Monte Carlo valuation approach. Likewise, the cashflows need not simply be a vector of floating point values, and instead it could be the representation of a generic financial contract. As long as the model knew how to value it, an appropriate present value could be derived.

Where previously it was:

```
present_value(yield_curve,cashflow_vector)
```

Now, it was

```
present_value(model,contract)
```

Second, that a model was simply some generic box that had been “fit” to previously observed prices for similar types of contracts we would be trying to value in the model. The combination of a contract and a price constituted a “quote” and with multiple quotes a model could be fit using various algorithms.

With these changes, the package that was originally called Yields.jl was renamed to FinanceModels.jl. The updated code from the earlier example now would be implemented like this:

```
using FinanceModels
risk_free_rates = [0.05, 0.055, 0.06]
tenors = [1.0, 5.0, 10.0]
quotes = ParYield.(risk_free_rates, tenors)

model = fit(Spline.Cubic(), quotes, Fit.Bootstrap())

cashflow_vector = [1_000_000.0, 3_000_000.0, 1_000.0]
present_value(model, cashflow_vector)
```

It's slightly more verbose, but notice how much more powerful and extensible `fit(Spline.Cubic(), quotes, Fit.Bootstrap())` is than `Yields.Par(risk_free_rates, tenors)`. The end result is the same, but now the same package and interface can clearly interchange other options, such as a NelsonSiegelSvensson curve instead of a spline. And the quotes could be a combination of observed bonds of different technical parameters (though still sharing characteristics which make it relevant for the model being constructed).

The same pattern also applies to option valuation, such as this example of vanilla euro options with an assumed constant volatility assumption:

```
a = Option.EuroCall(CommonEquity(), 1.0, 1.0)                                ①
b = Option.EuroCall(CommonEquity(), 1.0, 2.0)

qs = [                                                               ②
    Quote(0.0541, a),
    Quote(0.072636, b),
]

model = Equity.BlackScholesMerton(0.01, 0.02, Volatility.Constant())          ③

m = fit(model, qs)                                                       ④

present_value(m,qs[1].instrument)                                              ⑤
```

- ① The arguments to EuroCall are the underlying asset type, strike, and maturity time.
- ② A vector of observed option prices.
- ③ A BSM model with a given risk free rate, dividend yield, and a to-be-fit constant volatility component.
- ④ Fits the model and derives an estimated volatility close to 0.15 for the 1-year call (given the quoted price).
- ⑤ Values the contract; in this simple, noiseless setup we recover the original price of approximately 0.0541.

With a consistent interface that handles a wide variety of situations, you're free to expand the model in new directions. The built-in functionality lets you compose pieces together in ways that weren't possible with the less abstracted design. For example, the equity option example had no parallel when all of the available constructors were `Yields.Zero` or `Yields.Par` and would have required a completely from-scratch implementation with newly defined functions.

Further, and critically, the new design allows modelers to create their own models or contracts¹ and extend the existing methods rather than needing to create their own: the function signature `fit(model, quotes)` handles a very wide variety of cases, as does `present_value(model, contract)`.

8.4.3. Abstraction Design Checklist

When you are introducing a new abstraction in a production model, walk through the following quick diagnostic:

1. **Audience** - Who will consume this interface (internal quants, downstream systems, regulators)? Tailor naming and documentation to their vocabulary.
2. **Surface area** - Can the public API be expressed in a dozen functions or fewer? If not, consider breaking the concept into smaller submodules.

¹And projections, which is handled by defining a `ProjectionKind`, such as a cashflow or accounting basis. This topic is covered in more detail in the `FinanceModels.jl` documentation.

8. Higher Levels of Abstraction

3. **Extensibility** - Do you know how a new asset class, reporting basis, or scenario type would plug in without editing core code? If the answer is unclear, capture that as an interface requirement now.
4. **Testing strategy** - Identify the seams where unit tests or property-based tests will live. Abstractions without convenient test hooks tend to erode.
5. **Failure modes** - Decide what should happen when data is missing, calibration fails, or convergence is slow. Document these behaviors in the API docstrings so consumers are not surprised.

Keeping these questions in mind ensures the abstraction remains a tool, not a new source of opacity.

8.5. Macros & Homoiconicity

We have talked about transforming data and restructuring logic in order to make the model more effective. We can go even further: we can abstract the process of writing code itself. This subject is a bit advanced, so we are simply going to introduce it because you will likely find many convenient instances of it as a *user* even if you never need to implement it yourself.

Homoiconicity refers to the property of a programming language where the language's code can be represented and manipulated as a data structure in the language itself. Think of a recipe. You can follow the recipe's instructions to bake a cake — that's like executing code. But you could also treat the recipe itself as data to be analyzed or transformed: you could write a program to scan thousands of recipes, find every instance of "sugar," and reduce the quantity by 25%. This is the essence of homoiconicity: the same representation serves both as executable instructions and as data that can be inspected or transformed.

In a homoiconic language like Julia, the code is data and can be treated as such. This enables powerful metaprogramming (i.e. code that writes other code) capabilities, where code can generate or transform code during macro expansion (before runtime), producing expressions that are then compiled.

Macros are a metaprogramming feature that leverage homoiconicity in Julia. They allow the programmer to write code that generates or manipulates other code at compile-time. Macros take code as input, transform it based on certain rules or patterns, and return the modified code which then gets compiled.

For example, a built-in macro is `@time` which will measure the elapsed runtime for a piece of code².

```
@time exp(rand())
```

Will effectively expand to:

```
t0 = time_ns()
value = exp(rand())
t1 = time_ns()
println("elapsed time: ", (t1-t0)/1e9, " seconds")
value
```

Here it is when we run it:

²`@time` is a simple, built-in macro. For true benchmarking purposes, see Section 24.4.

```

@time exp(rand())
0.000000 seconds
1.168199290105802

```

8.5.1. Metaprogramming in Financial Modeling

In the context of financial modeling, macros can be used to simplify repetitive or complex code patterns, enforce certain conventions or constraints, or generate code based on data or configuration.

Here are a few potential use cases of macros in financial modeling. Again, these are more advanced use-cases but knowing that these paths exist may benefit your work in the future.

1. Defining custom DSLs (Domain-Specific Languages): Macros can be used to create expressive and concise DSLs tailored to financial modeling. For example, a macro could allow defining financial contracts using syntax closer to the domain language, which then gets expanded into the underlying implementation code.
2. Automating boilerplate code: Macros can help reduce code duplication by generating common patterns or boilerplate code. This can include generating accessor functions³, constructors, or serialization logic based on type definitions.
3. Enforcing conventions and constraints: Macros can be used to enforce coding conventions, such as naming rules or type checks, by automatically transforming code that doesn't adhere to the conventions. They can also be used to add runtime assertions or checks based on certain conditions.
4. Optimizing performance: Macros can be used to perform code optimizations at compile-time. For example, a macro could unroll loops, inline functions, or specialize generic code based on specific types or parameters, resulting in more efficient runtime code.
5. Generating code from data: Macros can be used to generate code based on external data or configuration files. For example, a macro could read a specification file and generate the corresponding financial contract types and functions.

8.6. Key Takeaways

- Start by clarifying which concern you are abstracting: behavior, domain ownership, or user experience. Each leads to different module boundaries and collaboration patterns.
- Build narrow public interfaces that accept abstract types (for example, `present_value(model, contract)`) so that new instruments or calibration routines can be composed without editing core logic.
- Package sensible defaults with escape hatches. Providing a ready-made spline curve or report template lowers adoption friction while still allowing dependency injection.
- Treat metaprogramming as a force multiplier. Even if you only consume macros, understanding how they enforce conventions and eliminate boilerplate will help you evaluate third-party packages and maintain your own.

³Accessor functions are useful when working with nested data structures. For example, if you have a `struct` within a `struct` and want to conveniently access an inner struct's field.

8. Higher Levels of Abstraction

Part IV.

Foundations: Building Performant Models

"Premature optimization is the root of all evil (or at least most of it) in programming." - Donald Knuth

After establishing foundational programming concepts, we turn our attention to performance - a critical consideration for real-world financial models. While modern computers are remarkably powerful, thoughtlessly constructed models can still grind to a halt when faced with large portfolios or complex analyses. This section explores how to harness computational resources effectively, starting from the hardware fundamentals that both constrain and enable our work.

Understanding performance requires looking beneath the abstractions we've built. Just as a financial modeler benefits from understanding market mechanics rather than treating them as black boxes, knowledge of computational infrastructure allows informed decisions about model architecture. We'll examine how hardware characteristics influence algorithm design, memory usage, and execution speed.

We'll introduce when optimization *does* matter, and equally important when it *doesn't*. The goal isn't to optimize prematurely or pursue performance at all costs. Rather, we aim to build models that scale gracefully as demands grow, whether through larger datasets, more sophisticated analyses, or tighter time constraints. We'll progress from single-threaded optimization techniques to parallel processing approaches, always with an eye toward practical application in financial contexts.

By the end of this section, you'll have the knowledge needed to diagnose performance bottlenecks and implement appropriate solutions, ensuring your models remain responsive and reliable as they evolve. Let's begin by examining the hardware foundation upon which all our computational work rests.

9. Hardware and Its Implications

CHAPTER AUTHORED BY: ALEC LOUDENBACK

a CPU is literally a rock that we tricked into thinking.

not to oversimplify: first you have to flatten the rock and put lightning inside it.

- Twitter user daisyyowl, 2017

9.1. Chapter Overview

A basic understanding of computing hardware is essential for optimizing financial models. Understanding how data is stored and processed helps you make better design decisions and avoid common pitfalls. We'll cover memory architecture, data storage, and how hardware affects computational speed.

9.2. Introduction

The quote that opens the chapter is a silly way of describing that most modern computers are made with silicon, a common mineral found in rocks. However, we will not concern ourselves with the raw materials of computers and instead will focus on the key architectural aspects.

A computer handles data at rest (in memory) or data being acted upon (processed). This chapter will try to explain both of those processes in a way that reveals key reasons why different approaches to programming can yield different results in terms of processing speed, memory usage, and compiled outputs. The goal is not to turn you into a hardware designer, but to help you reason about why one valuation run takes seconds while another takes hours.

9.3. Memory and Moving Data Around

The core of modeling on computers is to perform computations on data, but unfortunately the speed at which data can be *accessed* has grown much slower than the rate the actual computations can be performed. This is often called the **memory bottleneck**. Further, the size of the available persistent data storage (HDDs/SSDs) has ballooned, exacerbating the problem: the overall throughput of memory is the typical workflow constraint. To try to address the bottleneck (memory throughput), solutions have been developed to create a pipeline to efficiently shuttle data to and from the processor and the persistent storage. This memory and processing architecture applies at both the single computer level as well as extending to workflows between different data stores and computers.

We will focus primarily on the architecture of a single computer, as even laptop computers today contain enough power for most modeling tasks, *if the computer is used effectively*. Further, learning how to optimize a program for a single computer/processor is almost always a precursor step to effective parallelization as well.

9. Hardware and Its Implications

9.3.1. Memory Types and Location

Memory has an inverse relationship between size and proximity to the central processing unit (CPU). The closer the data is to the processor units, the smaller the storage and the less likely the data will persist at that location for very long.

Kind	Typical capacity	Typical latency	Persistence
SSD or HDD	1–10+ TB	100,000–10,000,000 ns (read/seek dependent)	Persistent
RAM (DRAM)	8–256+ GB	60–120 ns	Volatile (power-dependent)
CPU Cache – L3	4–128 MB	10–40 ns	Volatile
CPU Cache – L2	256 KB–8 MB	3–10 ns	Volatile
CPU Cache – L1	32–128 KB	0.5–1.5 ns	Volatile
CPU Registers	a few hundred bytes	single-cycle access	Volatile

After requesting data from a persistent location like a Solid State Drive (SSD), the memory is read into Random Access Memory (RAM). The advantage of RAM over a persistent location is speed—typically that memory can be accessed and modified many times faster than the persistent data location. The tradeoff is that RAM is not persistent: when the computer is powered down, the RAM loses the information stored within.

When data is needed by the CPU, it is read from RAM into a small hierarchy of caches before being accessed by the CPU. The **CPU caches** are small and very fast, and they move data in fixed-size ‘cache lines’ (typically 64 bytes). The caches are also physically co-located with the CPU for efficiency. Data is organized and funneled through the caches as an intermediary between the CPU and RAM and is fed from Level 3 (L3) cache in steps down to L1 cache as the data gets closer to the processor.

At the very top of the memory hierarchy are **CPU registers**—tiny storage locations on the chip that the processor can read and write in a single cycle. This is where the CPU actually performs calculations. The CPU’s execution units operate on values held in registers. Most arithmetic, logical, and comparison instructions take their operands from registers and write results back to registers; separate load/store instructions move data between memory and registers.

Note

Memory units at a glance:

- 1 bit is a single 0 or 1; 8 bits = 1 byte.
- **Binary multiples** (common for RAM/caches and many OS readouts):
 - 1 KiB = 1,024 bytes; 1 MiB = 1,024 KiB; 1 GiB = 1,024 MiB.
- **Decimal multiples** (common for storage devices and network speeds):
 - 1 kB = 1,000 bytes; 1 MB = 1,000,000 bytes; 1 GB = 1,000,000,000 bytes.
- **Bytes vs bits:** uppercase B = bytes; lowercase b = bits. Network speeds are often quoted in bits (e.g., 1 Gbps = 1 gigabit per second = 125 MB/s).

Rule of thumb: treat memory sizes (RAM, caches) as base-2, and disk/network specs as base-10. The “i” in KiB/MiB/GiB explicitly means base-2 ($2^{10} = 1,024$).

9.3.2. Stack vs Heap

Sitting within the RAM region of memory are two sections called Stack and Heap. These are places where variables created from our program’s code can be stored. In both cases, the program will request memory space but they have some differences to be aware of.

- The **stack** stores small, fixed-size (known bit length) data and program components. The stack is a last-in-first-out queue of data that is able to be written to and read from very quickly. The stack is CPU cache friendly.
- The **heap** is a region which can be dynamically sized and has random read/write (you need not access the data in a particular order). The heap is much slower but more flexible. The heap is not very CPU cache friendly.

💡 Why this matters for models

When a Monte Carlo engine allocates a new vector for every inner-loop step, it is hitting the heap repeatedly and inviting garbage-collector work. Reusing buffers or preallocating arrays keeps data on the stack and in caches, which can reduce runtimes by an order of magnitude. See Chapter 24 for tips on how to optimize Julia code.

9.3.2.1. Garbage Collector

The **garbage collector** is a program that gets run to free up previously requested/allocated memory. It accomplishes this by keeping track of references to data in memory by section of your code. If a section of code is no longer reachable (e.g. inside a function that will never get called again, or a loop that ran earlier in the program but is now complete), then, periodically, the garbage collector will pause execution of the primary program in order to “sweep” the memory. This step marks the space as able to be reused by your program or the operating system.

💡 Tip

Julia uses a generational, stop-the-world garbage collector to reclaim heap memory that is no longer reachable by your program. Periodically, execution pauses briefly while the GC identifies unreachable objects and frees them. In performance-sensitive code, reducing heap allocations (e.g., by reusing buffers, working in-place, and writing type-stable code) minimizes GC pressure and improves throughput.

9.4. Processor

The processor reads cache lines (typically 64 bytes) from caches and moves the needed values into registers and then executes instructions. An example would be to take the bytes from register 10 and add the bytes from register 11 to them. This is really all a processor does at the lowest level: combining bits of data using logical circuits.

9. Hardware and Its Implications

Logical circuits (**transistors**) are an arrangement of wires that output a new electrical signal that varies depending on the input. From a collection of smaller building block gates (e.g. AND, OR, NOR, XOR) more complex operations can be built up¹, into operations like addition, multiplication, division, etc. Electric impulses move the state of the program forward once per CPU cycle (controlled by a master “clock” ticking billions of times per second). CPU cycle speed is what’s quoted for chip performance, e.g. when a CPU is advertised as 3.0 GHz (or 3 billion cycles per second).

The programmer (or compiler, if we are working in a higher level language like Julia) tells the CPU which instruction to run. The set of instructions that are valid for a given processor is called the Instruction Set Architecture, or ISA. In computer Assembly language (roughly one-level above directly manipulating the bits), the instructions are given names like ADD, SUB, MUL, DIV, and MOV. These instructions mirror the raw instruction that is part of the ISA.

Not all instructions are created equal, however. Some instructions take many CPU cycles to complete. For example, a floating point division instruction (FDIV) has a latency of 10-20 CPU cycles, while floating point addition (FADD) completes in 3-4 cycles. At 3 GHz, that’s the difference between ~1 ns and ~6 ns for the raw operation.

Some architecture examples that may be familiar:

- **x86-64** (also called AMD64 or Intel 64) processors use 64-bit registers and the x86 instruction set. This architecture dominates desktops, laptops, and servers. The “64” refers to the register width; the prior generation was 32-bit x86.
- **ARM** processors, including Apple’s M-series chips, use the ARM instruction set. The 64-bit variant is called **AArch64** (or ARM64). You’ll see “aarch64” in file paths, download options, and Julia’s `Sys.ARCH` output when running on these processors.

The ARM architecture is a **Reduced Instruction Set Computer** (RISC) design. Despite the name, modern ARM chips have extensive instruction sets including SIMD extensions (NEON, SVE). The “reduced” refers to instruction *complexity*: each instruction does less work but executes in predictable time, enabling simpler pipelines and better power efficiency. x86’s **Complex Instruction Set Computer** (CISC) heritage means some instructions do more work (e.g., memory access and computation in one instruction) but with variable timing.

In practice, both architectures have converged: x86 chips decompose complex instructions into simpler micro-operations internally, while ARM has added specialized instructions over time. For specialized workloads, specific instructions can make programs 10-100x faster. For example, Intel’s AVX-512 extensions (see Chapter 11) can dramatically accelerate vectorizable numerical code, while ARM’s SVE (Scalable Vector Extension) provides similar benefits on that architecture.

Tip

Trying to optimize your program via selecting specialized chips should be one of the *last* ways that you seek to optimize runtime, as generally a similar order of magnitude speedup can be achieved through more efficient algorithm design or general parallelization techniques. Developing programs in this way makes the performance *portable*, able to be used on other systems and not just special architectures.

When writing in Julia, you need not be concerned with the low-level instructions as the compiler will optimize the execution for you. However, should it be useful, it is easy to inspect the compiled code. For example, if we create a function to add three numbers, we can see that the ADD instruction is called twice: first adding the first and second arguments, and then adding the third argument to that intermediate sum.

¹In fact, only two logical gates are needed to reproduce all boolean logical gates: NAND (Not AND) and NOR gates can be composed to create AND, OR, NOR, etc. gates.

```

myadd(x, y, z) = x + y + z
@code_native myadd(1, 2, 3)

.section    __TEXT,__text,regular,pure_instructions
.build_version macos, 16, 0
.globl _julia_myadd_3082           ; -- Begin function julia_myadd_3082
.p2align   2
_julia_myadd_3082:                ; @julia_myadd_3082
; Function Signature: myadd(Int64, Int64, Int64)
; | @ /Users/alecloudenback/prog/julia-fin-book/hardware.qmd:121 within `myadd`
; %bb.0:                                ; %top
; DEBUG_VALUE: myadd:x <- $x0
; DEBUG_VALUE: myadd:x <- $x0
; DEBUG_VALUE: myadd:y <- $x1
; DEBUG_VALUE: myadd:y <- $x1
; DEBUG_VALUE: myadd:z <- $x2
; DEBUG_VALUE: myadd:z <- $x2
; | @ operators.jl:642 within `+` @ int.jl:87
    add x8, x1, x0
    add x0, x8, x2
    ret
; LL
; -- End function
.subsections_via_symbols

```

Compilers are complex, hyper-optimized programs which turn your source code into the raw bits executed by the computer. Key steps in the process of converting Julia code you write all the way to binary machine instructions include the items in Table 9.2. Note the Julia `@code_...` macros allow the programmer to inspect the intermediate representations. The table covers the lowered AST, LLVM intermediate representation, and native machine code.

Table 9.2.: Part of the key to Julia's speed is to be able to compile down to a different, specialized version of the machine code depending on the types given to a function. As described in the table above, the instructions for adding floating-point numbers or integer numbers are different. Julia code can reflect that distinction by compiling a different method for each combination of input types.

Step	Description	Example
Julia Source Code Lowered Abstract Syntax Tree (AST)	The level written by the programmer in a high level language. An intermediate representation of the code after the first stage of compilation, where the high-level syntax is simplified into a more structured form that's easier for the compiler to work with.	<pre>myadd(x,y,z) = x + y + z julia> @code_lowered myadd(1,2,3) CodeInfo(1 - %1 = x + y + z └─ return %1)</pre>
LLVM	Low-Level Virtual Machine language, which is a massively popular compiler used by languages like Julia and Rust. The core logic are the three lines with add, add, and ret. Note that the add instruction is add i64, which means an addition operation of 64-bit integers.	<pre>julia> @code_llvm myadd(1,2,3) ; @ REPL[7]:1 within `myadd` define i64 @julia_myadd_2022(i64 signext %0, i64 signext %1, i64 signext %2) #0 { top: ; ┌ @ operators.jl:587 within `+` @ int.jl:87 %3 = add i64 %1, %0 %4 = add i64 %3, %2 ret i64 %4 ; }</pre>

Step	Description	Example
Native	<p>The final machine code output, specific to the target CPU architecture. This is at the same level as Assembly language. The core logic are the three lines beginning with add, add, and ret.</p> <p>If we used floating point addition instead, the CPU instruction would be fadd instead of add.</p>	<pre> julia> @code_native myadd(1,2,3) .section --TEXT,--text,regular,pure_instructions .build_version macos, 14, 0 .globl _julia_myadd_1851 ; -- Begin function julia_myadd_1851 .p2align 2 -julia-myadd_1851: ; @julia-myadd_1851 ; %bb.0: ; @ REPL[7]:1 within `myadd` ; %top ; @ operators.jl:587 within `+` @ int.jl:87 add x8, x1, x0 add x0, x8, x2 ret ; LL </pre>

9. Hardware and Its Implications

9.4.0.1. Increasing Complexity in Search of Performance

Transistors are the building-block that creates the CPU and enables the physical process which governs the computations. For a very long time, the major source of improved computer performance was simply to make smaller transistors, allowing more of them to be packed together to create computer chips. This worked for many years and the propensity was for the transistor count to double about every two years. In this way, software performance improvements came as a side effect of the phenomenal scaling in hardware capability. However, raw single core performance and clock frequency (CPU cycle speed) dramatically flattened out starting a bit before the year 2010. This was due to the fact that transistor density has been starting to be limited by:

1. Pure physical constraints (transistors can be measured in width of atoms) where we have limited ability to manufacture something so small.
2. Thermodynamics, where heat can't be removed from the CPU core fast enough to avoid damaging the core and therefore operations per second are capped.

To obtain increasing performance, two main strategies have been employed in lieu of throwing more transistors into a single core:

1. **Parallelism:** utilize multiple, separate cores and operate in an increasingly parallel way.
2. **Microarchitectural techniques:** clever tricks to predict, schedule, and optimize the computations to make better use of the memory pipeline and otherwise idle CPU cycles.

We will cover techniques to utilize concurrent/parallel processing in Chapter 11. As for the second technique, it is capable of very impressive accelerations (on the order of 2x to 100x faster than a naive implementation). However, it has sometimes caused issues. There have been some famous security vulnerabilities such as Spectre and Meltdown, which exploited speculative execution – a technique used to optimize CPU performance which will execute code *before being explicitly asked to* because the scheduler *anticipates* the next steps (with very good, but imperfect accuracy).

For practitioners this means: expect hardware vendors to keep adding cores, vector units, and specialized instructions (AVX, SVE, BF16) while single-thread speed inches forward. Models that parallelize cleanly—scenario loops, policy seriatim projections, P&L explain—will benefit automatically from newer CPUs. Models that rely on single-core performance or constantly move data back and forth to main memory will hit a wall regardless of silicon improvements.

9.5. Logistics Warehouse Analogy

The problem is analogous to a logistics warehouse (persistent data) which needs to package up orders (processor instructions). There's a conveyor belt of items being constantly routed to the packaging station. In order to keep the packing station working at full capacity, the intermediate systems (RAM & CPU caches) are funneling items they *think* will be needed to the packager (data that's *expected* to be used in the processor). Most of the time, the necessary item (data) is optimally brought to the packaging station (process), or a nearby holding spot (CPU cache).

This system has grown very efficient, but sometimes the predictions miss or a never-before-ordered item needs to be picked from the far side of the warehouse and this causes significant delays to the system. Sometimes a package will start to be assembled before the packager has even gotten to that order (branch prediction) which can make the system faster most of the time, but if the predicted package isn't actually what the customer ordered, then the work is lost and has to be redone (branch mis-predict).

There are a lot more optimizations along the way:

- Since the items are already mostly arranged so that related items are next to each other, the conveyor belt will bring nearby items at the same time it brings the requested item (memory blocks).
- If an item is usually ordered after another one, the conveyor system will start to bring that second item as soon as the first one is ordered (prefetching).
- Different types of packaging stations might be used for specialized items (e.g. vector processing or cryptography instructions in the CPU).

For finance workloads you control the pick list: structuring arrays so related fields are adjacent, batching valuation requests by product type, and avoiding random “walks” through large scenario tables all help the warehouse keep its conveyor belts full.

Financial Modeling Pro Tip

Julia arrays are column-major. Access memory contiguously to exploit caches:

```
X = randn(10_000, 100) # rows x cols
# Fast: process columns (contiguous memory)
sums = map(sum, eachcol(X))

# Slower: process rows (strided access)
sums_rows = map(sum, eachrow(X))
```

In portfolio/risk code, organize factor loadings or scenario matrices so your hot loops iterate along columns where possible. This often yields 2–10× speedups purely from better cache locality.

9.6. Speed of Computer Actions

In a financial model, even small delays (such as main memory references vs. L1 cache access) can accumulate quickly in high-frequency trading or risk calculation routines. Understanding these timings can guide decisions on structuring data access patterns and deciding what data to cache or load in memory for optimal performance.

Representative time is given in Table 9.3 for a variety of common actions performed on a computer. It’s clear that having memory access from a local source is better for computer performance!

9. Hardware and Its Implications

Table 9.3.: How long different actions take on a computer. As an interesting yardstick, the distance a beam of light travels is also given to provide a sense of scale (this comparison originally comes from Admiral Grace Hopper). Source for the timings comes from: Brown University CS 0300 Course Staff (2022). Also note that timings vary substantially by hardware and workload; treat these as order-of-magnitude references.

Operation	Time (ns)	Distance light traveled
Single CPU cycle (e.g., one add on a register)	0.3	9 centimeters
L1 cache reference	1	30 centimeters
Branch mispredict	5	150 centimeters
L2 cache reference	5	150 centimeters
Main memory reference	100	30 meters
Read 1 MB sequentially from RAM	250,000	75 km (\approx 2 marathons)
Round trip within a datacenter	500,000	150 km (thickness of Earth's atmosphere)
Read 1 MB sequentially from SSD	1,000,000	300 km (Washington, D.C. to New York City)
Hard disk seek	10,000,000	3,000 km (width of the continental U.S.)
Send packet CA→Netherlands→CA	150,000,000	45,000 km (Earth's circumference)

As introduced in Section 9.4, calculations on the CPU vary widely. For example, Table 9.4 shows runtime varying quite a bit for common mathematical operations. Despite mathematically all essentially being elementary operations, the compute intensity varies widely:

Table 9.4.: Benchmarked runtime of Julia functions on Float64 arguments. These timings include function call overhead; raw CPU instruction latencies differ (e.g., hardware division is typically 10-20× slower than addition). The similar times for basic arithmetic reflect that these operations compile to single instructions where the benchmark overhead dominates. The key takeaway is the relative cost of transcendental functions like $^$, \exp , and \log .

Mathematical Operation	Julia Function	Runtime in nanoseconds (Float64 arguments)
$a + b$	$+$	1.7
$a - b$	$-$	1.7
a/b	$/$	1.7
$a \times b$	$*$	1.7
a^b	$^$	10.8
$\exp(a)$	\exp	3.9
$\log_{10}(a)$	$\log10$	5.4
$\ln(a)$	\log	3.7
$\text{abs}(a)$	abs	1.7

 Tip

One way to speed up financial models is to utilize techniques to avoid using the \wedge operation, as it's one of the most expensive basic operations, but unfortunately is also one of the most common when dealing with rates and compounding.

Some strategies:

- Instead of using interest rates which require exponentiation, utilize continuously compounded rates and use the `exp` and `log` functions instead.
- For small r , use `expm1` and `log1p` to improve accuracy:
 - `expm1(x)` computes $\exp(x) - 1$ accurately for small x .
 - `log1p(x)` computes $\log(1 + x)$ accurately for small x .
- Say you are running a monthly model with inputs that are annual rates. Before running through the 'hot loop', pre-compute the annual rates into a vector of monthly rates to avoid re-computing this transformation at the cell/seriatim level.

9. Hardware and Its Implications

10. Writing Performant Single-Threaded Code

CHAPTER AUTHORED BY: ALEC LOUDENBACK

Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away. - Antoine de Saint-Exupéry, *Airman's Odyssey*

10.1. Chapter Overview

Understand single-threaded performance by focusing on memory access, type stability, and branch prediction; profile and benchmark before reaching for parallelism; learn how to upgrade hot loops and data structures in Julia so that parallel versions start from a solid baseline.

10.2. Introduction

On today's hardware, the highest-throughput workloads often run on GPUs via massive parallelism. However, writing correct, performant parallel code relies on understanding efficient sequential patterns first. Additionally, many problems are not "massively parallelizable" and a sequential model architecture is required. For these reasons, it's critical to understand sequential patterns before moving onto parallel code.

For those coming from fundamentally slower languages (such as R or Python), the common advice to speed things up is often to try to parallelize code or use array-based techniques. With many high level languages the only way to achieve reasonable runtime is to utilize parallelism. In contrast, fast languages like Julia can produce surprisingly quick single-threaded programs when you keep an eye on allocations, types, and cache behavior.

Further, it may be that a simpler, easier to maintain sequential model is preferable to a more complex parallel version if maximum performance is not required. Like the quote that opened the chapter, you may prefer a simpler sequential version of a model to a more complex parallel one.

💡 Tip

Developer time (your time) is often more expensive than wall-clock runtime, so be prepared to accept a good-enough but sub-optimal implementation instead of spending days chasing microseconds that the business will never notice.

ℹ️ Note

This section of the book continues to focus on concepts that are more general than just Julia. We will elaborate on Julia-specific techniques in the section starting with Chapter 21.

10.3. Patterns of Performant Sequential Code

Sequential code performance depends on five key patterns:

1. Efficient memory usage
2. Optimized memory access
3. Appropriate data types
4. Type stability
5. Branch prediction optimization

Understanding these patterns helps you write faster code by leveraging CPU architecture and Julia's compiler optimizations. We'll look at each of these in turn.

10.3.1. Minimize Memory Allocations

Allocating memory onto the Heap takes a lot more time than (1) not using intermediate memory storage at all, or (2) utilizing the Stack. Each allocation requires time for memory management and requires the garbage collector, which can significantly impact performance, especially in tight loops or frequently called functions.

 Note

Tight loops or **hot loops** are the performance critical section of the code that are performed many times during a computation. They are often the “inner-most” loop of a nested loop algorithm.

A general rule of thumb is that dynamically sizable or mutable objects (arrays, mutable structs) will be heap allocated, while small fixed size objects can be stack allocated. For mutable objects, a common technique is to pre-allocate an array and then re-use that array for subsequent calculations. In the following stylized example we accumulate bond-level cashflows into a shared vector rather than constructing (**pre-allocating**) intermediate arrays and summing them later:

```
end_time = 10
cashflow_output = zeros(end_time)

par_bonds = map(1:1000) do i
    (tenor=rand((3, 5, 10)), rate=rand() / 10)
end

# sum up all of the bond cashflows into cashflow_output
for asset in par_bonds
    for t in 1:end_time
        if t == asset.tenor
            cashflow_output[t] += 1 + asset.rate
        else
            cashflow_output[t] += asset.rate
        end
    end
end
cashflow_output
```

```
10-element Vector{Float64}:
50.82937988250599
50.82937988250599
```

```

384.8293798825067
50.82937988250599
402.8293798825068
50.82937988250599
50.82937988250599
50.82937988250599
50.82937988250599
364.8293798825068

```

Julia's `@allocated` macro will display the number of bytes allocated by an expression, helping you identify and eliminate unnecessary allocations.

```

random_sum_alloc() = sum([rand() for _ in 1:10])      # allocates
random_sum_noalloc() = sum(rand() for _ in 1:10)      # generator, no array

println("Allocating version: ", @allocated(random_sum_alloc()), " bytes")
println("Non-allocating version: ", @allocated(random_sum_noalloc()), "
    ↳ bytes")

```

```

Allocating version: 144 bytes
Non-allocating version: 0 bytes

```

10.3.2. Optimize Memory Access Patterns

Optimizing memory access patterns is essential for leveraging the CPU's cache hierarchy effectively. Modern CPUs have multiple levels of cache (L1, L2, L3), each with different sizes and access speeds. By structuring your code to access memory in a cache-friendly manner, you can significantly reduce memory latency and improve overall performance. For finance this shows up when you value a matrix of policy projections or simulate thousands of Monte Carlo paths: the order in which you touch memory dominates runtime once the arithmetic becomes trivial.

What is cache-friendly memory access? Essentially it boils down to spatial and temporal locality.

10.3.2.1. Spatial Locality

Spatial locality refers to accessing data that is physically near each other in memory (e.g. contiguous blocks of data in an array).

For example, it is better to access data in a linear order rather than random order. In actuarial projections we often sum along policy-by-policy arrays; keeping the loop aligned with the storage order avoids cache misses. The benchmark below highlights the gap:

```

using BenchmarkTools, Random

# Create a large array of structs to emphasize memory access patterns
struct DataPoint
    value::Float64
    # Add padding to make each element 64 bytes (a typical cache line size)
    padding ::NTuple{7,Float64}
end

function create_large_array(n)
    [DataPoint(rand(), tuple(rand(7)...)) for _ in 1:n]
end

```

10. Writing Performant Single-Threaded Code

```
# Create a large array
const N = 1_000_000
large_array = create_large_array(N)

# Function for sequential access
function sequential_sum(arr)
    total = 0.0
    for i in eachindex(arr)
        total += arr[i].value
    end
    total
end

# Function for random access
function random_sum(arr, indices)
    total = 0.0
    for i in indices
        total += arr[i].value
    end
    total
end

# Create shuffled indices
shuffled_indices = shuffle(1:N)

# Benchmark
println("Sequential access:")
@btime sequential_sum($large_array)

println("\nRandom access:")
@btime random_sum($large_array, $shuffled_indices)
```

Sequential access:
516.500 µs (0 allocations: 0 bytes)

Random access:
2.228 ms (0 allocations: 0 bytes)

499997.43729574303

When the data is accessed in a linear order, it means that the computer can load chunks of data into the cache and it can operate on that cached data for several cycles before new data needs to be loaded into the cache. In contrast, when accessing the data randomly, then the cache frequently needs to be populated with a different set of bits from a completely different part of our array.

10.3.2.1.1. Column vs Row Major Order

All multi-dimensional arrays in computer memory are actually stored linearly. When storing the multi-dimensional array, an architectural decision needs to be made at the language-level and Julia is column-major, similar to many performance-oriented languages and libraries (e.g. LAPACK, Fortran, Matlab). Values are stored going down the columns instead of across the rows.

For example, this 2D array would be stored as [1, 2, 3, ...] in memory, which is made clear via `vec` (which turns a multi-dimensional array into a 1D vector):

```

let
    array = [
        1 4 7
        2 5 8
        3 6 9
    ]

    vec(array)
end

9-element Vector{Int64}:
1
2
3
4
5
6
7
8
9

```

When working with arrays, prefer accessing elements in column-major order (the default in Julia) to maximize spatial locality. This allows the CPU to prefetch data more effectively.

You can see how summing up values across the first (column) dimension is much faster than summing across rows, because column-major storage keeps column elements contiguous in memory:

```

@btime sum(arr, dims=1) setup = arr = rand(1000, 1000)
@btime sum(arr, dims=2) setup = arr = rand(1000, 1000)

72.833 μs (3 allocations: 8.08 KiB)
110.083 μs (3 allocations: 8.08 KiB)

1000×1 Matrix{Float64}:
496.7606374537808
496.2992672636098
518.4739198488005
520.0941906601452
488.97696326354236
520.297021163007
511.04251073422205
511.8882145115975
494.83667763105615
502.1365529207229
⋮
503.0363612702995
491.829140978227
489.3551301833636
504.1127661414221
513.1732299433362
513.6860980759984
507.9893297896889
492.7209212998694
512.4811392457705

```

i Note

In contrast to the prior example, a row-major memory layout would have the associated data stored in memory as:

[1, 4, 7, 2, 5, 8, 3, 6, 9]

The choice between row and column major reflects the historical development of scientific computing and mathematical conventions. Column-major originated in Fortran and LAPACK, cornerstones of high-performance computing and linear algebra. This aligns with mathematical notation where vectors are typically represented as matrix columns.

Row major languages include:

- C/C++
- Python (NumPy arrays)
- C#
- Java
- JavaScript
- Rust

Column major languages (and libraries):

- Julia
- MATLAB/Octave
- R
- Fortran
- LAPACK/BLAS libraries

10.3.2.2. Temporal Locality

Hardware prefetchers (predictors) and the cache hierarchy exploit temporal locality by keeping recently used data “hot” and proactively fetching nearby data based on recent access patterns. This is an example of keeping “hot” data more readily accessible to the CPU than “cold” data.

When working sets exceed available RAM, the operating system may page memory to a “swap file” on disk. Recently accessed pages tend to remain in RAM, while “colder” pages are more likely to be swapped out.

10.3.3. Use Efficient Data Types

The right data type can lead to more compact memory representations, better cache utilization, and more efficient CPU instructions. This is another case of where having a smaller memory footprint allows for higher utilization of the CPU since computers tend to be memory-constrained in speed.

On some CPUs, you may find performance by using the smallest data type that can accurately represent your data. For example, prefer Int32 over Int64 if your values will never exceed 32-bit integer range. For floating-point numbers, use Float32 instead of Float64 if the reduced precision is acceptable for your calculations. These smaller types not only save memory but also allow for more efficient vectorized operations (see Chapter 11) on modern CPUs.

⚠ Warning

When choosing data types, consider that on modern 64-bit architectures, using smaller integer or floating-point types does not always improve performance. In fact, many operations are optimized for 64-bit (or even larger SIMD) registers, and using smaller data types may introduce unnecessary overhead due to conversions or misalignment.

As a rough guideline, if your data naturally fits within 64 bits (e.g. `Float64` or `Int64`), starting there is often the best balance of performance and simplicity. You can experiment with smaller types if you know your values never exceed certain ranges and memory footprint is critical. However, always benchmark to confirm any gains - simply using a smaller type does not guarantee improved throughput on modern CPUs.

For collections, choose appropriate container types based on your use case. Arrays are efficient for calculations that loop through all or most elements, while Dictionaries are better for sparse look-ups or outside of the “hot loop” portion of a computation.

Consider using small, statically sized collections when the data is suited for it. Small, fixed-size arrays (such as `StaticArrays.jl` in Julia) can be allocated on the stack and lead to better performance in certain scenarios than dynamically sizable arrays. The trade-off is that the static arrays require more up-front compile time and after a certain point (length in the 50-100 element range) it usually is not worth trying to use them.

10.3.4. Avoid Type Instabilities

Type instability occurs when Julia’s compiler cannot infer a concrete result type for a variable or expression; this inhibits specialization and often triggers performance degradation through dynamic dispatch. A separate but related performance trap is using abstractly typed containers (e.g., `Vector{Any}`) in hot loops, which forces dynamic dispatch on each element even if the function’s overall return type is inferrable.

To avoid type instabilities, ensure that functions have inferrable, concrete types across all code paths. This includes ensuring that all variables used within functions are either local or passed as arguments — accessing global variables from within a function inhibits type inference (see also Section 24.2). For example:

```
function unstable_example(array)
    x = []
    for y in array
        push!(x, y)
    end
    sum(x)
end

function stable_example(array)
    x = eltype(array)[]
    for y in array
        push!(x, y)
    end
    sum(x)
end
```

①

②

10. Writing Performant Single-Threaded Code

```
data = rand(1000)
@btime unstable_example(data)
@btime stable_example(data)
```

- ① Without a type given, [] will create a `Vector{Any}` which can contain elements of `Any` type which is flexible but requires runtime dispatch to determine correct behavior.
- ② `eltype(array)` returns the type contained within the array, which for `data` is `Float64`. Thus `x` is created as a `Vector{Float64}` which is more efficient code.

```
11.042 μs (2008 allocations: 49.91 KiB)
809.735 ns (10 allocations: 18.69 KiB)
```

497.2881916663594

The `unstable_example` function illustrates a common anti-pattern wherein an `Any` typed array is created and then elements are added to it. Because any type can be added to an `Any` array (we happen to just add floats to it) then Julia's not sure what types to expect inside the container and therefore has to determine it at runtime.

Note

Heterogeneous return types are not the same thing as type instability. In the example above, the return type is not unstable: the compiler recognizes that the single parametric type `Union{Float64, Int64}` will be returned. While Julia can optimize some unions with a small number of elements, you should prefer concrete, predictable types in hot code.

Tip

See Section 24.6.1 and Section 24.6.2 for tools in Julia to troubleshoot type instabilities.

10.3.5. Optimize for Branch Prediction

Modern CPUs use branch prediction to speculatively execute instructions before knowing the outcome of conditional statements. This was described in the prior chapter in the logistics warehouse analogy as trying to predict where a package will be going before you've inspected the label. CPUs execute one branch of the instructions without seeing the data (either true data, or data in the form of CPU instructions) because the CPU has higher speed/capacity than the memory throughput allows. This is another example of a technique developed for performance in a memory-throughput constrained world.

Optimizing your code for branch prediction can significantly improve performance, especially in tight loops or frequently executed code paths.

To optimize for branch prediction:

1. Structure your code to make branching patterns more predictable. For instance, in if-else statements, put the more likely condition first. This allows the CPU to more accurately predict the branch outcome.
2. Use loop unrolling to reduce the number of branches. This technique involves manually repeating loop body code to reduce the number of loop iterations and associated branch instructions. See Section 11.4 for more on what this means.

3. Consider using Julia's `@inbounds` macro to eliminate bounds checking in array operations when you're certain the accesses are safe. This reduces the number of conditional checks the CPU needs to perform.
4. For performance-critical sections with unpredictable branches, consider using branch-free algorithms or bitwise operations instead of conditional statements. This can help avoid the penalties associated with branch mispredictions.
5. In some cases, it may be beneficial to replace branches with arithmetic operations (e.g., using the ternary operator or multiplication by boolean values) to allow for better vectorization and reduce the impact of branch mispredictions. An example of this would be using a statement like `y += max(x, 0)` instead of `if x > 0; y += x; end`.

Here's an example demonstrating the impact of branch prediction:

```
function process_data(data, threshold)
    total = 0.0
    for x in data
        if x > threshold
            total += log(x)
        else
            total += x
        end
    end
    total
end

# Random data = unpredictable branches
rand_data = rand(1_000_000)

# Sorted data = predictable branches
sorted_data = sort(rand(1_000_000))

@btime process_data($rand_data, 0.5)
@btime process_data($sorted_data, 0.5);

5.150 ms (0 allocations: 0 bytes)
1.663 ms (0 allocations: 0 bytes)
```

In this example, having sorted data means that the CPU will predict which branch of the `if` statement is likely to be utilized. Sorting makes the branch outcome highly predictable for long stretches (all below-threshold values, then all above), improving the effectiveness of the branch predictor. By speculatively executing the code that it thinks will be used, the overall program time is faster when processing `sorted_data`.

Remember that optimizing for branch prediction often involves trade-offs. The benefits can vary depending on the specific hardware and the nature of your data. If performance critical, profile your code to ensure that your optimizations are actually improving performance in your specific use case. Over-optimizing on one set of hardware (e.g. local computer) may not translate the same on another set of hardware (e.g. server deployment).

10.3.6. Further Reading

- What scientists must know about hardware to write fast code
- Optimizing Serial Code, SciML Book

10.3.7. Key Takeaways

- Measure first: use `@btime`, `@allocated`, and profiling tools to find hot loops before redesigning algorithms.
- Favor cache-friendly access patterns; touching memory in the order it is stored often doubles throughput for actuarial grids or scenario cubes.
- Keep containers concrete and preallocate buffers in tight loops to avoid hidden heap allocations and dynamic dispatch.
- Start with `Float64/Int64` unless memory pressure forces a smaller type; benchmark any change.
- Make branches predictable when you can and sort or bucket data if it helps align sequential behavior.

11. Parallelization

CHAPTER AUTHORED BY: ALEC LOUDENBACK

Quantity has a quality all its own. - Attributed to Vladimir Lenin

11.1. Chapter Overview

Fundamentals of parallel workloads, different mechanisms to distribute work: vectorization (SIMD), multi-threading, GPU, and multi-device workflows. Different programming models: map-reduce, arrays, and tasks.

11.2. Amdahl's Law and the Limits of Parallel Computing

An important ground truth in computing is that there is an upper limit to how fast a workload can be sped up through distributing the workload among multiple processor units. For example, if there is a modeling workload wherein 90% of the work is independent (say policy or asset level calculations) and the remaining 10% of the workload is an aggregate (say company or portfolio level), then the theoretical maximum speedup of the process is 10x faster (1 / 10% serial load). This is captured in a law known as **Amdahl's Law**, and it reflects the *theoretical* maximum speedup a workload could see. In practice, the speedup is worse than this due to overhead of moving data around, scheduling the tasks, and aggregating results. This is why in many cases a good effort in sequential workloads (see Chapter 10) is often a more fruitful effort than trying to parallelize some workloads.

That said, there are still many modeling use-cases for parallelization. Modern investment and insurance portfolios can easily contain hundreds of thousands or millions of seriatim holdings. In many cases, these can be evaluated independently, though often there is interaction with the total portfolio (contract dividends, non-guaranteed elements, profit sharing, etc.). Further, even if the holdings are not parallelizable across the holdings dimension, we are often interested in independent evaluations across economic scenarios, which is amenable to parallelization.

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

Where:

- $S(n)$ is the theoretical speedup of the execution of the whole task
- n is the number of processors
- p is the proportion of the execution time that benefits from improved resources

We can visualize this for different combinations of p and n in Figure 11.1.

```
using CairoMakie
```

11. Parallelization

```
function amdahl_speedup(p, n)
    return 1 / ((1 - p) + p / n)
end

function main()
    fig = Figure()
    ax = Axis(fig[1, 1],
              title="Amdahl's Law",
              xlabel=L"Number of processors ($n$)",
              ylabel="Speedup",
              xscale=log2,
              xticks=2 .^ (0:16),
              xtickformat=x -> "2^" .* string.(Int.(log.(2, x))),
              yticks=0:2:20
    )
    n = 2 .^ (0:16)
    parallel_portions = [0.5, 0.75, 0.9, 0.95]
    linestyles = [:solid, :dash, :dashdot, :solid]
    for (i, p) in enumerate(parallel_portions)
        speedup = [amdahl_speedup(p, ni) for ni in n]
        lines!(ax, n, speedup, label="$(Int(p*100))%", 
               ↴ linestyle=linestyles[i])
    end
    xlims!(ax, 1, 2^16)
    ylims!(ax, 0, 20)
    axislegend(ax, L"Parallel portion ($p$)", position=:lt)
    fig
end

main()
```

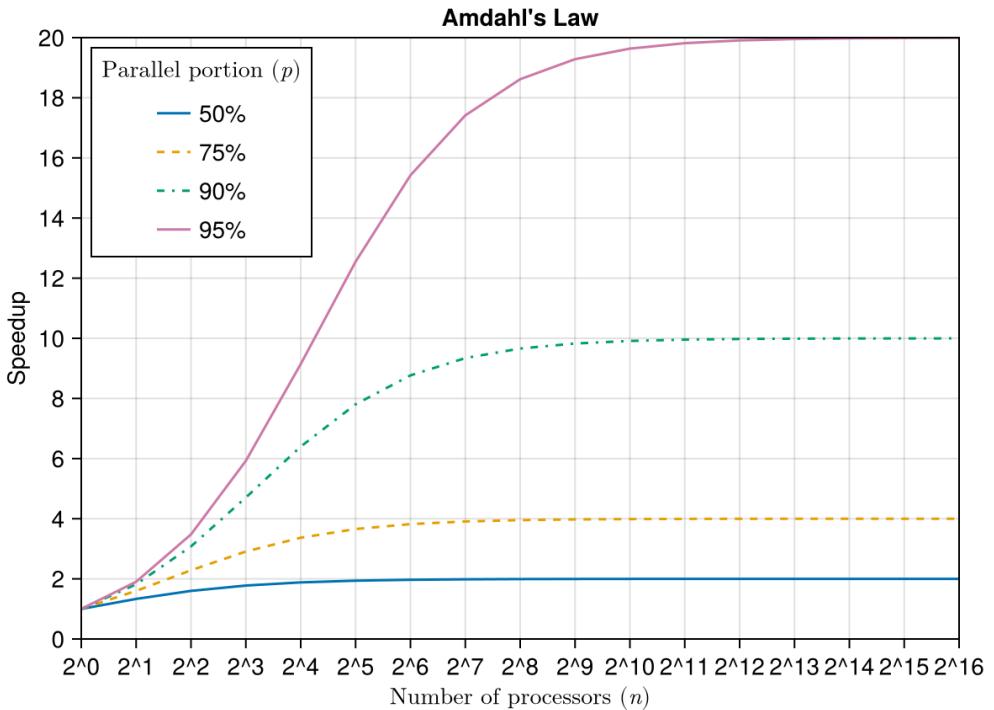


Figure 11.1.: Theoretical upper bound for speedup of a workload given the parallelizable portion p and number of processors n .

A real-world analogy: imagine building a house. You can hire many workers to frame the walls, install plumbing, and run electrical wires simultaneously (the parallel part). However, all work must stop to wait for the single foundation to be poured and cured (the serial part). No matter how many workers you add, the total project time can never be faster than the time it takes to cure the foundation. This fundamental bottleneck is what Amdahl's Law describes.

With this understanding, we will be able to set expectations and analyze the benefit of parallelization.

11.3. Types of Parallelism

Parallel processing comes in different flavors and is related to the details of hardware as discussed in Chapter 9. We will necessarily extend the discussion of hardware here, as parallelization is (mostly) inextricably tied to hardware details. We revisit this in Section 11.8 when we connect the techniques back to programming abstractions and real models.

11. Parallelization

Table 11.1.: Major types of computational parallelism highlighting their key characteristics, advantages, and potential drawbacks.

Type	Description	Strengths	Weaknesses
Vectorization (SIMD)	Performs same operation on multiple data points simultaneously	Efficient for data-parallel tasks, uses specialized CPU instructions	Limited to certain types of operations, data must be contiguous
Multi-Threading	Executes multiple threads concurrently on a single CPU	Good for task parallelism, utilizes multi-core processors effectively	Overhead from thread management, potential race conditions
GPU	Uses graphics processing units (GPUs) for parallel computations	Excellent for massively parallel tasks, high throughput	Specialized programming required, data transfer overhead
Multi-Device / Distributed	Spreads computation across multiple machines or devices	Scales to very large problems, can use heterogeneous hardware	Complex to implement and manage, network latency issues

11.4. Vectorization

Vectorization in the context of parallel processing refers to special circuits within the CPU wherein the CPU will load multiple data units (e.g. 4 or 8 floating point numbers) in a contiguous block and perform the same instruction on them at the same time. This is also known as **SIMD, or Single-Instruction Multiple Data**. Think of it like a for loop where each iteration does four or eight operations at a time instead of one.

The requirements for SIMD-able code are that:

- The intended section for SIMD is inside the inner-most loop.
- There are no branches (if-statements) inside the loop body.

💡 Tip

Note that indexing an array actually introduces a branch in the code, as two cases could arise: the index is either inbounds or out-of-bounds. To avoid this, either use `for x in collection`, `for i in eachindex(collection)` or `for i in 1:n; @inbounds collection[i]`, though the last of these is discouraged in favor of the prior, safer options.

```
using BenchmarkTools

function prevent_simd(arr)
    sum = 0
    for x in arr
        if x > 0
            sum += x
        end
    end
end
```

```

        return sum
end

function allow_simd(arr)
    sum = 0
    for x in arr
        sum += max(x, 0)
    end
    return sum
end

let
    x = rand(10000)

    @btime prevent_simd($x)
    @btime allow_simd($x)
end

```

44.166 µs (0 allocations: 0 bytes)
4.821 µs (0 allocations: 0 bytes)

4965.163320122393

In testing the above code, the `allow_simd` version should be several times faster than the `prevent_simd` example. The reason is that `prevent_simd` has a branch (`if x > 0`) where the behavior of the code may change depending on the value in `arr`. Conversely, the behavior of `allow_simd` is always the same in each iteration, no matter the value of `x`. This allows the compiler to generate vectorized code automatically.

Tip

Note that Julia's compiler is able to identify vectorizable code in many cases, though some cases may benefit from a more manual hint to the compiler through macro annotations (enter `?@simd` in the REPL for details). See Section 24.9.4 for more.

Other types of parallelism that we will discuss in this chapter have some risk of errors or data corruption if not used correctly. SIMD isn't prone to issues like this because if the code is not SIMD-able then the compiler will not auto-vectorize the code block. Note that this safety guarantee applies to Julia's built-in `@simd` macro. Third-party macros like `@turbo` (from `LoopVectorization.jl`) are more aggressive and may produce incorrect results if loop iterations have data dependencies (e.g., `x[i] = x[i-1]`).

11.4.1. Hardware

Vectorization is hardware dependent. If the CPU does not support vectorization you will not see speedups from it. Many consumer and professional chips have AVX2 (Advanced Vector Extensions 2), which uses 256-bit registers allowing four simultaneous 64-bit floating-point operations. AVX-512, with 512-bit registers (eight simultaneous 64-bit operations), is common on Intel server chips and recent AMD processors. However, AVX-512 can cause thermal throttling on some chips—wider SIMD uses more power and generates more heat—so real-world speedups depend on sustained workloads and cooling.

On ARM processors (AArch64), the equivalent is NEON (128-bit, two 64-bit operations) and the newer SVE/SVE2 (Scalable Vector Extension) which supports variable-width vectors up to 2048 bits. Apple's M-series chips use NEON; ARM server chips like AWS Graviton3 support SVE.

11.5. Multi-Threading

This subsection starts by introducing *tasks* - lightweight units of computation. Next, we will see how tasks can communicate using *channels*, and then how multiple tasks (and channels) can be leveraged to achieve true parallelism through *multi-threading*. Think of it as layers building on one another: tasks define work units, channels allow them to share data, and multi-threading enables tasks to run simultaneously on multiple CPU cores. Exact details regarding tasks, channels, and multi-threading vary by language but the general ideas remain the same.

11.5.1. Tasks

To understand multi-threading examples, we first need to discuss **Tasks**, which are chunks of computation that get performed together, but after which the computer is free to switch to a new task. Technically, there are some instructions within a task that will let the computer pause and come back to that task later (such as `sleep`).

Tasks do not, by themselves, allow for multiple computations to be performed in parallel. For example, one task might be loading a data file from persistent storage into RAM. After that task is complete, the computer continues on with another task in the queue (rendering a web page, playing a song, etc.). In this way even a single processor computer could be “doing multiple things at once” (or “multi-tasking”) even though nothing is running in parallel. The scheduling of the tasks is handled automatically by the program’s compiler or the operating system.

Here’s an example of a couple of tasks where we write to an array. Despite being called last, the second task should actually write to the array before the first task. This is because we asked the first task to `sleep` (pause, while allowing the computer to yield to other tasks in the queue)¹.

```

let
    shared_array = zeros(5)

    task1 = @task begin
        sleep(1)
        shared_array[1] = 1

        println("Task 1: ", shared_array)
    end

    task2 = @task begin
        shared_array[2] = 2
        println("Task 2: ", shared_array)
    end

    schedule(task1)
    schedule(task2)
    wait(task1)
    wait(task2)

    println("Main: ", shared_array)
end

```

¹Technically, it’s possible that the second task doesn’t write to the array first. This could happen if there are enough tasks (from our program or others on the computer) that saturate the CPU during the first task’s `sleep` period such that the first task gets picked up again before the second one does.

```
Task 2: [0.0, 2.0, 0.0, 0.0, 0.0]
Task 1: [1.0, 2.0, 0.0, 0.0, 0.0]
Main: [1.0, 2.0, 0.0, 0.0, 0.0]
```

11.5.1.1. Channels

Channels are a way to communicate data in a managed way between tasks. You specify a type of data that the buffer (a chunk of assigned memory) will contain and how many elements it can hold. It then stores items (via `put!`) in a first-in-first-out (FIFO) queue, which can be popped off the queue (via `take!`) by other tasks.

Here's an example of a system which generates trades in the financial markets at random time intervals, and a monitoring task takes the results and tabulates running statistics:

```
let

    # simulate random trades and the associated profits
    function trade_producer(channel, i)
        sleep(rand())
        profit = randn()
        put!(channel, profit)
        println("Producer: Trade Result #$i $(round(profit, digits=3))")
    end

    # intake trades via the communication channel
    function portfolio_monitor(channel, n)
        sum = 0.0
        for _ in 1:n
            profit = take!(channel)
            sum += profit
            p = round(profit, digits=3)
            s = round(sum, digits=3)
            println("Monitor: Received $p, Cumulative: $s")
        end
    end

    channel = Channel{Float64}(32)

    # Start producer and consumer tasks
    @sync begin
        for i in 1:5
            @async trade_producer(channel, i)
        end
        @async portfolio_monitor(channel, 5)
    end

    # Close the channel and wait for tasks to finish
    close(channel)
end
```

- ① Random sleep between 0 and 1 seconds to simulate real trading activity and latency.
- ② Generate a random number from standard normal distribution to simulate profit or loss from a trade.

11. Parallelization

- ③ In this teaching example, we've limited the system to produce just five "trades". In practice, this could be kept running indefinitely via, e.g., `while true`.
- ④ Create a channel with a buffer size of 32 floats (in this limited example, we could have gotten away with just 5 since that's how many the demonstration produces). In practice, you want this to be long enough that the consumer of the channel never gets so far behind that the channel fills up. The channel is created outside of the `@sync` block so that `channel` is in scope when we close it.
- ⑤ `@sync` waits (like `wait(task)`) for all of the scheduled tasks within the block to complete before proceeding with the rest of the program.
- ⑥ `@async` does the combination of creating a task via `@task` and `schedule-ing` in one, simpler call.

```
Producer: Trade Result #1 0.193
Monitor: Received 0.193, Cumulative: 0.193
Producer: Trade Result #3 0.608
Producer: Trade Result #2 0.77
Monitor: Received 0.608, Cumulative: 0.801
Monitor: Received 0.77, Cumulative: 1.571
Producer: Trade Result #4 -0.913
Monitor: Received -0.913, Cumulative: 0.658
Producer: Trade Result #5 0.723
Monitor: Received 0.723, Cumulative: 1.382
```

This is really useful for handling events that are "external" to our program. If we were just doing a modeling exercise using static data, then we could control the order of processing and not need to worry about monitoring a volatile source of data. Nonetheless, tasks can still be useful in some cases even if a model is not using "live" data: for example if one of the steps in a model is to load a very large dataset, it may be possible to perform some computations while chunked task requests are queued to load more data from the disk.

A garbage collector will usually clean up unused channels that are still open. However, it's a good practice to explicitly close them to ensure proper resource management, clear signaling of completion, and to avoid potential blocking or termination issues in your programs.

Caution

If the task never finishes properly inside the `@sync`, then your program may get stuck in an infinite loop and hang. Such as if one of the tasks never has a termination condition such as an upper bound on a loop, or a clear way to break out of a `while true` loop. While not different than a normal loop, such issues become less obvious underneath the layer of task abstractions.

The key takeaway for tasks is that it's a way to chunk work into bundles that can be run in a concurrent fashion, even if nothing is technically being processed in parallel. The multi-threading and parallel programming paradigms sections build off of tasks so an understanding of tasks is helpful. However, some of the higher level libraries hide the task-based building blocks from you as the user/developer and so an intricate understanding of tasks is not required to be successful in parallelizing your Julia code.

11.5.2. Multi-Threading Overview

When a program starts on your computer, the operating system creates a **process**. It allocates bookkeeping structures (to track code, stack frames, and heap allocations) and reserves a block of memory in RAM for that process. Different processes do not have access to each other's allocated memory.

i Note

Readers may be familiar with starting Excel in different processes. When workbooks are opened within the same process (e.g. when creating a new workbook from Excel's File menu), the workbooks may seamlessly talk to each other (copy and paste from one to another). However, when Microsoft Excel is opened in different processes, then the workbooks in each respective process do not share memory and cannot create links or use full copy/paste functionality between them (this is what happens when you hold the control button and open Excel multiple times).

Within each process a main thread is created. That thread is where the running of the code occurs. For the level of the discussion here, you can think of a process as a container with shared memory for threads, which do the real work (as illustrated in Figure 11.2). Besides the main thread, other threads can be created within the process and access the same shared memory.

💡 Tip

Set the number of Julia threads explicitly via the `JULIA_NUM_THREADS` environment variable or the `--threads` flag (for example, `julia --threads=auto`). This must be done before the session starts; changing it inside a running REPL has no effect.

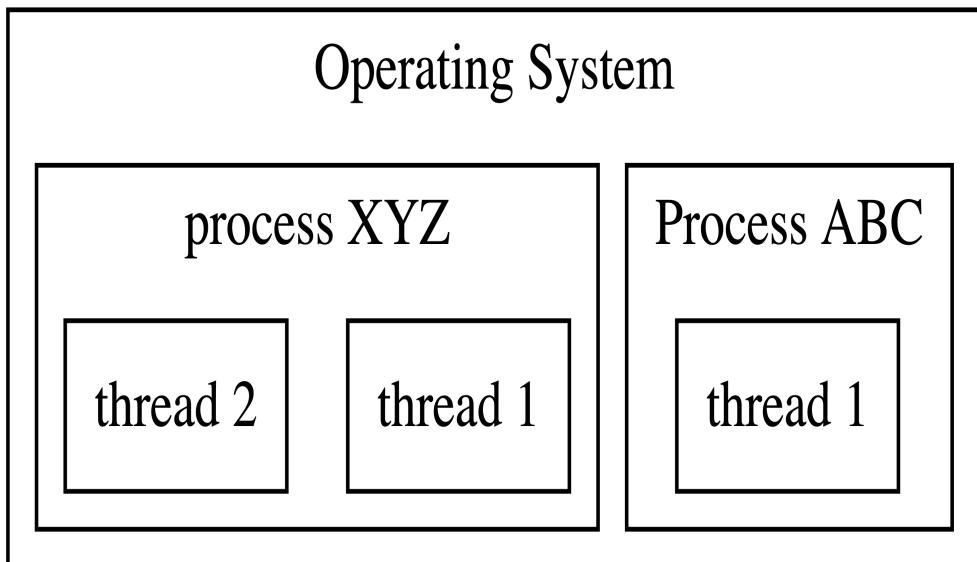


Figure 11.2.: When a program starts, the operating system creates a process for which multiple threads (a *main* thread plus optional additional threads) share memory.

The advantage of threads is that within a single physical processor there may be multiple cores. Those cores can access the shared process memory and run tasks from different threads simultaneously. Modern processors commonly have 8-16 cores (consumer) or 64-128+ cores (server), making multi-threading an effective way to utilize available hardware.

i Note

Thread jargon varies. A compact glossary to help:

- **Hardware threads** are the execution lanes exposed by the CPU. They are what the operating system schedules on each core.
- **Operating-system threads** are managed by the kernel and generally map one-to-one to hardware threads. They are heavier to create and destroy than Julia tasks.
- **Green threads, cooperative threads, fibers, or user threads** are Julia's lightweight tasks. They live entirely inside the Julia runtime and can be created in large numbers with minimal overhead.
- **Multi-tasking** is a single core rapidly switching between tasks so multiple activities appear to make progress even without true parallel execution.

Parallelism occurs at many layers - hardware, operating system, language runtime, and network - and the terminology is often overloaded. When collaborating, be explicit about which kind of "thread" you mean.

11.5.2.1. Multi-Threading Pitfalls

Different threads being able to access the same memory is a double-edged sword. It is useful because we do not need to create multiple copies of the data in RAM or in the cache² and can improve the overall throughput of our usually memory-bandwidth-limited machines. The downside is that if we are mutating the shared data for which our program relies upon, then our program may produce unintended results if the modification occurs carelessly. There are a couple of related issues to be aware of:

11.5.2.1.1. Race Conditions

The first issue is known as a **race condition**, which occurs when a block of memory has been read from or written to in an unintended order. For example, if we have two threads which are accumulating a sub-total, each process may read the running sub-total before the other thread has finished its update.

In the following example, we use the `Threads.@threads` to tell Julia to automatically distribute the work across threads.

```
function sum_bad(n)
    subtotal = 0
    Threads.@threads for i in 1:n
        subtotal += i
    end
    subtotal
end

sum_bad(100_000)
```

470023130

The result will be less than the expected sum (5000050000) due to a race condition. Here's what happens:

²There are some chips which do not have access to the same memory in a multi-threading context, and are known as non-uniform memory access (NUMA). These architectures work more like those in Section 11.7.

1. Multiple threads read the current value of `subtotal` simultaneously.
2. Each thread adds its own, local value to that reading.
3. Only one thread writes its result back to `subtotal` first.
4. A different thread then overwrites `subtotal` with its calculation based on the out-dated starting point for `subtotal`.

This means some thread contributions are lost when they overwrite each other's results. The threads may not see each other's updates, leading to missing values in the final sum.

11.5.2.2. Avoiding Multi-threading Pitfalls

We will cover several ways to manage multi-threading race conditions, but it is the recommendation of the authors to primarily utilize higher level library code, which will be demonstrated after covering some of the more basic, manual techniques.

11.5.2.2.1. Chunking up work into single-threaded work

First, let's level-set with a single-threaded result:

```
function sum_single(a)
    s = 0
    for i in a
        s += i
    end
    s
end
@btime sum_single(1:100_000)
```

1.666 ns (0 allocations: 0 bytes)

5000050000

Note that in the single-threaded case, Julia is able to identify this common pattern and use a shortcut, calculating the sum of the integers 1 through n as $\frac{n(n+1)}{2}$ through a compiler optimization and essentially avoid the loop entirely.

We can implement a correct threaded version by splitting the work into different threads, each of which is independent. Then, we can aggregate the results of each of the chunks.

```
function sum_chunker(a)

    chunks = Iterators.partition(1:a, a ÷ Threads.nthreads())           ①

    tasks = map(chunks) do chunk
        Threads.@spawn sum_single(chunk)
    end

    chunk_sums = fetch.(tasks)                                         ③

    return sum_single(chunk_sums)                                       ④

end
```

11. Parallelization

```
| @btime sum_chunker(100_000)
```

- ① Split 1:a into roughly equal ranges across available threads.
- ② Launch a task for each chunk to compute a single-threaded sum.
- ③ Fetch each task's result once it finishes.
- ④ Aggregate the partial sums on a single thread; this is cheap because there are only as many partial sums as threads.

```
9.208 μs (72 allocations: 4.16 KiB)
```

```
5000050000
```

11.5.2.2. Using Locks

Locks prevent memory from being accessed from more than one thread at a time.

```
function sum_with_lock(n)
    subtotal = 0
    lock = ReentrantLock()
    Threads.@threads for i in 1:n
        Base.@lock lock begin
            subtotal += i
        end
    end
    subtotal
end
@btime sum_with_lock(100_000)
```

- ① Initialize a running total to zero.
- ② Create a reentrant lock to ensure only one thread updates `subtotal` at a time.
- ③ Parallelize the loop over 1 to n using `Threads.@threads`.
- ④ Acquire the lock before updating the shared variable `subtotal`. This ensures that only one thread updates `subtotal` at a time, preventing race conditions. The lock is automatically released at the end of the block.

```
5.912 ms (199514 allocations: 3.05 MiB)
```

```
5000050000
```

11.5.2.2.3. Using Atomics

Atomics are certain primitive values with a reduced set of operations for which Julia and the compiler can automatically create thread-safe code. This is often significantly faster than the context-switching overhead needed with locking and unlocking memory for threaded tasks. Compared with locks, atomics are simpler to implement and easier to reason about. The downside is that atomics are limited to the available primitive atomics types and methods.

```
function sum_with_atomic(n)
    subtotal = Threads.Atomic{Int}(0)
    Threads.@threads for i in 1:n
        Threads.atomic_add!(subtotal, i)
    end

```

```

        subtotal[]
end

@btime sum_with_atomic(100_000)

```

- ① Initialize an atomic integer (`Threads.Atomic{Int}`) to store the subtotal. An atomic variable ensures that increments are performed atomically, preventing race conditions without needing explicit locks.
- ② Atomically add `i` to `subtotal`. The `Threads.atomic_add!` function ensures that the addition and update of `subtotal` happens as one atomic step, preventing multiple threads from interfering with each other's updates.

`1.300 ms (53 allocations: 3.75 KiB)`

`5000050000`

11.6. GPUs and TPUs

11.6.1. Hardware

Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs) are hardware accelerators for massively parallel computations. A TPU is very similar to a GPU but has a special ability to handle data types and instructions that are more specialized for linear algebra operations; going forward we will simply refer to these types of accelerators as GPUs.

GPUs have similar components as the CPU as discussed in Chapter 9. They have RAM, caches for the cores, and cores that run the coded instructions on the data. The differences from a CPU are primarily:

- A GPU typically has thousands of simple cores, while CPUs have fewer but more sophisticated cores (4-16 on consumer chips, up to 128+ on server chips like AMD EPYC or Intel Xeon).
 - GPU cores operate at *slower* clock speeds than CPU cores (1-2 GHz vs 3-5 GHz) but achieve throughput through massive parallelism.
- The GPU cores essentially have to be running the same set of instructions on all of the data, not unlike vectorization (Section 11.4).
 - GPU code is not suited for code with branching conditions (e.g. `if` statements) and so is more limited in the kinds of computations it can handle compared to the CPU.
- GPU memory (VRAM) is typically more constrained than system RAM. Consumer GPUs have 8-24 GB; high-end data center GPUs (NVIDIA H100) have up to 80 GB, while system RAM can easily be 128-512 GB on servers.
 - As a result, GPUs may need strategies to move chunks of data to and from GPU memory for large datasets. Using lower-precision datatypes (e.g., `Float16` or `Float32` instead of `Float64`) is common to fit more data and improve throughput, though this trades off numerical precision.
- The caches are similar in concept to CPU, but unlike most CPU caches, there is relative locality to data (wherein core #1 will have much quicker access to a different subset of data than, say, core #1024).
- A GPU is usually a secondary device of sorts: it physically and in device architecture is separate from the CPU. The CPU remains in charge of overall computer execution.

11. Parallelization

- The implication of this (as with any movement of memory) is that there is overhead to moving data to and from the GPU. Your calculations will need to be in the single milliseconds range of time in order to start to see benefit from utilizing a GPU.
- To some extent, separable CPU, RAM, and GPU are changing with some of the latest computer hardware. For example, the M-series of Apple processors have the CPU, GPU, and RAM in a single tightly integrated package for efficiency and computational power.

11.6.1.1. Notable Vendors and Libraries

Like the difference between x86 and ARM architectures, GPUs also have specific architectures which vary by the vendor. To make full use of the hardware, the vendors need to (1) provide device drivers which allow the CPU to talk to the GPU, and (2) provide the libraries (lower level application programming interfaces, or APIs) which allow developers to utilize different hardware features without needing to write machine code.

As of the mid 2020s, here are the most important GPU vendors and the associated programming library for utilizing their specific hardware:

Table 11.2.: GPU and TPU vendors with representative hardware and programming interfaces. NVIDIA dominates the data center market; Apple Metal is only available on macOS/iOS.

Vendor	Hardware Examples	API/Library
NVIDIA	GeForce RTX (consumer), Quadro/RTX A-series (professional), A100/H100/H200 (data center)	CUDA
AMD	Radeon (consumer), Radeon Pro (professional), Instinct MI series (data center)	ROCM (HIP)
Intel	Arc (consumer), Data Center GPU Max / Ponte Vecchio (data center)	oneAPI (SYCL)
Apple	M-series integrated GPU (M1, M2, M3, M4)	Metal
Google	TPU v4, v5 (cloud only)	XLA (via JAX or TensorFlow)

11.6.2. Utilizing a GPU

With some of the key conceptual differences between CPUs and GPUs explained, let's explore how to incorporate these powerful hardware accelerators. We will use Julia libraries to illustrate GPU programming, though the concepts are generally applicable to other high-level languages that offer GPU interface libraries.

11.6.2.1. Julia GPU Libraries

There's essentially two types of GPU programming we will discuss here:

1. **Array-based programming**, where arrays are stored and operated on directly on the GPU memory. This approach abstracts away the low-level details, allowing you to work with familiar array operations that are automatically executed on the GPU.

2. **Custom kernels**, which are specialized functions that define exactly how each GPU thread should process data in parallel. A kernel explicitly specifies the computation that each GPU thread will perform on its portion of the data.

i Note

Kernels in this context are specialized functions that run directly on the GPU. Rather than relying solely on high-level array operations, kernels explicitly define the sequence of low-level, parallel instructions executed across many GPU threads. In other words, a kernel directly expresses the computation you want to perform on the data, enabling fine-grained control over GPU execution.

A third approach would be to implement GPU code in a low-level vendor toolkit (such as C++ and associate CUDA libraries), but this approach will not be illustrated here.

Julia has wonderful support for several of the primary vendors (at the time of writing, CUDA, Metal, OneAPI, and ROCm) via the JuliaGPU organization. Installation of the required dependencies is also very straightforward and the interfaces at the array and generated kernel levels are very similar. The differences are obvious at the lower level vendor-API wrappers (which is the lower-level technique that will not be covered here).

The benefit of the consistency of the higher level libraries we will use here is that examples written for one of the types of accelerators will be largely directly translatable to another. This is especially true for array programming, though less so for the kernel style as architecture-specific considerations often creep in³.

i Note

This book will be rendered on a Mac and therefore the examples will use Metal in order to run computational cells, however we'll show a CUDA translation for some of the examples in order to show how straightforward translating higher level GPU code in Julia is.

GPU API	GPU Array Type	Kernel Macro
CUDA	CuArray	@cuda
Metal	MtlArray	@metal
oneAPI	oneArray	@oneAPI
ROCm	ROCArrray	@roc

11.6.2.2. Array Programming on the GPU

First described in Section 6.5, array programming eschews writing loops and instead favors initializing blocks of heap-allocated memory and filling it with data to be operated on at a single point in time. While this is often not the most efficient way to utilize CPUs, it's essentially the required style of code to utilize GPUs.

For the example below, we will calculate the present value of a series of cashflows across a number of different scenarios. An explanation of the code is given below the example.

³The KernelAbstractions.jl library actually allows you to write generic kernels which then get compiled into different code depending on the backend you are using.

11. Parallelization

```
using Metal

function calculate_present_values!(present_values, cashflows,
    ↳ discount_matrices)
    # Perform element-wise multiplication and sum along the time dimension
    present_values .= sum(cashflows .* discount_matrices, dims=1)      ①
end

# Example usage using 100 time periods, 100k scenarios
num_scenarios = 10^5
pvs = zeros(Float32, 1, num_scenarios)
cashflows = rand(Float32, 100)                                         ③
discount_matrices = rand(Float32, 100, num_scenarios)                  ④

# copy the data to the GPU
pvs_GPU = MtlArray(pvs)
cashflows_GPU = MtlArray(cashflows)
discount_matrices_GPU = MtlArray(discount_matrices)                      ⑤

@btime calculate_present_values!($pvs, $cashflows, $discount_matrices)
@btime calculate_present_values!($pvs_GPU, $cashflows_GPU,
    ↳ $discount_matrices_GPU)                                              ⑥
```

- ① The function `calculate_present_values!` is written the same way as if we were just writing CPU code. Note that we are also passing a pre-allocated vector, `present_values` to store the result. This will allow us to isolate the performance of the computation, rather than including any overhead of allocating the array for the result.
- ② The code is broadcasted across the first dimension so that the single set of cashflows is discounted for each scenario's discount vector.
- ③ Metal only supports 32-bit floating point (`Float32`) in GPU shaders. NVIDIA data center GPUs (A100, H100) have full-speed FP64; consumer GPUs (RTX series) have heavily throttled FP64 (often 1/64th the FP32 rate). For financial calculations requiring double precision, this is an important hardware consideration.
- ④ Using 100 thousand scenarios for this example.
- ⑤ `MtlArray(array)` will copy the array values to the GPU.
- ⑥ Note that the data still lives on the GPU and is of the `MtlMatrix` (a type alias for a 2-D `MtlArray`).

```
1.141 ms (6 allocations: 38.55 MiB)
91.125 μs (316 allocations: 9.98 KiB)

1x100000 MtlMatrix{Float32, Metal.PrivateStorage}:
24.9435 25.9882 24.4573 27.4175 ... 24.6801 25.4949 24.0078 27.4271
```

The testing suggests significantly faster computation when performed on the GPU (the exact speedup depends on hardware and problem size). Note, however, that this does not include the overhead of (1) moving the data to the GPU (in the initial `MtlArray(cashflows)` call), or (2) returning the data to the CPU (since the return type for the GPU version is `MtlArray`). We can measure this overhead by wrapping the data transfer inside another function and benchmarking it:

```
function GPU_overhead_test(present_values, cashflows, discount_matrices)
    pvs_GPU = MtlArray(present_values)
    cashflows_GPU = MtlArray(cashflows)
    discount_matrices_GPU = MtlArray(discount_matrices)                      ⑤
```

```

calculate_present_values!(pvs_GPU, cashflows_GPU, discount_matrices_GPU)

Array(pvs_GPU) # convert to CPU array
end

@btime GPU_overhead_test($pvs, $cashflows, $discount_matrices)

5.101 ms (514 allocations: 415.78 KiB)

1x100000 Matrix{Float32}:
24.9435 25.9882 24.4573 27.4175 ... 24.6801 25.4949 24.0078 27.4271

```

With the additional overhead, the computation on the GPU takes more total time than if the work were done just on the CPU. This is a very simple example, and the balance tips heavily in favor of the GPU when:

1. The computational demands are significantly higher (e.g. we were to do more calculations than just a simple multiply/divide/sum).
2. The data size grows bigger.

i Note

The previous example can be translated to CUDA by simply exchanging `MtArray` for `CuArray`.

⚠ Warning

This example again underscores that hardware parallelization is not an automatic “win” for performance. A lot of uninformed discussion around modeling performance is to simply try to get things to run on the GPU and it is often *not* the case that the models will run faster. Further, as the modeling logic gets more complex, it does require greater care to keep in mind GPU constraints (acceptable data types, memory limitations, avoiding scalar operations, data transfer between CPU and GPU, etc.). A best practice is to contemplate sequential performance and memory usage before leveraging GPU accelerators.

11.6.2.3. Kernel Programming on the GPU

Another approach to GPU programming is often referred to as kernel programming, or being much more explicit about *how* a computation is performed. This is as opposed to the declarative approach in the array-oriented style (Section 11.6.2.2) wherein we specified *what* we wanted the computation to be.

The key ideas here are that we need to manually specify several aspects which came ‘free’ in the array-oriented style. The tradeoff is that we can be more fine-tuned about how the computation leverages our hardware, potentially increasing performance.

The GPU libraries in Julia abstract much of the low level programming typically necessary for this style of programming, but we still need to explicitly look at:

1. How the GPU will iterate across different cores/threads.
2. How many threads to utilize, the optimal number depends on the shape of the computation (long vectors, multi-dimensional arrays), memory constraints, and hardware specifics.

11. Parallelization

- GPU threads: Individual units of execution within a kernel. Each thread runs the same kernel code but operates on a different portion of the data.

3. How to chunk (group) the data to distribute the data to the different GPU threads

Our strategy for the present values example will be to distribute the work such that different GPU threads are working on different scenarios. Within a scenario, the loop is a very familiar approach: initialize a subtotal to zero and then accumulate the calculated present values.

```
function calculate_present_values_kernel!(present_values, cashflows,
    ↳ discount_matrices)
    idx = thread_position_in_grid_1d() ①

    pv = 0.0f0 ②
    for t in 1:size(cashflows, 1)
        pv += cashflows[t] * discount_matrices[t, idx]
    end ③

    present_values[idx] = pv ④
    return nothing
end ⑤
```

- ① As the work is distributed across threads, `thread_position_in_grid_1d()` will give the index of the current thread so that we can index data appropriately for the work as we decide to split it up (we've split up the work by scenario in this example).
- ② Recall that we are working with `Float32` on the GPU here, so the zero value is set via the `f0` notation indicating a 32-bit floating point number.
- ③ The loop is across timesteps within each thread, while the thread index is tracked with `idx`.
- ④ The result is written to the pre-allocated array of present values, and we avoid race conditions because the different threads are working on different scenarios.
- ⑤ We don't explicitly have to `return nothing` here, but it makes it extra clear that the intention of the function is to mutate the `present_values` array given to it. This mutation intention is also signaled by the `!` convention in the function name.

`calculate_present_values_kernel!` (generic function with 1 method)

The kernel above was fairly similar to how we might write code for CPU-threaded approaches, but we now need to specify the technicals of launching this on the GPU. The `threads` defines how many independent calculations to run at a given time, and the maximum will be dependent on the hardware used. The `groups` argument defines the number of threads that share memory and synchronize results together (meaning that group will wait for all threads to finish before moving onto the next chunk of data). The push-pull here is that threads that can share data avoid needing to create duplicate copies of that data in memory. However, if there is variability in how long each calculation will take, then the waiting time for synchronizing results may slow the overall computation down.

Our task utilizes shared memory of the cashflows for each thread, so through some experimentation in advance, we find that a relatively large group size of ~512 is optimal.

We bring this all together through the use of the kernel macro `@metal`:

```
threads = 1024
groups = cld(num_scenarios, 512)

@btme @metal threads = $threads groups = $groups
    ↳ calculate_present_values_kernel!(
        $pvs_GPU,
```

```

$cashflows_GPU,
$discount_matrices_GPU
);
16.625 µs (81 allocations: 2.20 KiB)

```

The kernel version is faster than the array-oriented style above, meaning that the GPU kernel version's computation is significantly faster than the CPU version. However, we saw previously that the cost of moving the data to the GPU memory and then back to the CPU memory was the biggest time sink of all - again we'd need to have more scale in the problem to make offloading to the GPU beneficial overall.

Note

The Metal GPUs are able to iterate threads across three different dimensions. In the prior example, we only used one dimension and thus used `thread_position_in_grid_1d()`. If we were distributing the threads across, say, two dimensions then we would use `thread_position_in_grid_2d()`.

How do you determine how many dimensions to use? A good approach is to mimic the data you are trying to parallelize. In the example of calculating a vector of present values across 100k scenarios, that was the primary 'axis of parallelization'. If instead of a one-dimensional set of cashflows (e.g. a single asset with fixed cashflows), we had a two-dimensional set of cashflows (e.g. a portfolio of many assets), then we may find the best balance of code simplicity and performance to iterate across two dimensions of threads (but we are still limited by the same number of total available threads).

Note

The above example would be translated to CUDA by changing a few things:

- The thread indexing would be `idx = (blockIdx().x - 1) * blockDim().x + threadIdx().x` instead of `idx = thread_position_in_grid_1d()`.
- The GPU arrays should be created with `CuArray` instead of `MtlArray`.
- The kernel macro would be `@cuda threads=threads blocks=blocks calculate_present_values_kernel!(...)` instead of `@metal threads=threads groups=groups calculate_present_values_kernel(...)`.
- The kernel needs a bounds check (`if idx <= ...`) because `threads × blocks` will typically overshoot the data size. For example, `cld(100_000, 1024)` gives 98 blocks, launching 100,352 threads for only 100,000 scenarios. Metal handles this differently via `thread_position_in_grid_1d()`.

Here is a complete, standalone example:

```

using CUDA

num_scenarios = 10^5
cashflows = rand(Float32, 100)
discount_matrices = rand(Float32, 100, num_scenarios)
pvs = zeros(Float32, num_scenarios)

cashflows_GPU = CuArray(cashflows)
discount_matrices_GPU = CuArray(discount_matrices)
pvs_GPU = CuArray(pvs)

function calculate_present_values_kernel!(present_values, cashflows,
    discount_matrices)
    idx = (blockIdx().x - 1) * blockDim().x + threadIdx().x

    if idx <= size(discount_matrices, 2)
        pv = 0.0f0
        for t in 1:size(cashflows, 1)
            pv += cashflows[t] * discount_matrices[t, idx]
        end
        present_values[idx] = pv
    end

    return nothing
end

threads = 1024
blocks = cld(num_scenarios, threads)

@cuda threads=threads blocks=blocks calculate_present_values_kernel!(
    pvs_GPU,
    cashflows_GPU,
    discount_matrices_GPU
)

```

11.7. Multi-Processing / Multi-Device

Multiple device, or **multi-device** computer refers to using separate groups of memory/processor combinations to accomplish tasks in parallel. This can be as simple as multiple distinct cores on within a single desktop computer, or many separate computers networked across the internet, or many processors within a high performance cluster or a computing-as-a-service provider like Amazon Web Services or JuliaHub.

Everything discussed previously related to hardware (Chapter 9, Section 11.5, Section 11.6) continues to apply. The additional complexity is attempting to synchronize the computation across multiple sets of the same (homogeneous) or different (heterogeneous) hardware.

As you might imagine, approaches to multi-device computing can vary widely. Julia's approach tries to strike the balance between capability and user-friendliness and uses a primary/worker model wherein one of the processors is the main coordinator while other processors are "workers". If only one processor is started, then the main processor is also a worker processor. This main/worker approach uses a "one-sided" approach to coordination. The main worker utilizes

high level calls and the workers respond, with some of the communication and hand-off handled by Julia transparently from the user's perspective.

A useful mental model is the asynchronous task-based concepts discussed in Section 11.5.1, as the main worker will effectively queue work with the worker processors. Because there may be a delay associated with the computation or the communication between the processors, the worker runs asynchronously.

Description	Task API	Distributed Analogue
Create a new task	Task()	@spawnat
Run task asynchronously	@async	@spawnat
Retrieve task result	fetch	fetch
Wait for task completion	@sync	sync
Communication between tasks	Channel	RemoteChannel

Adapting the trade producer and monitor example from above to run on multiple processors (see Section 11.5.1.1 to review the base model and algorithm), we make a few key changes:

- using `Distributed` loads the `Distributed` standard Julia library, providing the interface for multi-processing across different hardware.
- `addprocs(n)` will add `n` worker processors (the main processor is already counted as one worker). When adding local machine processors, the processors are part of the local machine. This starts new Julia processes (you can see this in the task manager of the machine) which inherit the package environment (i.e. `Project.toml` and environment variables) from the main process; this does not occur automatically if not part of the same local machine.
 - To add processors from other machines, see the Distributed Computing section of the Julia docs.
- `myid()` is the identification number of the given processor that's been spun up.
- We use a `RemoteChannel` instead of a `Channel` to facilitate communication across processors.
- Instead of `@async`, we use `@spawnat n` to create a task for processor number `n` (or `:any` will automatically assign a processor).

```
using Distributed

# Add worker processes if not already added
if nworkers() == 1
    addprocs(4) # Add 4 worker processes
end

@everywhere using Random

@everywhere function trade_producer(channel, i)
    sleep(rand())
    profit = randn()
    put!(channel, profit)
    println("Producer $(myid()): Trade #$i $(round(profit, digits=3))")
end

@everywhere function portfolio_monitor(channel, n)
    total = 0.0
    for _ in 1:n
```

11. Parallelization

```
    profit = take!(channel)
    total += profit
    p = round(profit, digits=3)
    s = round(total, digits=3)
    println("Monitor ${myid()): Received $p, Cumulative: $s")
end
end

function run_distributed_simulation()
    channel = RemoteChannel(() -> Channel{Float64}(32))

    # Start producer and consumer tasks
    @sync begin
        for i in 1:5
            @spawnat :any trade_producer(channel, i)
        end
        @spawnat :any portfolio_monitor(channel, 5)
    end

    # Close the channel and wait for tasks to finish
    close(channel)
    return nothing
end

# Run the simulation
run_distributed_simulation()

From worker 3: Producer 3: Trade #2 0.759
From worker 4: Producer 4: Trade #3 0.173
From worker 3: Monitor 3: Received 0.759, Cumulative: 0.759
From worker 3: Monitor 3: Received 0.173, Cumulative: 0.932
From worker 3: Monitor 3: Received -1.56, Cumulative: -0.628
From worker 2: Producer 2: Trade #1 -1.56
From worker 2: Producer 2: Trade #5 2.057
From worker 3: Monitor 3: Received 2.057, Cumulative: 1.429
From worker 3: Monitor 3: Received -0.104, Cumulative: 1.325
From worker 5: Producer 5: Trade #4 -0.104
```

Given the similarity to the single-process task-based version above, what's the motivation for bothering with a distributed approach? A few differences:

- In this simplified example, we are simply starting additional Julia processes on the same machine. Like with a threaded approach, the work will be split across the same multi-core processor. In this context, the main difference is that the processes do not share memory.
 - Communicating across processes generally has a little bit more overhead than communicating across threads.
- If distributing across machines, avoiding memory sharing is advantageous if using different machines that have their own memory stores, which need not compete with the main process (such as distributed chunks of large datasets). This essentially helps with memory constrained problems since you are no longer limited by the memory size or throughput of a single machine.
- The worker processors don't need to be the same architecture as the main processor, allowing use of different machines or cloud computing that is communicating with a local main process.

11.8. Parallel Programming Models

The previous sections have explained the different parallel programming models and how to directly utilize them to harness additional computing power. Each approach (multi-threading, GPU, distributed processing, etc.) has unique considerations and trade-offs. These approaches in Julia are generally much more accessible to beginning and intermediate users than other languages, but admittedly still require a decent amount of thought and care.

It is possible, if you are willing to give up some fine-grained control, to utilize some higher level approaches which look to abstract away some of the particularities of the implementation.

11.8.1. Map-Reduce

Map-Reduce (Section 6.4.4) operations are inherently parallelizable and various libraries provide parallelized versions of the base `mapreduce`. This is the workhorse function of many ‘big data’ workloads, and many statistical operations are versions of `mapreduce`.

11.8.1.1. Multi-Threading

11.8.1.1.1. OhMyThreads

`OhMyThreads.jl` provides the threaded versions of essential functions such as `tmap`, `tmapreduce`, `tcollect`, and `tforeach` (see Table 6.1). In most cases, the chunking and data sharing is handled automatically for you.

```
import OhMyThreads
@btime OhMyThreads.tmapreduce(x -> x, +, 1:100_000)

10.791 μs (75 allocations: 4.20 KiB)

5000050000
```

11.8.1.1.2. ThreadsX

`ThreadsX.jl` is built off of the wonderful `Transducers.jl` package, though the latter is a bit more advanced (more abstract, but as a result more composable and powerful). `ThreadsX` provides threaded versions of many popular base functions. It offers a wider set of ready-made threaded functions, but has a much more complex codebase. For the vast majority of threading needs, `OhMyThreads.jl` should be sufficient and performant. See the documentation for all of the implemented functions, but for our illustrative example:

```
import ThreadsX
@btime ThreadsX.mapreduce(x -> x, +, 1:100_000)

39.917 μs (890 allocations: 47.64 KiB)

5000050000
```

11. Parallelization

11.8.1.2. Multi-Processing

`reduce(op,pmap(f,collection))` will use a distributed map and reduce the resulting map on the main thread. This pattern works well if each application of `f` to elements of `collection` is costly.

`@distributed (op) for x in collection; f(x); end` is a way to write the loop with the reduction `op` for which the `f` need not be costly.

The difference between the two approaches is that, with `pmap`, `collection` is made available to all workers. In the `@distributed` approach, the collection is partitioned and only a subset is sent to the designated workers.

Here's an example of both of these, calculating a simple example of counting coin flips:

```
# this is an example of poor utilization of pmap, since the operation is
# fast and the overhead of moving the whole collection dominates
@btime reduce(+, pmap(x -> rand((0, 1)), 1:10^3))
```

73.503 ms (71605 allocations: 2.87 MiB)

526

```
function dist_demo()
    @distributed (+) for _ in 1:10^5
        rand((0, 1))
    end
end

@btime dist_demo()
```

189.792 μs (335 allocations: 14.52 KiB)

49931

11.8.2. Array-Based

Array-based approaches will often utilize the parallelism of SIMD on the CPU or many cores on the GPU. It's as simple as using generic library calls which will often be optimized at the compiler level. Examples:

```
let
    x = rand(Float32, 10^8)
    x_GPU = MtlArray(x)
    @btime sum($x)
    @btime sum($x_GPU)
end

4.976 ms (0 allocations: 0 bytes)
1.287 ms (513 allocations: 15.44 KiB)
```

5.0002452f7

`sum(x)` compiles to SIMD instructions on the CPU, while using the GPU array type in `sum(x_GPU)` is enough to let the compiler dispatch on the GPU type and emit efficient, parallelized code for the GPU.

Distributed array types allow for large datasets to effectively be partitioned across multiple processors, and have implementations in the `DistributedArrays.jl` and `Daggerjl` libraries.

11.8.3. Loop-Based

Loops that don't have race conditions can easily become multi-threaded. Here, we have three versions of updating a collection to square the contained values:

Basic single-threaded:

```
let v = collect(1:10000)

for i in eachindex(v)
    v[i] = v[i]^2
end
v[1:3]
end
```

```
3-element Vector{Int64}:
1
4
9
```

Using multi-threading:

```
let v = collect(1:10000)
Threads.@threads for i in eachindex(v)
    v[i] = v[i]^2
end
v[1:3]
end
```

```
3-element Vector{Int64}:
1
4
9
```

Using multi-processing:

```
using SharedArrays
let
    v = collect(1:10_000)
    sV = SharedArray{eltype(v)}(length(v))
    @sync Distributed.@distributed for i in eachindex(v)
        sV[i] = v[i]^2
    end
    sV[1:3]
end
```

11. Parallelization

```
3-element Vector{Int64}:
1
4
9
```

For more advanced usage, including handling shared memory see Section 11.7 and Section 11.5.

11.8.4. Task-Based

Task-based approaches attempt to abstract the scheduling and distribution of work from the user. Instead of saying how the computation should be done, the user specifies the intended operations and allows the library to handle the workflow. The main library for this in Julia is Dagger.jl.

Effectively, the library establishes a network topology (a map of how different processor nodes can communicate) and models the work as a directed, acyclic graph (a DAG, which is like a map of how the work is related). The library is then able to assign the work in the appropriate order to the available computation devices. The benefit of this is most apparent with complex workflows or network topologies where it would be difficult to manually assign, communicate, and schedule the workflow.

Here's a very simple example which demonstrates Dagger waiting for the two tasks which work in parallel (we already added multiple processors to this environment in Section 11.7):

```
import Dagger

# This runs first:
a = Dagger.@spawn rand(100, 100)

# These run in parallel:
b = Dagger.@spawn sum(a)
c = Dagger.@spawn prod(a)

# Finally, this runs:
d = Dagger.@spawn println("b: ", b, ", c: ", c)
wait(d)
```

11.9. Choosing a Parallelization Strategy

There is no one-size-fits-all strategy to parallelization. Here are some general guides to thinking about what parallelization technique to try given the circumstances:

- **CPU-bound workloads with manageable memory demands:** If your entire dataset fits comfortably in RAM and your operations are primarily arithmetic or straightforward loops, start by optimizing your single-threaded performance and consider **vectorization (SIMD)** for inner loops and **multi-threading** for parallelizable tasks. This approach leverages your CPU cores efficiently without introducing significant complexity.
- **Large-scale linear algebra or highly data-parallel computations:** If your problem involves large matrix operations, linear algebra routines, or embarrassingly parallel computations that can be batched over many independent elements, a GPU or other specialized accelerators may be beneficial. GPUs excel at uniform computations over large datasets and can provide substantial speedups, assuming data-transfer overhead and memory constraints are managed effectively. Note that standard linear algebra libraries are likely to already parallelize on the CPU without any explicit parallel code on your part.

- **Distributing work across multiple machines or heterogeneous resources:** If you need to scale beyond a single machine's CPU and GPU capabilities, whether due to extremely large datasets, the need for concurrent access to geographically distributed resources, or leveraging specialized hardware, then consider **distributed computing**. Spreading tasks across multiple processes, servers, or clusters can scale performance horizontally. Just keep in mind the overhead of communication, potential data-partitioning strategies, and the complexity of managing a distributed environment.

In practice, you may find that a combination of these approaches is ideal: start simple, measure performance, and iterate. By understanding your workload and hardware constraints, you can make informed decisions that balance complexity, cost, and the performance gains of parallel computing.

11.10. References

The following resources provide additional depth on parallel computing concepts and Julia-specific implementations:

- Rackauckas (2020b) provides an excellent overview of parallelism types in the context of scientific computing
- Julia Documentation (2024) is the official Julia documentation on parallel computing constructs
- ENCCS (2023) offers hands-on tutorials for high-performance computing with Julia

Part V.

Interdisciplinary Concepts and Applications

This section explores concepts from related fields that enhance financial modeling—ideas from computer science, statistics, and other disciplines that intersect with and improve modeling practices. Through examples, we uncover the theoretical foundations supporting advanced techniques.

This interdisciplinary approach aims to broaden your perspective and equip you with diverse tools for tackling complex financial problems.

12. Applying Software Engineering Practices

CHAPTER AUTHORED BY: ALEC LOUDENBACK

*"Programs must be written for people to read, and only incidentally for machines to execute." — Harold Abelson and Gerald Jay Sussman (*Structure and Interpretation of Computer Programs*, 1984)*

12.1. Chapter Overview

Modern software engineering practices—version control, testing, documentation, and automated pipelines—make modeling more robust and reproducible. This chapter also covers data practices and workflow guidance.

12.2. Introduction

In addition to the core concepts from computer science described so far, there are also neat ideas about the *practice* and *experience* of working with a code-based workflow that makes the end-to-end approach more powerful than the code itself.

It's likely that the majority of a professional financial modeler's time is often spent *doing things other than building models*, such as testing the model's results, writing documentation, collaborating with others on the design, and figuring out how to share the model with others inside the company. This chapter covers how a code-first workflow makes each one of those responsibilities easier or more effective.

12.2.1. Regulatory Compliance and Software Practices

Financial models often face regulatory requirements around model validation, change management, and reproducibility. Software engineering practices directly support these requirements - version control provides a complete audit trail of all model changes, automated testing helps validate model behavior and demonstrates ongoing quality control, and comprehensive documentation meets regulatory demands for model transparency. For example, the European Central Bank's Targeted Review of Internal Models (TRIM) Guide explicitly requires financial institutions to maintain documentation of model changes and validation procedures, which is naturally supported by Git commit history and continuous integration test reports that will be discussed in this chapter.

12.2.2. Chapter Structure

There are three essential topics covered in this chapter:

- **Testing** is the practice of implementing automated checks for desired behavior and outputs in the system.
- **Documentation** is the practice of writing plain English (or your local language) to complement the computer code for better human understanding.
- **Version Control** is the systematic practice of tracking changes and facilitating collaborative workflows on projects.

As the chapter progresses, some highly related topics are covered, bridging some of the conceptual ideas into practical implementations for your own code and models.

As a reminder, this chapter is heavily oriented to concepts that are applicable in any language, though the examples are illustrated using Julia for consistency and its clarity. The code examples would have direct analogues in other languages. In Chapter 21 many of these concepts will be reinforced and expanded upon with Julia-specific workflows and tips.

12.3. Testing

Testing is a crucial aspect of software engineering that ensures the reliability and correctness of code. In financial modeling, where accuracy is paramount, implementing robust testing practices is essential, and in many cases now often required by regulators or financial reporting authorities. It's good practice regardless of requirements.

A test is implemented by writing a boolean expression after a `@test` macro:

```
@test model_output == desired_output
```

If the expression evaluates to `true`, then the test passes. If the expression is anything else (`false`, or produces an error, or nothing, etc.) then the test will fail.

Here is an example of modeled behavior being tested. We have a function which will discount the given cashflows at a given annual effective interest rate. The cashflows are assumed to be equally spaced at the end of each period:

```
function present_value(discount_rate, cashflows)
    v = 1.0
    pv = 0.0
    for cf in cashflows
        v = v / (1 + discount_rate)
        pv = pv + v * cf
    end
    return pv
end

present_value (generic function with 1 method)
```

We might test the implementation like so:

```
using Test
```

```
@test present_value(0.05, [10]) ≈ 10 / 1.05
@test present_value(0.05, [10, 20]) ≈ 10 / 1.05 + 20 / 1.05^2
```

Test Passed

The above test passes because the expression is true. However, the following will fail because we have defined the function assuming the given `discount_rate` is compounded once per period. This test will fail because the test target presumes a continuous compounding convention. A failing `@test` reports a test failure (showing actual vs expected). A stacktrace appears only if an error is thrown.

```
@test present_value(0.05, [10]) ≈ 10 * exp(-0.05 * 1)
```

💡 Tip

When testing results of floating point math, it's a good idea to use the approximate comparison (`≈`, typed in a Julia editor by entering `\approx`) or the `\approx` function. Recall that floating point math is a discrete, computer representation of continuous real numbers. As perfect precision is not efficient, very small differences can arise depending on the specific numbers involved or the order in which the operations are applied.

In tests (as in the `isapprox`/`≈` function), you can also further specify a relative tolerance and an absolute tolerance:

```
@test 1.02 ≈ 1.025 atol = 0.01
@test 1.02 ≈ 1.025 rtol = 0.005
```

Test Passed

The testing described in this section is sort of a ‘sampling’ based approach, wherein the modeler decides on some pre-determined set of outputs to test and determines the desired outcomes for that chosen set of inputs. That is, testing that `2 + 2 == 4` versus testing that a positive number plus a positive number will always equal another positive number. There are some more advanced techniques that cover the latter approach in Section 13.6.1.

💡 Tip

More Julia-specific testing workflows are covered in Section 21.14.

12.3.1. Test Driven Development

Test Driven Development (TDD) is a software development approach where tests are written before the actual code. The process typically follows these steps:

1. Write a test that defines a desired function or improvement.
2. Run the test, which should fail since the feature hasn't been implemented.
3. Write the minimum amount of code necessary to pass the test.
4. Run the test again. If it passes, the code meets the test criteria.
5. Refactor the code for optimization while ensuring the test still passes.

TDD can be particularly useful in financial modeling as it helps clarify (1) intended behavior of the system and (2) how you think the system should work.

12. Applying Software Engineering Practices

For example, if we want to create a new function which calculates an interpolated value between two numbers, we might first define the test like this:

```
# interpolate between two points (0,5) and (2,10)
@test interp_linear(0,2,5,10,1) ≈ (10-5)/(2-0) * (1-0) + 5
```

We've defined how it should work for a value inside the bounding x values, but writing the test has us wondering... should the function error if x is outside of the left and right x bounds? Or should the function extrapolate outside the observed interval? The answer to that depends on exactly how we want our system to work. Sometimes the point of such a scheme is to extrapolate, other times extrapolating beyond known values can be dangerous. For now, let's say we would like to have the function extrapolate, so we can define our test targets to work like that:

```
@test interp_linear(0,2,5,10,-1) ≈ (10-5)/(2-0) * (-1 - 0) + 5
@test interp_linear(0,2,5,10,3) ≈ (10-5)/(2-0) * (3 - 0) + 5
```

By thinking through what the correct result for those different functions is, we have forced ourselves to think about how the function should work generically:

```
function interp_linear(x1,x2,y1,y2,x)
    # slope times difference from x1 + y1
    return (y2 - y1) / (x2 - x1) * (x - x1) + y1
end
```

interp_linear (generic function with 1 method)

And we can see that our tests defined above would pass after writing the function.

```
@testset "Linear Interpolation" begin
    @test interp_linear(0,2,5,10,1) ≈ (10-5)/(2-0) * (1-0) + 5
    @test interp_linear(0,2,5,10,-1) ≈ (10-5)/(2-0) * (-1 - 0) + 5
    @test interp_linear(0,2,5,10,3) ≈ (10-5)/(2-0) * (3 - 0) + 5
end;
```

	Pass	Total	Time
Linear Interpolation	3	3	0.0s

12.3.2. Test Coverage

Testing is great, but what if some things aren't tested? For example, we might have a function that has a branching `if/else` condition and only ever tests one branch. Then when the other branch is encountered in practice it is more vulnerable to having bugs because its behavior was never double checked. Wouldn't it be great to tell whether or not we have tested all of our code?

The good news is that there is! **Test coverage** is a measurement related to how much of the codebase is covered by at least one associated test case. In the following example, code coverage would flag that the ... other logic is not covered by tests and therefore encourage the developer to write tests covering that case:

```
function asset_value(strike,current_price)
    if current_price > strike
        # ... some logic
    else
        # ... other logic
    end
```

```
end

@test asset_value(10,11) ≈ 1.5
```

From the coverage statistics, it's possible to determine a score for how well a given set of code is tested. 100% coverage means every line of code has at least one test that double checked its behavior.

Warning

Testing is only as good as the tests that are written. You could have 100% code coverage for a codebase with only a single rudimentary test covering each line. Or the test itself could be wrong! Testing is not a cure-all, but does encourage best practices.

Test coverage is also a great addition when making modification to code. It can be set up such that you receive reports on how the test coverage changes if you were to make a certain modification to a codebase. An example might look like this for a proposed change which added 13 lines of code, of which only 11 of those lines were tested ("hit"). The total coverage percent has therefore gone down (-0.49%) because the proportion of new lines covered by tests is $11/13 = 84\%$, which is lower than the original coverage rate of 90%.

@@	Coverage	Diff	@@
##	original	modif	+/- ##
- Coverage	90.00%	89.51%	-0.49%
Files	2	2	
Lines	130	143	+13
+ Hits	117	128	+11
- Misses	13	15	+2

12.3.3. Types of Tests

Different tests can emphasize different aspects of model behavior. You could be testing a small bit of logic, or test that the whole model runs if hooked up to a database. The variety of this kind of testing has given rise to various named types of testing, but it's somewhat arbitrary and the boundaries between the types can be fuzzy. Common types include unit testing, integration testing, and regression testing.

Test Type	Description
Unit Testing	Verifies the functionality of individual components or functions in isolation. It ensures that each unit of code works as expected.
Integration Testing	Checks if different modules or services work together correctly. It verifies the interaction between various components of the system.
End-to-End Testing	Simulates real user scenarios to test the entire application flow from start to finish. It ensures the system works as a whole.
Functional Testing	Validates that the software meets specified functional requirements and behaves as expected from a user's perspective.
Regression Testing	Ensures that new changes or updates to the code haven't broken existing functionality. It involves re-running previously completed tests.

12. Applying Software Engineering Practices

There are other types of testing that can be performed on a model, such as performance testing, security testing, acceptance testing, etc., but these types of tests are outside of the scope of what we would evaluate with an @test check. It is possible to create more advanced, mathematical-type checks and tests, which are introduced in Section 13.6.1.

💡 Financial Modeling Pro Tip

Test reports and test coverage are a wonderful way to demonstrate regular and robust testing for compliance. It is important to read and understand any limitations related to testing in your choice of language and associated libraries.

12.4. Documentation

The most important part of code for maintenance purposes is plainly written notes for humans, not the compiler. This includes in-line comments, docstrings, reference materials, and how-to pages, etc. Even as a single model developer, writing comments for your future self is critical for model maintenance and ongoing productivity.

12.4.1. Comments

Comments are meant for the developer to aid in understanding a certain bit of code. A bit of time-tested wisdom is that after several weeks, months, or years away from a piece of code, something that seemed ‘obvious’ at the time tends to become perplexing at a later time. Writing comments is as much for yourself as it is your colleagues or successors. Focus comments on **why** something is done (business constraints, regulatory rules, data quirks) rather than restating the next line of code.

Here’s an example of documentation with single-line comments (indicated with the preceding #) and multi-line comments (enclosed by #= and =#):

```
function calculate_bond_price(face_value, coupon_rate, years_to_maturity,
    ↵ market_rate)
    # Convert annual rates to semi-annual
    semi_annual_coupon = (coupon_rate / 2) * face_value
    semi_annual_market_rate = market_rate / 2
    periods = years_to_maturity * 2

    # Calculate the present value of coupon payments
    pv_coupons = 0
    for t in 1:periods
        pv_coupons += semi_annual_coupon / (1 + semi_annual_market_rate)^t
    end

    # Calculate the present value of the face value
    pv_face_value = face_value / (1 + semi_annual_market_rate)^periods

    #=
    Sum up the components for total bond price
    1. Present value of all coupon payments
    2. Present value of the face value at maturity
    =#
```

```

    bond_price = pv_coupons + pv_face_value

    return bond_price
end

```

12.4.2. Docstrings

Docstrings (documentation strings) are intended to be a user-facing reference and help text. In Julia, docstrings are just strings placed in front of definitions. Markdown¹ is available (and encouraged) to add formatting within the docstring.

Here's an example with some various features of documentation shown:

```

"""
    calculate_bond_price( # <1>
        face_value,
        coupon_rate,
        years_to_maturity,
        market_rate
    )

Calculate the price of a bond using discounted cash flow method.

Parameters:
- `face_value`: The bond's par value
- `coupon_rate`: Annual coupon rate as a decimal
- `years_to_maturity`: Number of years until the bond matures
- `market_rate`: Current market interest rate as a decimal

Returns:
- The calculated bond price

## Examples: # <2>

```julia-repl
julia> calculate_bond_price(1000, 0.05, 10, 0.06)
925.6126256977221
```

## Extended help: # <3>

This function uses the following steps to calculate the bond price:
1. Convert annual rates to semi-annual rates
2. Calculate the present value of all future coupon payments
3. Calculate the present value of the face value at maturity
4. Sum these components to get the total bond price

The calculation assumes semi-annual coupon payments, which is standard in
↳ many markets.
"""

```

¹Markdown is a type of plain text which can be styled for better communication and aesthetics. E.g. ****some text**** would render as boldface: **some text**. This book was written in Markdown and all of the styling arose as a result of plain text files written with certain key elements.

12. Applying Software Engineering Practices

```
function calculate_bond_price(face_value, coupon_rate, years_to_maturity,
    ↵ market_rate)
    # ... function definition as above
end
```

- ① The typical docstring on a method includes the signature, indented so it's treated like code in the Markdown docstring.
- ② It's good practice to include a section that includes examples of appropriate usage of a function and the expected outcomes.
- ③ The *Extended help* section is a place to put additional detail that's available on generated docsites and in help tools like the REPL help mode.

The last point, the *Extended help* section is shown when using help mode in the REPL and including an extra ?. For example, in the REPL, typing ?calculate_bond_price will show the docstring up through the examples. Typing ??calculate_bond_price will show the docstring in its entirety.

Docstrings become even more valuable when you hook them into automated documentation generators such as Documenter.jl. Documenter ingests Markdown pages plus in-code docstrings, renders math, runs doctests, and publishes a searchable site every time CI runs. Writing rich docstrings up front means your docsite, help mode, and notebook snippets stay perfectly in sync.

12.4.3. Docsites

Docsites, or documentation sites, are websites that are generated to host documentation related to a project. Modern tooling associated with programming projects can generate a really rich set of interactive documentation while the developer/modeler focuses on simple documentation artifacts.

Specifically, modern docsite tooling generally takes in markdown text pages along with the in-code docstrings and generates a multi-page site that has navigation and search.

Typical contents of a docsite include:

- A *Quickstart* guide that introduces the project and provides an essential or common use case that the user can immediately run in their own environment. This conveys the scope and demonstrates intended usage.
- *Tutorials* are typically worked examples that introduce basic aspects of a concept or package usage and work up to more complex use cases.
- *Developer documentation* is intended to be read by those who are interested in understanding, contributing, or modifying the codebase.
- *Reference documentation* describes concepts and available functionality. A subset of this is API, or **Application Programming Interface**, documentation which is the detailed specification of the available functionality of a package, often consisting largely of docstrings like the previous example.

Docsite generators are generally able to look at the codebase and from just the docstrings create a searchable, hierarchical page with all of the content from the docstrings in a project. This basic docsite feature is incredibly beneficial for current and potential users of a project.

To go beyond creating a searchable index of docstrings requires additional effort (time well invested!). Creating the other types of documentation (quick start, tutorials, etc.) is mechanically as simple as creating a new markdown file. The hard part is learning how to write quality documentation. Good technical writing is a skill developed over time - but at least the technical and workflow aspects have been made as easy as possible!

12.5. Version Control

Version control systems (VCS) refer to the tracking of changes to a codebase in a verifiable and coordinated way across project contributors. VCS underpins many aspects of automating the mundane parts of a modeler's job. Benefits of VCS include (either directly, or contribute significantly to):

- Access control and approval processes
- Versioning of releases
- Reproducibility across space and time of a model's logic
- Continuous testing and validation of results
- Minimization of manual overrides, intervention, and opportunity for user error
- Coordinating collaboration in parallel and in sequence between one or many contributors

These features are massively beneficial to a financial modeler! A lot of the overhead in a modeler's job becomes much easier or automated through this tooling.

Among several competing options (CVS, Mercurial, Subversion, etc.), Git is the predominant choice as of this book's writing and therefore we will focus on Git concepts and workflows.

12.5.1. Git Overview

This section will introduce Git related concepts at a level deeper than "here's the five commands to run" but not at a reference-level of detail. The point of this is to reveal some of the underlying technology and approaches so that you can recognize where similar ideas appear in other areas and understand some of the limitations of the technology.

Git is free and open-source software that tracks changes in files using a series of snapshots. Git itself is the underlying software tool and is a command-line tool at its core. However, you will often interact with it through various interfaces (such as a graphical user interface in VS Code, GitKraken, or other tool).

Each snapshot, or **commit**, stores references to the files that have changed². All of this is stored in a `.git` / subfolder of a project, which is automatically created upon initializing a repository. This folder may be hidden by default by your filesystem. You generally never need to modify anything inside the `.git` / folder yourself as Git handles this for you. Git tracks files in the repository working tree (the directory containing the `.git` folder and its subdirectories). It does not track directories outside the repository.

Note

Think of Git as a series of 'save points' in a video game. Each time you reach a milestone, you create a 'commit' - a snapshot of your entire project. If you make a mistake later, you can always reload a previous save point. 'Branches' are like alternate timelines, allowing you to experiment with new features without affecting your main saved game.

Hashes indicate a verifiable snapshot of a codebase. For example, a full commit ID (a hash) `40f141303cec3d58879c493988b71c4e56d79b90` will *always* refer to a certain snapshot of the code, and if there is a mismatch in the git history between the contents of the repository and the commit's hash, then the git repository is corrupted and it will not function. A corrupted repository usually doesn't happen in practice, of course! You might see hashes shortened to the first several characters

²Git uses a content-addressable filesystem, meaning it stores data as key-value pairs. The key is a hash of the content, ensuring data integrity and allowing efficient storage of identical content. For more on hashes, see the Content Hashes callout later in this chapter.

12. Applying Software Engineering Practices

(e.g. 40f1413) and in the following examples we'll shorten the hypothetical hashes to just three characters (e.g. 40f).

A codebase can be **branched**, meaning that two different versions of the same codebase can be tracked and switched between. Git lends itself to many different workflows, but a common one is to designate a primary branch (call it `main`) and make modifications in a new, separate branch. This allows for non-linear and controlled changes to occur without potentially tainting the `main` branch. It's common practice to always have the `main` branch be a 'working' version of the code that can always 'run' for users, while branches contain works-in-progress or piecemeal changes that temporarily make the project unable to run.

The commit history forms a directed acyclic graph (DAG) representing the project's history. That is, there is an order to the history: the second commit is dependent on the first commit. From the second commit, two child commits may be created that both have the second commit as their parent. Each one of these commits represents a stored snapshot of the project at the time of the commit.

Note

A directory structure demonstrating where git data is stored and what content is tracked after initializing a repository in the `tracked-project` directory. Note how `untracked-folder` is not the parent directory of the `.git` directory so it does not get tracked by Git.

```
/home/username
└── untracked-folder
    ├── random-file.txt
    └── ...
└── tracked-project
    ├── .git
    │   ├── config
    │   ├── HEAD
    │   └── objects
    │       └── ...
    ├── .gitignore
    ├── README.md
    └── src
        ├── MainProject.jl
        ├── module1.jl
        └── module2.jl
```

Under the hood, the Git data stored inside the `.git/` includes things such as binary blobs, trees, commits, and tags. Blobs store file contents, trees represent directories, commits point to trees and parent commits, and tags provide human-readable names for specific commits. This structure allows Git to efficiently handle branching, merging, and maintaining project history.

Table 12.2 shows an example workflow for trying to fix an erroneous function `present_value` which was written as part of a hypothetical `FileXYZ.jl` file. This example is trivial, but in a larger project where a 'fix' or 'enhancement' may span many files and take several days or weeks to implement, this type of workflow becomes like a superpower compared to traditional, manual version control approaches. It sure beats an approach where you end up with filenames like `FileXYZ v23 July 2022 Final.jl!`

Table 12.2.: A workflow demonstrating branching, staging, committing, and merging in order to fix an incorrect function definition for a present value (`pv`) function. The branch name/commit ID shows which version you would see on your own computer/filesystem. The inactive branches are tracked in Git but do not manifest themselves on your filesystem unless you checkout that branch.

| Branch main, <code>FileXYZ.jl</code> file | Branch <code>fix_function</code> , <code>FileXYZ.jl</code> file | Action | Active Branch
(Abbreviated Commit ID) |
|--|---|--|--|
| | | Does not yet exist | |
| <code>function pv(rate,amount,time)</code> | | | |
| <code>amount / (1+rate)</code> | | | |
| <code>end</code> | | | |
| " | | | |
| | <code>function pv(rate,amount,time)</code> | Write original function which forgets to take into account the <code>time</code> . Stage and commit it to the main branch. | |
| | <code>amount / (1+rate)</code> | | |
| | <code>end</code> | | |
| | " | | |
| | | Git command:
<code>git add FileXYZ.jl</code> , then <code>git commit -m 'add pv function'</code> | |
| | | Create a new branch <code>fix_function</code> . | |
| | | Git command:
<code>git branch fix_function</code> | |
| | | Checkout the new branch and make it active for editing. The branch is different but is starting from the existing commit. | |
| | | Git command: <code>git checkout fix_function</code> | |
| | | Edit and save the changed file. No git actions taken. | |
| | " | | |
| | <code>function pv(rate,amount,time)</code> | <code>fix_function</code> | |
| | <code>amount / (1+rate)^(t-time)</code> | | |
| | <code>end</code> | | |
| | " | | |
| | | "Stage" the modified file, telling git that you are ready to record ("commit") a new snapshot of the project. | |
| | | Git command: <code>git add FileXYZ.jl</code> | |
| | | Commit a new snapshot of the project by committing with a note to collaborators saying fix: present value logic | |
| | | Git command: <code>git commit -m 'fix: present value logic'</code> | |
| | | Switch back to the primary branch | |
| | | Git command: <code>git checkout main</code> | |
| | | Merge changes from other branch into the <code>main</code> branch, incorporating the corrected version of the code. | |
| | | Git command: <code>git merge fix_function</code> | |
| | " | | |
| | <code>function pv(rate,amount,time)</code> | <code>main</code> | |
| | <code>amount / (1+rate)^(t-time)</code> | | |
| | <code>end</code> | | |

12. Applying Software Engineering Practices

A visual representation of the git repository and commits for the actions described in Table 12.2 might be as follows, where the ...XXX is the shortened version of the hash associated with that commit.

```
main branch      : ...58b      → ...b90
                  ↓          ↗
fix_function branch : ...58b → ...6ac
```

The “staging” aspect will be explained next.

12.5.2. Change governance checklist

Most teams wrap the branch workflow inside a pull-request (PR) or merge-request process. A typical cadence for an actuarial change is:

1. Open a PR from your feature branch with a concise summary (`fix: correct PV logic for monthly products`) and a link to the model-change ticket or associated GitHub Issue number.
2. Request a peer review so another actuary/developer confirms the implementation, assumptions, and data lineage.
3. Once tests are green and approvals recorded, merge the branch; the PR becomes part of the model-change audit trail.

Writing clear commit messages (e.g., Conventional Commits) and using PR templates that ask for “risk assessment” or “validation evidence” makes it easy to satisfy SOX/CECL/Solvency documentation requirements without a separate Word document.

12.5.2.1. Git Staging

Staging (sometimes called the “staging area” or “index”) is an intermediate step between making changes to files and recording those changes in a commit. Think of staging as preparing a snapshot - you’re choosing which modified files (or even which specific changes within files) should be included in your next commit. When you modify files in your Git repository, those changes are initially “unstaged.” Using the `git add` command (or your Git GUI’s staging interface) moves changes into the staging area. This two-step process - staging followed by committing - gives you precise control over which changes get recorded together. Here’s a typical workflow:

1. You modify several files in your modeling project
2. You review the changes and decide which ones are ready to commit
3. You stage only the changes that belong together as a logical unit
4. You create a commit with just those staged changes
5. Repeat as needed with other modified files

This is particularly useful when you’re working on multiple features or fixes simultaneously. For example, imagine you’re updating a financial model and you:

- Fix a bug in your present value calculation
- Add comments to improve documentation
- Start working on a new feature

You could stage and commit the bug fix and documentation separately, keeping the work-in-progress feature changes unstaged until they’re complete. This creates a cleaner, more logical project history where each commit represents one coherent change.

Think of staging as preparing a shipment: you first gather and organize the items (staging), then seal and send the package (committing). This extra step helps maintain a well-organized project history where each commit represents a logical, self-contained change.

12.5.2.2. Git Tooling

Git is traditionally a command-line based tool, however we will not focus on the command line usage as more beginner friendly and intuitive interfaces are available from different available software. The branching and nodes are well suited to be represented visually.

i Note

Some recommend Git tools with a **graphical user interface** (GUI):

- *Github Desktop* interfaces nicely with Github and provides a GUI for common operations.
- *Visual Studio Code* has an integrated Git pane, providing a powerful GUI for common operations.
- *GitKraken* is free for open source repositories but requires payment for usage in enterprise or private repository environments. GitKraken provides intuitive interfaces for handling more advanced operations, conflicts, or issues that might arise when using Git.

12.5.3. Collaborative Workflows

Git is a distributed VCS, meaning that copies of the repository and all its history and content can live in multiple locations, such as on two colleagues' computer as well as a server. In this distributed model, what happens if you make a change locally?

Git maintains a local repository on each user's machine, containing the full project history. This local repository includes the working directory (current state of files), staging area (changes prepared for commit), and the .git directory (metadata and object database). When collaborating, users push and pull changes to and from remote repositories. Git uses a branching model, allowing multiple lines of development to coexist and merge when ready.

i Note

The primary branch of a project is typically named the main branch. This change reflects a shift from the older master branch name, which some projects may still use.

12.5.3.1. Pull Requests

Layered onto core Git functionality, services like Github provide interfaces which enhance the utility of VCS. A major example of this is **Pull Requests** (or PRs), which are a process of merging Git branches in a way that allows for additional automation and governance.

The following is an example of a PR on a repository adding a small bit of documentation to help future users. We'll walk through several elements to describe what's going on:

Referencing Figure 12.1, several elements are worth highlighting. In this pull request the author of this change, alecloudenback, is proposing to modify the QuartoNotebookRunner repository,

12. Applying Software Engineering Practices

Note that Quarto doesn't follow project-wide engine yet #157

Merged MichaelHatherly merged 1 commit into `PumasAI:main` from `alecloudenback:patch-1`  on Jun 20

Conversation 2 Commits 1 Checks 9 Files changed 1 +1 -1

alecloudenback commented on Jun 19 · edited  ...
Let users know how engine needs to be configured.

alecloudenback mentioned this pull request on Jun 19
Engine not able to be set project-wide #156 

MichaelHatherly approved these changes on Jun 19  View reviewed changes

MichaelHatherly left a comment
Thanks.

codecov bot commented on Jun 20
Codecov Report
All modified and coverable lines are covered by tests 
Thoughts on this report? [Let us know!](#)

MichaelHatherly merged commit `30c61f8` into `PumasAI:main` on Jun 20

9 checks passed

| | |
|---|-------------------------|
|  test (1.6, ubuntu-latest) | Details |
|  test (1.10.0, ubuntu-latest) | Details |
|  test (1.10.0, macos-13) | Details |
|  test (1.10.0, windows-latest) | Details |
|  test (1.7, macos-13) | Details |
|  test (1.7, windows-latest) | Details |
|  format | Details |
|  finalize | Details |

Figure 12.1.: A Pull Request on Github, demonstrating several utilities which enhance change management, automation, and governance.

which is not a repository for which the user alecloudenback has any direct rights to modify. After having made a copy of the repository (a “fork”), creating a new branch, making modifications, and committing those changes... a pull request has been made to modify the primary repository.

- All changes are recorded using Git, keeping track of authorship, timestamps, and history.
- At the top of the screenshot, the title “Note that Quarto...” allows the author to summarize what is changed in the branch to be merged.
- The “Conversation” tab allows for additional details about the change to be discussed.
- In the top right, a **+1 -1**  is an indication of what’s changed. In this case, a single line of code (documentation, really) was removed (-1) and replaced with another (+1)
- Not shown in the Figure 12.1, the “Files Changed” tab shows a file-by-file and line-by-line comparison of what has changed (see Figure 12.2).
- The reviewer (user MichaelHatherly) is a collaborator with rights to modify the destination repository and has been assigned to review this change before merging it into the **main** branch.
- “CodeCoverage” was discussed above in testing, and in this case tests were automatically run when the PR was created, and the coverage indicates that there was no added bit of code that was untested.
- A section of “9 checks” that pass, which validated that tests within the repository still passed, on different combinations of operating systems and Julia versions. Additionally, the repository was checked to ensure that formatting conventions were followed.
- Additionally, there are a number of project management features not really showcased here. A few to note:
 - The cross-referencing to another related issue in a different repository (the reference to issue #156).
 - Assignees (assigned doers), labels, milestones, and projects are all functionality to help keep track of codebase development.

The features described here can take modelers many hours of time - testing, review of changes, sign-off tracking, etc. In this Git-based workflow, the workflow above happens frictionlessly and much of it is automated. This is such a powerful paradigm that should be adopted within the financial industry and especially amongst modelers.

12.5.3.2. Continuous integration (CI)

CI pipelines turn “remember to rerun the model” into code. A minimal Julia CI workflow usually runs:

1. `julia --project -e 'using Pkg; Pkg.instantiate(); Pkg.test()'`
2. Optional data validation scripts (e.g., confirm new market data falls within sanity bands).
3. Documentation builds (`docs/make.jl`) and style checks (using `JuliaFormatter; format(".")`).

Because GitHub Actions, GitLab CI, and Azure Pipelines capture logs and artifacts automatically, each pull request carries a timestamped record of which Julia version, OS, and dependency set produced the numbers. Regulators can accept these logs as part of model change evidence because they are immutable and repeatable.

12.5.4. Data Version Control

Git alone is not well suited for large, frequently changing binary files. Instead, the approach is to combine git with a **data version control** tool. These tools essentially replace the data (often called

Note that Quarto doesn't follow project-wide engine yet #157

The screenshot shows a GitHub commit comparison for a file named README.md. The commit message indicates that MichaelHatherly merged 1 commit from alecloudenback:patch-1 into PumasAI:main on Jun 20. The diff interface displays two columns of code, with line numbers on the left. Red highlights indicate text that has been removed or changed, while green highlights indicate new text added by the merge commit. A specific line of code is highlighted in green, showing a URL that was added to the original text.

```

@@ -11,7 +11,7 @@
11 >
12 > Starting from the **pre-release**
[`v1.5.29`](https://github.com/quarto-
dev/quarto-cli/releases/tag/v1.5.29)
13 > this engine is available out-of-the-box
with `quarto` when you set `engine: julia` in
14 -> your Quarto notebook files. You don't need
to follow the developer instructions
15 > below.
16
17 ## Developer Documentation

```

Figure 12.2.: A diff shows a file-by-file and line-by-line comparison of what has changed between commits in a codebase. Red indicates something was removed or changed, and green shows what replaced it. Note that even within a line, there's extra green highlighting to show the newly added text while the unchanged text remains a lighter shade.

binary data or a blob) with a content hash in the git snapshots. The actual content referred to by the hash is then located elsewhere (e.g. a hosted server).

i Note 1: Content Hashes

Content hashes are the output of a function that transforms arbitrary data into a string of data. Hashes are very common in cryptography and in areas where security/certainty is important. Eliding the details of how they work exactly, what's important to understand for our purposes is that a content hash function will take arbitrary bits and output the same set of data each time the function is called on the original bits.

For example, it might look something like this:

```
using SHA

data = "A data file contents 123"
bytes2hex(sha256(codeunits(data)))
```

"b37600bffd200c98b9bb9f8deef14e1612851fa2ff5e22f24bff870423607c66"

If it's run again on the same inputs, you get the same output:

```
# Same input → same hash
bytes2hex(sha256(codeunits(data)))
```

"b37600bffd200c98b9bb9f8deef14e1612851fa2ff5e22f24bff870423607c66"

And if the data is changed even slightly, then the output is markedly different:

```
# Slight change → very different hash
data2 = "A data file contents 124"
bytes2hex(sha256(codeunits(data2)))
```

"773c90b32c6a8b927d5f7e422e2bd5240db587edcaa9cf6b753273b097782080"

Tip

Use a cryptographic hash (e.g., SHA-256) to derive a stable, deterministic content hash. Base.hash is intentionally randomized for security and is not suitable for content-addressed storage.

This is used in content-addressed systems like Data Version Control (Section 12.5.4) and Artifacts (Section 12.6.3) to ask for, and confirm the accuracy of, data instead of trying to address the data by its location. That is, instead of trying to ask to get data from a given URL (e.g. <http://adatasource.com/data.csv>) you can set up a system which keeps track of available locations for the data that matches the content hash. Something like (in Julia-ish pseudocode):

```
storage_locations = Dict(
    "0x4d7e8e449af1c48" => [
        "http://datasets.com/1231234",
        "http://companyintranet.com/admin.csv",
        "C:/Users/your_name/Documents/Data/admin.csv"
    ]
)

function get_data(hash,locations)
    for location in locations[hash]
        if is_available(location)
            return get(location)
        end
    end
    # if loop didn't return data
    return nothing
end
```

12.6. Distributing the Package

Once you have created a package, the next best feeling after having it working is having someone else also use the tool. Julia has a robust way to distribute and manage dependencies. This section will cover essential and related topics to distributing your project publicly or with an internal team.

12.6.1. Registries

Registries are a way to keep track of packages that are available to install, which is more complex than it might seem at first. The registry needs to keep track of:

- What dependencies your package has.
- Which versions of the dependencies your package is compatible with.
- Metadata about the package (name, unique identifier, authors).
- Versions of your package that you have made available and the Git hash associated with that version for provenance and tracking
- The location where your package's repository lives so that the user can grab the code from there.

The Julia General Registry (“General”) is the default registry that comes loaded when installing Julia. From a capability standpoint, there’s nothing that separates General from other registries, including ones that you can create yourself. At its core, a registry can be seen as a Git repository where each new commit just adds information associated with the newly registered package or version of a package.

For distributing packages in a private, or smaller public group see Section 23.9.2.1.

12.6.1.1. General Registry and other Hosted Registries

At its core, General is essentially the same as a local registry described in the prior section. However, there's some additional infrastructure supporting General. Registered packages get backed up, cached for speed, and multiple servers across the globe are set up to respond to Pkg requests for redundancy and latency. Nothing would stop you from doing the same for your own hosted registry if it got popular enough!



Tip

A local registry is a great way to set up internal sharing of packages within an organization. Services do exist for "managed" package sharing, adding enterprise features like whitelisted dependencies, documentation hosting, a 'hub' of searchable packages.

12.6.2. Versioning

Versioning is an important part of managing a complex web of dependencies. Versioning is used to let both users and the computer (e.g. Pkg) understand which bits of code are compatible with others. For this, consider your model/program's **Application Programming Interface** (API). The API is essentially defined by the outputs produced by your code given the same inputs. If the same inputs are provided, then for the same API version the same output should be provided. However, this isn't the only thing that matters. Another is the documentation associated with the functionality. If the documentation said that a function would work a certain way, then your program should follow that (or fix the documentation)!

Another case to consider is what if new functionality was *added*? Old code need not have changed, but if there's new functionality, how to communicate that as an API change? This is where **Semantic Versioning** (SemVer) comes in: *semantic* means that your version of something is intended to convey some sort of meaning over and above simply incrementing from v1 to v2 to v3, etc.

12.6.2.1. Semantic Versioning

Semantic Versioning (SemVer) is one of the most popular approaches to software versioning. It's not perfect but has emerged as one of the most practical ways since it gets a lot about version numbering right. Here's how SemVer is defined³:

Given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make incompatible API changes
- MINOR version when you add functionality in a backward compatible manner
- PATCH version when you make backward compatible bug fixes

So here are some examples of SemVer, if our package's functionality for v1.0.0 is like this:

```
""" my_add(x,y)

Add the numbers x and y together
"""

my_add(x,y) = x - y
```

³You can read more at SemVer.org.

12. Applying Software Engineering Practices

Patch change (v1.0.1): Fix the bug in the implementation:

```
""" my_add(x,y)
Add the numbers x and y together
"""
my_add(x,y) = x + y
```

This is a patch change because it fixes a bug without changing the API. The function still takes two arguments and returns their sum, as originally documented.

Minor change (v1.1.0): Add new functionality in a backward-compatible manner:

```
""" my_add(x,y)
Add the numbers x and y together

my_add(x,y,z)
Add the numbers x, y, and z together
"""
my_add(x,y) = x + y
my_add(x,y,z) = x + y + z
```

This is a minor change because it adds new functionality (the ability to add three numbers) while maintaining backward compatibility with the existing two-argument version.

Major change (v2.0.0): Make incompatible API changes:

```
""" add_numbers(numbers...)
Add any number of input arguments together. This
function replaces 'my_add' from prior versions.
"""
add_numbers(numbers...) = sum(numbers)
```

This is a major change because it fundamentally alters the API. The function name has changed, and it now accepts any number of arguments instead of specifically two or three. This change is not backward compatible with code using the previous versions, so a major version increment is necessary.

Note

Numbers need not roll over to the next digit when they hit 10. That is, it's perfectly valid to go from v1.09.0 to v1.10.0 in SemVer.

Tip

Sometimes you'll see a package with a version that starts with zero, such as v0.23.1. **We recommend that as soon as you register a package, to make it a v1.** v1 need not indicate the package is "complete" (what software is?), so don't hold back on calling it v1. You're letting users install it easily, so you might as well call it the first version and move on!

According to SemVer's rules:

*Major version zero (0.y.z) is for initial development. Anything MAY change at any time.
The public API SHOULD NOT be considered stable.*

Most packages put an upper bound on compatibility so that major or minor changes in upstream packages are less likely to cause issues in their own packages. This can be somewhat painful to depend on a package which has a major version 0 (i.e. v0.x.x). You have to assume minor version bumps are backward incompatible changes, but the author might intend it to be a feature release. You should have skepticism of just upgrading to the new version of the dependency. It takes work on the downstream dependencies to decide if they should upgrade, adding mental and time loads to other authors and users.

This is why once you have anyone else depending on the package, going straight to v1 is helpful because you have finer-grained control over the release versioning and the meaning behind each new version number.

12.6.3. Artifacts

Artifacts are a way to distribute content-addressed (Note 1) data and other dependencies. An example use case is if you want to distribute some demonstration datasets with a package. When a package is added or updated, the associated data is pulled and un-archived by Pkg instead of the author of the package needing to manually handle data dependencies. Aside from this convenience, it means that different packages could load the same data without duplicating the data download or storage (since the data is content-addressed). The use-case is not real-time data, as the content-hash can only be updated per package version.

For example, the MortalityTables.jl package redistributes various publicly available, industry mortality tables. Inside the repository, there's an `Artifacts.toml` file specified like:

```
#| code-overflow: wrap
#/Artifacts.toml

["mort.soa.org"]
git-tree-sha1 = "6164a6026f108fe95828b689fc3b992acb7c3" ①

[[["mort.soa.org"].download]]
sha256 =
  ↳ "6f5eb4909564b55a3397ccf4f4c74290e002f7e2e2474cebeb224bb23a9a2606" ②
url =
  ↳ "https://github.com/JuliaActuary/Artifacts/raw/v2020-02-15/mort.soa.org/2020-02-15.t
```

- ① The sha1 hash of the un-archived data once downloaded, used to verify that extraction was successful.
- ② The sha256 hash of the archived (compressed) data to ensure that the data downloaded was as intended.

Then, within the package the data artifact can be referenced and handled by the artifact system rather than needing to manually handle it. That is, the data is reference-able like this:

```
using Pkg.Artifacts
table_dir = artifact"mort.soa.org"
```

- ① `artifact"..."` is a string macro, which is a special syntax for macros that interact with strings. `md"..."` is another example, specifying that the content of the string is Markdown content.

As opposed to something like this:

12. Applying Software Engineering Practices

```
# pseudo Julia code
table_dir = if is_first_package_run

    data_path = download("url_of_data.tar.gz") # download to a temp location
    mv(data_path, "/somewhere/to/keep/data.tar.gz") # move to a 'permanent'
        ↳ location
    extract("/somewhere/to/keep/data.tar.gz") # extract contents
    "/somewhere/to/keep/data/" # return data path
else
    "/somewhere/to/keep/data/" # return pre-processed path
end
```

Note

Utility Packages such as `ArtifactUtils.jl` can assist in creating correct entries for `Artifacts.toml` files.

Note

Artifacts support `.tar` (uncompressed) and `.tar.gz` because that compression format enjoys more universal support and features than the `.zip` format most common on Windows systems.

12.7. Example Repository Structure

A well-structured Julia package demonstrates key software engineering principles in action. The `JuliaTemplateRepo` repository demonstrates best practices for:

- Logical file organization and code structure
- Comprehensive test coverage and continuous integration
- Clear, accessible documentation with examples
- Standard tooling configuration for package development

This open-source template serves as a reference implementation.

The `PkgTemplates.jl` package will allow you to create an empty repository with all of the testing, documentation, Git, and continuous integration scaffolding already in place.

13. Elements of Computer Science

CHAPTER AUTHORED BY: ALEC LOUDENBACK

"Fundamentally, computer science is a science of abstraction—creating the right model for a problem and devising the appropriate mechanizable techniques to solve it. Confronted with a problem, we must create an abstraction of that problem that can be represented and manipulated inside a computer. Through these manipulations, we try to find a solution to the original problem." — Al Aho and Jeff Ullman (1992)

13.1. Chapter Overview

Computer science concepts adapted for financial professionals: computability, complexity, algorithms, data structures, and advanced testing techniques.

13.2. Computer Science for Financial Professionals

Computer science as a term can be a bit misleading because of the overwhelming association with the physical desktop or laptop machines that we call "computers". The discipline of computer science is much richer than consumer electronics: at its core, computer science concerns itself with areas of research and answering tough questions—many of which surface immediately in quantitative finance:

- **Algorithms and Optimization.** How can a problem be solved efficiently? How can that problem be solved *at all*? Given constraints, how can we find an optimal solution for rebalance schedules, scenario selection, or funding plans?
- **Theory of Computation.** What sorts of questions are even answerable? Is an answer easy to compute or will resolving it require more resources than the entire known universe? Will a computation ever stop calculating (e.g., will a nested capital calculation or Monte Carlo loop converge in time)?
- **Data Structures.** How to encode, store, and use data? How does that data relate to each other and what are the trade-offs between different representations of that data?
- **Information Theory**¹. Given limited data, what *can* be known or inferred from it?

For a reader in the twenty-first century, we hope that it is patently obvious how impactful applied computer science — enabling the internet, artificial intelligence, computational photography, safety control systems, etc. — has been to our lives. It is a testament to the utility of being able to harness computer science ideas for practical use.

Advances often occur at the boundary between disciplines. This chapter brings finance and computer science together, equipping you with the language and concepts to leverage computer science's most relevant ideas.

¹This topic will be covered in Chapter 14.

13.3. Algorithms

An **Algorithm** is a general term for a process that transforms an input to an output. It's the set of instructions dictating how to carry out a process. That process needs to be specified in sufficient detail to be able to call itself an algorithm (versus a *heuristic* which lacks that specificity).

An algorithm might be the directions to accomplish the following task: summing a series of consecutive integers from 1 to n . There are multiple ways that this might be accomplished, each one considered a distinct algorithm:

- iterating over each number and summing them up (starting with the smallest number)
- iterating over each number and summing them up (starting with the largest number)
- summing up the evens and then the odds, then adding the two subtotals
- and many more distinct algorithms...

We will look at some specific examples of these alternate algorithms as we introduce the next topic, computational complexity.

13.4. Complexity

13.4.1. Computational Complexity

We can characterize the computational complexity of a problem by looking at how long an algorithm takes to complete a task when given an input of size n . We can then compare two approaches to see which is computationally less complex for a given n . This is a way of systematically evaluating an algorithm to determine its efficiency when being computed.

Warning

Note that computational complexity isn't quite the same as how fast an algorithm will run on your computer, but it's a very good guide. Modern computer architectures can sometimes execute multiple instructions in a single cycle of the CPU making an algorithm that is, on paper, slower than another actually run faster in practice. Additionally, sometimes algorithms are able to substantially limit the number of *computations* to be performed, at the expense of using a lot more *memory* and thereby trading CPU usage with RAM usage.

You can think of computational complexity as a measure of how much work is to be performed. Sometimes the computer is able to perform certain kinds of work more efficiently.

Further, when we analyze an algorithm, recall that ultimately our code gets translated into instructions for the computer hardware. Some instructions are implemented in a way that for any type of number (e.g. floating point), it doesn't matter if the number is 1.0 or 0.41582574300044717; the operation will take the exact same time and number of instructions to execute (e.g. for the addition operation).

Sometimes a higher-level operation is implemented in a way that takes many machine instructions. For example, division instructions may require many CPU cycles when compared to addition or multiplication. Sometimes this is an important distinction and sometimes not. For this book we will ignore this granularity of analysis.

13.4.1.1. Example: Sum of Consecutive Integers

Take for example the problem of determining the sum of integers from 1 to n . We will explore three different algorithms and the associated computational complexity for them.

13.4.1.1.1. Constant Time

A mathematical proof can show a simple formula for the result. This allows us to compute the answer in **constant time**, which means that for any n , our algorithm is essentially the same amount of work.

```
nsum_constant(n) = n * (n + 1) / 2

nsum_constant (generic function with 1 method)
```

In this we see that we perform three operations: a multiplication, a sum, and a division, no matter what n is. If n is 10_000_000 we'd expect this to complete in about a single unit of time.

13.4.1.1.2. Linear Time

This algorithm performs a number of operations which grows in proportion with n by individually summing up each element in 1 through n :

```
function nsum_linear(n)
    result = 0
    for i in 1:n
        result += i
    end

    result
end

nsum_linear (generic function with 1 method)
```

If n were 10_000_000, we'd expect it to run with roughly 10 million operations, or about 3 million times as many operations as the constant time version. We can say that this version of the algorithm will take approximately n steps to complete.

13.4.1.1.3. Quadratic Time

What if we were less efficient, and instead we were only ever able to increment our subtotal by one. That is, instead of adding up 1 + 3, we had to instead do four operations: 1 + 1 + 1 + 1. We can add a second loop which increments our result by a unit instead of simply adding the current i to the running total $result$. This makes our algorithm work much harder since it has to add numbers so many more times (Recall that to a computer adding two numbers is the same computational effort regardless of what the numbers are).

```
function nsum_quadratic(n)
    result = 0
    for i in 1:n
        for j in 1:i
            result += 1
        end
    end
end
```

(1)

(2)

13. Elements of Computer Science

```

    result
end

```

- ① The outer loop with iterator i loops over the integers 1 to n.
- ② The inner loop with iterator j does the busy work of adding 1 to our subtotal i times.

`nsum_quadratic (generic function with 1 method)`

Breaking down the steps:

- When i is 1 there is 1 addition in the inner loop
- When i is 2 there are 2 additions in the inner loop
- ...
- When i is n there are n additions in the inner loop

Therefore, this computation takes $(1 + 2 + \dots + n)$ steps to complete. Algebraically, this simplifies down to our constant time formula: it requires $n * (n + 1) / 2$ or $(n^2 + n)/2$ steps to complete.

13.4.1.2. Comparison

13.4.1.2.1. Big-O Notation

We can categorize the above implementations using a convention called **Big-O Notation**² which is a way of distilling and classifying computational complexity. We characterize the algorithms by the most significant term in the total number of operations. Table 13.1 shows the description, order, and approximate step count for each example.

Table 13.1.: Complexity comparison for the three sample cases of summing integers from 1 to n.

| Function | Computational Cost | Complexity Description | Big-O Order | Steps ($n = 10,000$) |
|-----------------------------|--------------------|------------------------|-------------|------------------------|
| <code>nsum_constant</code> | fixed | Constant | $O(1)$ | ~1 |
| <code>nsum_linear</code> | n | Linear | $O(n)$ | ~10,000 |
| <code>nsum_quadratic</code> | $(n^2 + n)/2$ | Quadratic | $O(n^2)$ | ~100,000,000 |

Table 13.2 shows a comparison of a more extended set of complexity levels. For the most complex categories of problems, the cost to compute grows so fast that it boggles the mind. What sorts of problems fall into the most complex categories? $O(2^n)$, or exponential complexity, examples include the traveling salesperson problem³ if solved with dynamic programming or the recursive approach to calculating the n th Fibonacci number. The beastly $O(n!)$ algorithms include brute force solving the traveling salesperson problem or enumerating all partitions of a set. In financial modeling, we may encounter these sorts of problems in portfolio optimization (using the brute-force approach of testing every potential combination of assets to optimize a portfolio).

Table 13.2.: Different Big-O Orders of Complexity

| Big-O Order | Description | $n = 10$ | $n = 1,000$ | $n = 1,000,000$ |
|-------------|---------------|----------|-------------|-----------------|
| $O(1)$ | Constant Time | 1 | 1 | 1 |

²"Big-O", so named because the 'O' denotes the order of growth, as in $O(1)$. $O(n)$, etc.

³The Traveling Salesperson Problem is a classic computer science problem where you need to find the shortest possible route that visits each city in a given set exactly once and returns to the starting city. It's a seemingly simple problem that becomes computationally intensive very quickly as the number of cities increases.

| Big-O Order | Description | $n = 10$ | $n = 1,000$ | $n = 1,000,000$ |
|-----------------------|-------------------|-----------|------------------|---------------------|
| $O(n)$ | Linear Time | 10 | 1,000 | 1,000,000 |
| $O(n^2)$ | Quadratic Time | 100 | 1,000,000 | 10^{12} |
| $O(\log(n))$ | Logarithmic Time | 3 | 10 | 20 |
| $O(n \times \log(n))$ | Linearithmic Time | 33 | 10,000 | 20,000,000 |
| $O(2^n)$ | Exponential Time | 1,024 | $\sim 10^{300}$ | $\sim 10^{301029}$ |
| $O(n!)$ | Factorial Time | 3,628,800 | $\sim 10^{2567}$ | $\sim 10^{5565708}$ |

i Note

We care only about the most significant term because when n is large, the most significant term tends to dominate. For example, in our quadratic time example which has $(n^2 + n) \div 2$ steps, if n is a large number like 10^6 , then we see that it will result in:

$$\begin{aligned} \frac{n^2 + n}{2} &= \frac{(10^6)^2 + 10^6}{2} \\ &= \frac{10^{12} + 10^6}{2} \end{aligned}$$

10^{12} is significantly more important than $\frac{10^6}{2}$ (two million times as important, to be precise). This is why Big-O notation reduces the problem down to only the most significant complexity cost term.

If n is small then we don't really care about computational complexity in general. This is a lesson for our efforts as developers: focus on the most intensive parts of calculations when looking to optimize, and don't worry about seldom-used portions of the code.

13.4.1.2.2. Empirical Results

The preceding examples of constant, linear, and quadratic times are *conceptually* correct but if we try to run them in practice we see that the description doesn't seem to hold at all for the linear time version, as it runs as quickly as the constant time version.

```
using BenchmarkTools
@btime nsum_constant(10_000)

0.916 ns (0 allocations: 0 bytes)
50005000

@btime nsum_linear(10_000)

1.666 ns (0 allocations: 0 bytes)
50005000
```

13. Elements of Computer Science

```
@btime nsum_quadratic(10_000)
```

```
1.108 μs (0 allocations: 0 bytes)
```

```
50005000
```

What happened was that the compiler was able to understand and optimize the linear version such that it effectively transformed it into the constant time version and avoided the iterative summation that we had written. For examples that are simple enough to use as a teaching problem, the compiler can often optimize different written code down to the same efficient machine code (this is the same Triangular Number optimization we saw in Section 5.4.3.4).

13.4.1.3. Expected versus worst-case complexity

Another consideration is that there may be one approach that performs better in the majority of cases, at the expense of having very poor performance in specific cases. Sometimes we may risk those high cost cases if we expect the benefit to be worthwhile on the rest of the problem set.

This often happens when the data we are working with has some concept of “distance”. For example, in multi-stop route planning we can use the idea that it’s likely to be more efficient to visit nearby destinations first. Generally, this works, but sometimes the nearest distance actually has a high cost (such as needing to avoid real-world obstacles in the way that force you to drive past other farther away locations to get there).

13.4.2. Space Complexity

So far we have focused on computational complexity, however similar analysis could be performed for **space complexity**, which is how much computer memory is required to solve a problem. Sometimes, an algorithm will trade computational complexity for space complexity. That is, we might be able to solve a problem much faster if we have more memory available.

For example, there has been research to improve the computational efficiency of matrix multiplication that do indeed run faster than traditional techniques. However, those algorithms don’t get implemented in general linear algebra libraries because they require way more memory than is available!

13.4.3. Complexity: Takeaways

The idea of algorithmic complexity is important because it grounds us in the harsh truth that some problems are *very* difficult to compute. It’s in these cases that a lot of the creativity and domain specific heuristics can become the foremost consideration.

We must remember to be thoughtful about the design of our models. When searching for additional performance, hunt for the “loops-within-loops”—that is where combinatorial explosions tend to happen. Focusing on the places that have large n or poor Big-O order can transform the performance of the overall model. Sometimes the fundamental complexity of the problem forbids greater efficiency, but many finance workflows benefit from reframing the problem (e.g. relaxing binary constraints or using approximation algorithms) before brute force becomes unavoidable.

We’ll conclude this sub-section with an example demonstrating complexity implications on portfolio optimization, a common problem in financial modeling.

13.4.4. Finance Example: Complexity and Portfolio Selection

Many real portfolio decisions include discrete choices: buy or skip a lot, include or exclude an ETF, cap the number of names, or enforce minimum ticket sizes. These discrete elements quickly turn an otherwise smooth optimization into a combinatorial search.

Consider a simplified subset selection problem with n candidate assets. Each asset i has an expected return r_i and a “cost” c_i (e.g., capital required for a minimum lot). We wish to select a subset that maximizes total expected return subject to a budget B . This is the classic 0–1 knapsack problem:

$$\max_{x \in \{0,1\}^n} \sum_{i=1}^n r_i x_i \quad \text{subject to} \quad \sum_{i=1}^n c_i x_i \leq B$$

- Brute force enumerates all 2^n subsets to find the best feasible one. Complexity: $O(2^n \cdot n)$ work; it explodes rapidly.
- A greedy heuristic sorts by value-to-cost ratio r_i/c_i and takes items while budget remains. Complexity: $O(n \log n)$; fast, but not guaranteed optimal.
- Branch-and-bound (B&B) remains worst-case exponential, but prunes large parts of the search using an optimistic upper bound (here, the fractional knapsack relaxation), often exploring far fewer nodes in practice.

Rather than relying on wall-clock times (which depend on your machine), we count “visited” nodes—how many partial solutions each method examines—to illustrate growth in work.

```
using Random

# Generate a synthetic instance. Pass an RNG for reproducibility.
function synth_instance(rng::AbstractRNG, n::Int; budget_ratio::Float64=0.35)
    r = 0.04 .+ 0.08 .* rand(rng, n)      # expected returns in [4%, 12%]
    c = Float64.(rand(rng, 50:150, n))   # costs (min lot "sizes"), as reals
    B = budget_ratio * sum(c)            # budget as a real, not floored
    return r, c, B
end

# Greedy heuristic: sort by value-to-cost ratio
# and pack until budget is exhausted.
function greedy_knapsack(r::AbstractVector{<:Real},
    c::AbstractVector{<:Real},
    B::Real)
    n = length(r)
    ratio = r ./ c
    idx = sortperm(ratio, rev=true)        # highest ratio first
    total = 0.0
    cost = 0.0
    chosen = falses(n)
    @inbounds for i in idx
        if cost + c[i] <= B + 1e-12
            chosen[i] = true
            total += r[i]
            cost += c[i]
        end
    end
    return total, chosen
end
```

```

# Brute-force: enumerate all 2^n subsets
# (only feasible for small n).
function brute_force_knapsack(r::AbstractVector{<:Real},
    c::AbstractVector{<:Real},
    B::Real)
    n = length(r)
    best = -Inf
    bestmask = 0
    visited = 0
    total_subsets = (Int(1) << n)
    @inbounds for mask in 0:(total_subsets-1)
        visited += 1
        value = 0.0
        cost = 0.0
        for i in 1:n
            if ((mask >>> (i - 1)) & 0x1) == 1
                value += r[i]
                cost += c[i]
                if cost > B
                    break
                end
            end
        end
        if cost <= B && value > best
            best = value
            bestmask = mask
        end
    end
    chosen = falses(n)
    for i in 1:n
        chosen[i] = ((bestmask >>> (i - 1)) & 0x1) == 1
    end
    return best, chosen, visited
end

# Branch-and-Bound with a fractional-knapsack upper bound (optimistic).
# The upper bound relaxes  $x_i \in \{0,1\}$  to  $x_i \in [0,1]$  and
# "fills" the remaining budget fractionally.
function branch_and_bound_knapsack(r::AbstractVector{<:Real},
    c::AbstractVector{<:Real},
    B::Real)
    n = length(r)
    rs = Float64.(r)
    cs = Float64.(c)
    Bf = float(B)

    ratio = rs ./ cs
    order = sortperm(ratio, rev=true)
    rs = rs[order]
    cs = cs[order]

    best = Ref(-Inf)           # current best objective
    best_choice = falses(n)    # in sorted order

```

```

visited = Ref(0)           # node counter
choice = falses(n)         # current partial choice (sorted order)

@inline function ubound(idx::Int, value::Float64, cost::Float64)
    if cost > Bf
        return -Inf
    end
    cap = Bf - cost
    ub = value
    @inbounds for j in idx:n
        if cs[j] <= cap + 1e-12
            ub += rs[j]
            cap -= cs[j]
        else
            ub += rs[j] * (cap / cs[j])      # fractional "fill" for bound
            break
        end
    end
    return ub
end

function dfs(idx::Int, value::Float64, cost::Float64)
    visited[] += 1
    if idx > n
        if cost <= Bf && value > best[]
            best[] = value
            best_choice .= choice
        end
        return
    end
    # Prune if even the optimistic bound cannot beat current best.
    if ubound(idx, value, cost) <= best[] + 1e-12
        return
    end
    # Branch: include item idx
    choice[idx] = true
    dfs(idx + 1, value + rs[idx], cost + cs[idx])
    # Branch: exclude item idx
    choice[idx] = false
    dfs(idx + 1, value, cost)
end

dfs(1, 0.0, 0.0)

# Map best_choice back to the original indexing
selected = falses(n)
@inbounds for (k, i) in pairs(order)
    selected[i] = best_choice[k]
end
return best[], selected, visited[]
end

# Small demonstration: brute force is feasible at n ≈ 20;
# for n ≈ 40 we skip brute force.

```

13. Elements of Computer Science

```

let
    rng = MersenneTwister(1234)

    # Small instance where brute force is still possible
    n_small = 20
    r, c, B = synth_instance(rng, n_small; budget_ratio=0.35)

    bf_val, bf_sel, bf_visited = brute_force_knapsack(r, c, B)
    bb_val, bb_sel, bb_visited = branch_and_bound_knapsack(r, c, B)
    gr_val, gr_sel = greedy_knapsack(r, c, B)

    println("n = $n_small")
    println(" total subsets = ", 1 << n_small)                      # 1,048,576
    println(" brute force visited = ", bf_visited,
           ", optimal value = ", bf_val)
    println(" branch-and-bound visited = ", bb_visited,
           ", optimal value = ", bb_val)
    println(" greedy value = ", gr_val,
           ", optimality gap = ", bf_val - gr_val)

    # Larger instance—note how the search space explodes ( $2^{40} \approx 10^{12}$ )
    n_large = 40
    r2, c2, B2 = synth_instance(rng, n_large; budget_ratio=0.35)
    bb2_val, bb2_sel, bb2_visited = branch_and_bound_knapsack(r2, c2, B2)
    gr2_val, gr2_sel = greedy_knapsack(r2, c2, B2)

    println("\nn = $n_large")
    println(" total subsets = ", BigInt(1) << n_large)                 # ≈ 1.1×10^12
    println(" branch-and-bound visited = ", bb2_visited,
           ", optimal value = ", bb2_val)
    println(" greedy value = ", gr2_val,
           ", optimality gap = ", bb2_val - gr2_val)
end

n = 20
total subsets = 1048576
brute force visited = 1048576, optimal value = 0.7605388597519076
branch-and-bound visited = 317, optimal value = 0.7605388597519077
greedy value = 0.703897494058528, optimality gap = 0.05664136569337963

n = 40
total subsets = 1099511627776
branch-and-bound visited = 115, optimal value = 1.4909468614490333
greedy value = 1.4909468614490333, optimality gap = 0.0

```

What this shows in practice:

- Brute force scales as 2^n . At $n = 20$, there are 1,048,576 subsets—still manageable. At $n = 40$, there are $\approx 10^{12}$ subsets—completely infeasible to enumerate.
- Greedy is fast and often near-optimal, especially when ratios r_i/c_i align with the true optimum. However, it can be arbitrarily sub-optimal in adversarial cases.
- Branch-and-bound exploits structure. The fractional-knapsack bound is optimistic—it assumes you could take fractions of remaining items—so any subtree whose bound cannot beat the current best is safely pruned. Worst-case complexity remains exponential, but typical search can drop from billions of nodes to thousands or millions.

Financial Modeling Pro Tip

In realistic portfolio construction, practitioners frequently avoid pure subset selection by relaxing to continuous weights (e.g., mean-variance) or by formulating a mixed-integer model and using modern MILP solvers. These solvers implement sophisticated branch-and-bound/branch-and-cut strategies, presolve, and cutting planes. When exact optimality is not essential, heuristics (greedy, local search, genetic/evolutionary methods) can deliver high-quality portfolios quickly under operational constraints (cardinality, sector caps, turnover budgets).

See Chapter 27 for more discussion of portfolio optimization.

13.5. Data Structures

Data structures concern how data is represented—both conceptually and in computer memory.

For example, how should the string “abcdef” be represented and analyzed?

There are many common data structures and many specialized subtypes. We will describe some of the most common ones here. Julia has many data structures available in the Base library, but an extensive collection of other data structures can be found in the DataStructures.jl package.

13.5.1. Arrays

An **array** is a contiguous block of memory containing elements of the same type, accessed via integer indices. Arrays have fast random access and are the fastest data structure for linear/iterated access of data.

In Julia, an array is a very common data structure and is implemented with a simple declaration, such as:

```
x = [1,2,3]
```

In memory, the integers are stored as consecutive bits representing the integer values of 1, 2, and 3. The actual bits in memory would look like this (with the different integers shown on separate lines for clarity):

This is great for accessing the values one-by-one or in consecutive groups, but it's not efficient if values need to be inserted in-between. For example, if we wanted to insert 0 between the 1 and 2 in x, then we'd need to overwrite the second position in the array, ask the operating system to allocate more memory⁴, and re-write the bytes that come after our new value. Inserting values at the end (`push!(array, value)`) is usually fast unless more memory needs to be allocated.

⁴In practice, the operating system may have already allocated space for an array that's larger than what the program is actually using so far, so this step may be 'quick' at times, while other times the operating system may actually need to extend the block of memory allocated to the array.

13.5.2. Linked Lists

A **linked list** is a chain of nodes where each node contains a value and a pointer to the next node. Linked lists allow for efficient insertion and deletion but slower random access compared to arrays.

In Julia, a simple linked list node can be implemented generically so it remains type-stable:

```
mutable struct Node{T}
    value::T
    next::Union{Node{T}, Nothing}
end

z = Node(3, nothing)
y = Node(2, z)
x = Node(1, y)
```

① `Nothing` represents the end of the linked list.

Inserting a new node between existing nodes is efficient - if we wanted to insert a new node between the ones with value 2 and 3, we could do this:

```
a = Node(0, z)                                ①
y.next = a                                     ②
```

Accessing the n th element requires traversing the list from the beginning to check each `Node`'s `next` value. This iterative approach makes it $O(n)$ time complexity for random access. This is in contrast to an array, where you know right away where the n th item will be in the data structure.

Also, the linked list is one-directional. Items further down the chain don't know what node points to them, so it's impossible to traverse the list backwards.

There are many related implementations that make random access or traversal in reverse order more efficient, such as doubly-linked lists or "trees" (Section 13.5.6) that organize the data not as a chain but as a tree with branching nodes.

13.5.3. Records/Structs

A **record** (or `struct` in Julia) is an aggregate of named fields, typically of fixed size and sequence. Records group related data together. We've encountered structs in Section 5.4.7, but here we'll note that simple structs with primitive fields can themselves be represented without creating pointers to the data stored:

```
struct SimpleBond
    id::Int
    par::Float64
end

struct LessSimpleBond
    id::String
    par::Float64
end

a = SimpleBond(1, 100.0)
b = LessSimpleBond("1", 100.0)
```

`isbits(a), isbits(b)`
`(true, false)`

Because `a` is comprised of simple elements, it can be represented as a contiguous set of bits in memory. It would look something like this in memory:

- ① The bits of 1
 - ② The bits of 100.0

In contrast, the `LessSimpleBond` uses a `String` to represent the ID of the bond. In Julia, `String` is an immutable type that internally references a buffer of bytes; because it holds a reference, a struct containing a `String` is not an `isbits` type.

- ① A pointer/reference to the array of characters that comprise the string ID
 - ② The bits of 100.0

In performance critical code, having data that is represented with simple bits instead of references/pointers can be much faster (see Chapter 30 for an example).

i Note

For many mutable types, there are immutable, bits-types alternatives. For example:

- Arrays have a `StaticArray` counterpart (from the `StaticArrays.jl` package).
 - Strings have `InlineStrings` (from the `InlineStrings.jl` package) which use fixed-width representations of strings.

The downsides to the immutable alternatives (other than the loss of potentially desired flexibility that mutability provides) are that they can be harder on the compiler (more upfront compilation cost) to handle the specialized cases involved.

13.5.4. Dictionaries (Hash Tables)

13.5.4.1 Hashes and Hash Functions

Hashes are the result of a **hash function** that maps arbitrary data to a fixed size value. It's sort of a "one way" mapping to a simpler value which has the benefits of:

1. One way so that if someone knows the hashed value, it's *very* difficult to guess what the original value was. This is most useful in cryptographic and security applications.
 2. Creating (probabilistically) unique IDs for a given set of data.

For example, we can calculate an **SHA hash** on any data:

```
import SHA
let
    a = SHA.sha256("hello world") ▷ bytes2hex
    b = SHA.sha256(rand(UInt8, 10^6)) ▷ bytes2hex
```

13. Elements of Computer Science

```
    println(a)
    println(b)
end

b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9
a5c535190816703a17b50f3a2b9e4e24ab5659d7525c58343d8c46ec5254eaf8
```

We can easily verify that the sha256 hash of "hello world" is the same each time, but it's virtually impossible to guess "hello world" if we are just given the resulting hash. This is the premise of trying to "crack" a password when the stored password hash is stolen (real systems "salt" and "stretch" hashes to make the attack even harder—these topics are out of scope here).

One way to check if two sets of data are the same is to compute the hash and see if the resulting hashes are equal. For example, you might want to see if two data files with different names contain the same data—if the hashes differ, the files are definitely different; if the hashes match, the files are almost certainly identical (collisions are theoretically possible but astronomically unlikely with cryptographic hashes like SHA-256).

13.5.4.2. Dictionaries

Dictionaries map a *key* to a *value*. More specifically, they use the *hash of a key* to store a reference to the *value*.

Dictionaries offer constant-time average case access but must handle potential collisions of keys (generally, more robust collision handling means a higher fixed cost for access).

Here's an illustrative portfolio of assets indexed by CUSIPs:

```
assets = Dict(
    # CUSIP ⇒ Asset
    "037833AH4" ⇒ Bond("General Electric",...),
    "912828M80" ⇒ Equity("AAPL",...),
    "594918BQ1" ⇒ Bond("ENRON",...),
)
```

Then, lookup is performed by indexing the dictionary by the desired key:

```
assets["037833AH4"] # gives the General Electric Bond
```

13.5.5. Graphs

A **graph** is a collection of nodes (also called vertices) connected by *edges* to represent relationships or connections between entities. Graphs are versatile data structures that can model exposure networks, clearing relationships, or counterparty dependencies, in addition to classics like social networks or transportation systems.

In Julia, a simple graph could be implemented using a dictionary where keys are nodes and values are lists of connected nodes:

```
struct Graph{T}
    nodes::Dict{T, Vector{T}}
end

function add_edge!(graph::Graph{T}, node1::T, node2::T) where {T}
    push!(get!(graph.nodes, node1, T[]), node2)
```

```

    push!(get!(graph.nodes, node2, T[]), node1)
end

g = Graph(Dict{Int, Vector{Int}}())
add_edge!(g, 1, 2)
add_edge!(g, 2, 3)
add_edge!(g, 1, 3)

```

This implementation represents an undirected graph. For a directed graph, you would only add the edge in one direction.

To make graphs a bit more concrete in the financial context: in risk analytics, for example, you might treat each node as a legal entity and each edge as an exposure or collateral link. Once the network is encoded, running BFS from a failed counterparty highlights potential contagion paths; centrality measures help identify “too-connected-to-fail” hubs.

Graphs can be traversed using various algorithms such as depth-first search (DFS) or breadth-first search (BFS). These traversals are useful for finding paths, detecting cycles, or exploring connected components.

For more advanced graph operations, the `Graphs.jl` package provides a comprehensive set of tools for working with graphs in Julia.

13.5.6. Trees

A tree is a hierarchical data structure with a root node and child subtrees. Each node in a tree can have zero or more child nodes, and every node (except the root) has exactly one parent node. Trees are widely used for representing hierarchical relationships, organizing data for efficient searching and sorting, and in various algorithms.

A simple binary tree node in Julia could be implemented as:

```

mutable struct TreeNode{T}
    value::T
    left::Union{TreeNode{T}, Nothing}
    right::Union{TreeNode{T}, Nothing}
end

# Creating a simple binary tree
root = TreeNode(1,
    TreeNode(2,
        TreeNode(4, nothing, nothing),
        TreeNode(5, nothing, nothing)
    ),
    TreeNode(3,
        nothing,
        TreeNode(6, nothing, nothing)
    )
)

```

Trees have various specialized forms, each with its own properties and use cases:

- Binary Search Trees (BST): Each node has at most two children, with all left descendants less than the current node, and all right descendants greater.
- AVL Trees: Self-balancing binary search trees, ensuring that the heights of the two child subtrees of any node differ by at most one.

13. Elements of Computer Science

- B-trees: Generalization of binary search trees, allowing nodes to have more than two children. Commonly used in databases and file systems.
- Trie (Prefix Tree): Used for efficient retrieval of keys in a dataset of strings. Each node represents a common prefix of some keys.

Trees support efficient operations like insertion, deletion, and searching, often with $O(\log n)$ time complexity for balanced trees. They are fundamental in many algorithms and data structures, including heaps, syntax trees in compilers, and decision trees in machine learning.

13.5.7. Data Structures Conclusion

Data structures have strengths and weaknesses depending on whether you want to prioritize computational efficiency, memory (space) efficiency, code simplicity, and/or mutability. Due to the complexity of real world modeling needs, it can be the case that different representations of the data are more natural or more efficient for the use case at hand.

A good rule of thumb is to map the question to the access pattern: sequential valuation → arrays, hierarchical reporting → trees or dictionaries, network relationships → graphs. Picking the right shape for the data often matters more than micro-optimizing the code.

13.6. Verification and Advanced Testing

13.6.1. Formal Verification

Formal verification is a technique used to prove or disprove the correctness of algorithms with respect to a certain formal specification or property. In essence, it's a mathematical approach to ensuring that a system behaves exactly as intended under all possible conditions.

In formal verification, we use mathematical methods to:

1. Create a formal model of the system
2. Specify the desired properties or behaviors
3. Prove that the model satisfies these properties

This process can be automated using specialized software tools called theorem provers or model checkers. In finance, formal methods appear in high-assurance corners such as core valuation libraries, exchange protocols, or regulatory reporting engines—areas where a production bug could have systemic consequences.

13.6.1.1. Formal Verification in Practice

It sounds like the perfect risk management and regulatory technique: prove that the system works exactly as intended. However, there has been very limited deployment of formal verification in industry for several reasons:

1. Incomplete Coverage: It's often impractical to formally verify entire large-scale financial systems. Verification, if at all, is typically limited to critical components.
2. Incomplete Specification: Reasoning through how the system should behave in all scenarios requires contemplating mathematically complete and rigorous possibilities that could occur.
3. Model-Reality Gap: The formal model may not perfectly represent the real-world system, especially in finance where market behavior can be unpredictable.
4. Changing Requirements: Financial regulations and market conditions change rapidly, potentially outdated formal verifications.

5. Performance Trade-offs: Systems designed for easy formal verification might sacrifice performance or flexibility.
6. Cost: The process can be expensive in terms of time and specialized labor.

13.6.2. Property Based Testing

Testing will be discussed in more detail in Chapter 12, but an intermediate concept between Formal Verification and typical software testing is **property-based** testing, which tests for general rules instead of specific examples.

For example, a function which is associative ($(a + b) + c = a + (b + c)$) or commutative ($a + b = b + a$) can be tested with simple examples like:

```
using Test

myadd(a,b) = a + b

@test myadd(1,2) == myadd(2,1) # commutative
@test myadd(myadd(1,2),3) == myadd(1,myadd(2,3)) # associative
```

However, we really haven't proven the associative and commutative properties in general. There are techniques to do this, which are a more comprehensive alternative to testing specific examples above. Packages like `Supposition.jl` provide functionality for this. In actuarial practice, you might encode invariants such as "present value is non-negative under positive discount rates" or "portfolio Greeks add up across sub-portfolios" and let the library hammer your code with random inputs. Note that, like Formal Verification, property-based testing is a more advanced topic.

 **Warning**

Property-based testing is also tied in with concepts related to types we talked about in Section 5.4.1: floating point numbers are not associative when summing more than two numbers at a time: the order in which the floats get added affects the accumulated imprecision.

13.6.3. Fuzzing

Fuzzing is like property-based testing, but instead of testing general rules, it generalizes simple examples using randomness. For example, we could test the commutative property using random numbers, thereby statistically checking that the property holds:

```
@testset for i in 1:10000
    a = rand()
    b = rand()

    @test myadd(a,b) == myadd(b,a)
end
```

This is a good advancement over the simple `@test myadd(1,2) == myadd(2,1)`, in terms of checking the correctness of `myadd`, but it comes at the cost of more computational time and non-deterministic tests. Fuzzing typically seeks to find edge-cases: crashes or unexpected behavior by exploring random/invalid inputs. It is especially useful for feeds that ingest external

13. Elements of Computer Science

data (e.g., vendor trade files) where malformed records are a bigger threat than pure numerical error.

14. Statistical Inference and Information Theory

CHAPTER AUTHORED BY: ALEC LOUDENBACK

"My greatest concern was what to call [the amount of unpredictability in a random outcome]. I thought of calling it 'information,' but the word was overly used, so I decided to call it 'uncertainty.'

When I discussed it with John von Neumann, he had a better idea. Von Neumann told me, 'You should call it entropy, for two reasons. In the first place, your uncertainty function has been used in statistical mechanics under that name, so it already has a name. In the second place, and more important, no one really knows what entropy really is, so in a debate you will always have the advantage.' — Claude Shannon (1971)

14.1. Chapter Overview

A brief introduction to information theory and its foundational role in statistics. Entropy and probability distributions. Bayes' rule and model selection comparison via likelihoods. A brief tour of modern Bayesian statistics.

14.2. Introduction

Statistics has an invaluable role in any data-driven modeling enterprise. As financial professionals dealing inherently with risk and uncertainty, we use probability and statistics to understand, model, and communicate these aspects.

Statistics curricula and practice are undergoing a significant transformation, with a larger focus on information theory and Bayesian methods (as opposed to the common Frequentist methods that have dominated the statistics field for more than a century). Why the change? In short, these methods work better in a wider range of situations and convey more meaningful information about model performance and uncertainty. Until now, computational challenges have limited Bayesian methods to simpler problems, but newer algorithms and better hardware are overcoming this limitation.

In our experience, it is rare that a financial professional has had exposure to non-Frequentist theory and methods. Given how central probability and statistics are to this endeavor, we have drafted this chapter as an introduction - a proper treatment is beyond the scope of this book. Armed with this knowledge, some of the terminology and tools should be more accessible, and possibly included in new financial models.

14.3. Information Theory

Probability, statistics, machine learning, signal processing, and even physics have a foundational link in **information theory**, which is the description and analysis of how much useful data is contained within a signal or dataset. We will work through this with concrete examples.

14.3.1. Example: The Missing Digit

Let's consider the following situation: we are studying a poorly made copy of a financial statement. Amongst many associated exhibits, we are interested in the par value of a particular asset class. Unfortunately, for one reason or another one of the digits is completely indecipherable. Here's what you can read, with the $\underline{}$ indicating that one of the digits is missing from the scanned copy:

32,000, $\underline{}$ 00

It is likely that you quickly formed an opinion on what the missing number is, but let us make that intuition more formal and quantitative.

Given that we know that par values of assets tend to be nice round numbers, our **prior assumption** for what the probability of the missing digit is may be something like the $p(x_i)$ row of Table 14.1. This prior distribution assumes that the missing digit is most likely a 0. We shall call the individual outcomes x_i and the overall set of probabilities $\{x_0, x_1, \dots, x_9\}$ is called X .

The **information content** of an outcome, $h(x)$ is measured in bits and defined as¹:

$$h(x_i) = \log_2 \frac{1}{p(x_i)} = -\log_2 p(x_i) \quad (14.1)$$

Looking at Table 14.1, we can see that the information content of an outcome is *lower* when that outcome has a higher probability than the other potential outcomes. Specifically, if the digit was indeed 0, we have gained less information relative to our expectation than if the missing digit were anything other than 0.

Tip

The information content is sometimes referred to as a measure of *surprise* that one would have when observing a realized outcome. In our missing digit example (Table 14.1), we would not be surprised at all to find out that the missing digit were 0. In contrast, we would be more surprised to find out the digit were an 8.

We can characterize the entire distribution X via the **entropy**, $H(X)$, which is the ensemble's average information content:

$$H(X) = \sum_i p(x_i) \log_2 \frac{1}{p(x_i)} = -\sum_i p(x_i) \log_2 p(x_i) \quad (14.2)$$

The entropy $H(X)$ of the presumed outcomes in Table 14.1 is 0.722 bits.

Table 14.1.: Probability distribution of missing digit, knowing the human inclination to prefer round numbers for par values of assets.

| x_i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|----------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| $p(x_i)$ | .91 | .01 | .01 | .01 | .01 | .01 | .01 | .01 | .01 | .01 |
| $h(x_i)$ | 0.136 | 6.644 | 6.644 | 6.644 | 6.644 | 6.644 | 6.644 | 6.644 | 6.644 | 6.644 |

To be clear, we have taken a non-uniform view on the probability distribution for the missing digit, and we'll refer to this as the **prior assumption** (or just **prior**). This is unashamedly an opinionated

¹Log base two turns out to be the most natural representation of information content as it mimics the fundamental 0 or 1 value bit. A more complete introduction is available in MacKay (2003).

assumption, just like your intuition when you encountered 32,000,-00! All we are doing is giving a quantitative basis for describing this assumption. Taking a view on a prior distribution is quantitatively incorporating previously encountered data and professional judgment. Having a prior assumption like this is completely compatible with information theory.

Our professional judgment notwithstanding: what if we had another colleague who believed humans are completely rational and without bias for certain numbers? They think an asset's par value need not be rounded at all. They argue for a prior distribution consistent with Table 14.2.

With the uniform prior assumption, $H(X) = 3.322$ bits and $h(x_i)$ is also uniform. Note that H is higher for the uniform prior than the prior in Table 14.1. We will not prove it here, but a uniform probability over a set of outcomes is the highest entropy distribution that can be assumed for this problem. A higher entropy prior distribution can typically be viewed as a less biased prior assumption than a lower entropy prior.

Table 14.2.: Probability distribution of missing digit with uniform, maximal entropy for the assumed probability distribution.

| x_i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|----------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| $p(x_i)$ | .10 | .10 | .10 | .10 | .10 | .10 | .10 | .10 | .10 | .10 |
| $h(x_i)$ | 3.322 | 3.322 | 3.322 | 3.322 | 3.322 | 3.322 | 3.322 | 3.322 | 3.322 | 3.322 |

The choice of prior assumption can significantly impact the interpretation and analysis of the missing information. If we have strong reasons to believe that the human bias prior is more appropriate given the context (e.g., knowing that the number is likely a round number), then we would expect the missing digit to be '0' with high probability. However, if we have no specific knowledge about the nature of the number and prefer to make a more conservative assumption, the uniform prior may be more suitable.

In real-world scenarios, the choice of prior assumptions often depends on domain knowledge, available data, and the specific problem at hand. It is important to carefully consider and justify the prior assumptions used in information-theoretic and statistical analyses.

14.3.2. Example: Classification

In this example, we will determine the optimal splits for a decision tree² based on the information gained at each node in the tree.

```
using DataFrames

employed = [true, false, true, true, false, false, true]
good_credit = [true, true, false, true, false, false, true]
default = [true, false, true, true, true, false, true]
default_data = DataFrame(; employed, good_credit, default)
```

The entropy of the default rate data is, per Equation 14.2:

```
H0 = let
    p_default = sum(default_data.default) / nrow(default_data)
    p_good = 1 - p_default
    p_default * log2(1 / p_default) + p_good * log2(1 / p_good)
```

²A decision tree is a classification algorithm which attempts to optimally classify an output based on if/else type branches on the input variables.

14. Statistical Inference and Information Theory

Table 14.3.: Fictional data regarding loan attributes and whether or not a loan defaulted before its maturity.

| | employed | good_credit | default |
|---|----------|-------------|---------|
| | Bool | Bool | Bool |
| 1 | 1 | 1 | 1 |
| 2 | 0 | 1 | 0 |
| 3 | 1 | 0 | 1 |
| 4 | 1 | 1 | 1 |
| 5 | 1 | 0 | 1 |
| 6 | 0 | 0 | 1 |
| 7 | 0 | 0 | 0 |
| 8 | 1 | 1 | 1 |

| end

0.8112781244591328

Our goal is to determine which attribute (`employed` or `good_credit`) to use as the first split in the decision tree. Intuitively, we are looking for the most important factor in predicting default rates. We will quantitatively evaluate this by calculating the information gain, which is the difference in entropy between the prior node and the candidate node. Whichever criterion gains us the most information is the preferred attribute to create a decision split.

In our case we start with H_0 as calculated above for the output variable `default` and calculate the difference in entropy between it and the average entropy of the data if we split on that node. The name for this is the **information gain**, $IG(inputs, attributes)$:

$$IG(T, a) = H(T) - H(T|a)$$

In words, the information gain is simply the difference in entropy before and after learning the value of an outcome a . We will illustrate that by determining the first branch in the decision tree.

Let's first consider splitting the tree based on the `employed` status. We will calculate the entropy of each subset: with employment and without employment.

If we split the data based on being employed, we'd get two sub-datasets:

| df_employed = filter(:employed => ==(true), default_data)

| | employed | good_credit | default |
|---|----------|-------------|---------|
| | Bool | Bool | Bool |
| 1 | 1 | 1 | 1 |
| 2 | 1 | 0 | 1 |
| 3 | 1 | 1 | 1 |
| 4 | 1 | 0 | 1 |
| 5 | 1 | 1 | 1 |

and

```
df_unemployed = filter(:employed => ==(false), default_data)
```

| | employed | good_credit | default |
|---|----------|-------------|---------|
| | Bool | Bool | Bool |
| 1 | 0 | 1 | 0 |
| 2 | 0 | 0 | 1 |
| 3 | 0 | 0 | 0 |

Let's call its entropy H_{employed} , which should be zero because there is no variability in the default outcome for this subset.

```
H_employed = let
    p_default = sum(df_employed.default) / nrow(df_employed)
    p_good = 1 - p_default
    # p_default * log2(1 / p_default) + p_good * log2(1 / p_good)
    p_default * log2(1 / p_default) + 0
end
```

①

- ① In the case of $p_i = 0$ the value of h (the second term in the sum above) is taken to be 0, which is consistent with the $\lim_{p \rightarrow 0^+} p \log(p) = 0$.

0.0

And the corresponding candidate leaf is $H_{\text{unemployed}}$:

```
H_unemployed = let
    p_default = sum(df_unemployed.default) / nrow(df_unemployed)
    p_good = 1 - p_default
    p_default * log2(1 / p_default) + p_good * log2(1 / p_good)
end
```

0.9182958340544893

To balance these two results, we weight them according to the amount of data (number of observations) that would fall into each leaf:

```
H1_employment = let
    p_emp = nrow(df_employed) / nrow(default_data)
    p_unemp = 1 - p_emp
    p_emp * H_employed + p_unemp * H_unemployed
end
```

0.34436093777043353

The information gain for splitting the tree using employment status is the difference between the root entropy and the entropy of the employment split:

```
IG_employment = H0 - H1_employment
```

0.4669171866886993

14. Statistical Inference and Information Theory

We could repeat the analysis to determine the information gain if we were to split the tree based on having good credit. However, given that there are only two attributes we can already conclude that `employed` is a better attribute to split the data on. This is because the information gain of `IG_employment` (0.467) is the majority of the overall entropy H_0 (0.811). Entropy is non-negative and information gain is bounded by the original entropy, therefore no other attribute could have greater information gain. This also matches our intuition when looking at Table 14.3 as the eye can spot a higher correlation between `employed` and `default` than `good_credit` and `default`.

The above example demonstrates how we can use information theory to create more optimal inferences on data.

14.3.3. Maximum Entropy Distributions

Why is information theory a useful concept? Many financial models are statistical in nature and concepts of randomness and entropy are foundational. For example, when trying to estimate parameter distributions or assume a distribution for a random process you can lean on information theory to use the most conservative choice: the distribution with the highest entropy given known constraints. These distributions are referred to as **maximum entropy (maxent) distributions**.

In many real-world problems, we seek a distribution that is “least biased” or “most conservative” given certain known information (such as a mean or a range). Maximum entropy distributions accomplish this by spreading out probability as widely as possible under the constraints we know to be true (e.g., average value, bounded domain). They make no additional assumptions beyond those constraints, thereby avoiding unwarranted specificity.

By using a maxent distribution, we effectively acknowledge that everything else about the system’s behavior is unknown and should remain as “random” (unconstrained) as possible. This is a powerful principle because it aligns well with real-world modeling scenarios where we might know just a few key facts—like a process’s average rate or finite variance—but have no strong reason to assume anything else about its structure.

Many of the most common probability distributions (Normal, Exponential, Gamma, etc.) can be derived by applying the maximum entropy principle under simple, natural constraints:

- **Normal distribution** when the mean and variance are finite but otherwise unconstrained.
- **Exponential distribution** when we know only that the mean is positive, with outcomes over $[0, \infty)$.
- **Uniform distribution** when outcomes are bounded within a certain interval, and we have no further information about how likely each point is.

Additional maxent distributions and associated constraints are listed in Table 14.4. Those distributions arise again and again in nature because of the second law of thermodynamics - nature likes to have constantly increasing entropy and therefore it should be no surprise that (random) processes that maximize entropy pop up all over the place. The **second law of thermodynamics** in physics is an analogy: it states that a closed system tends to move toward higher entropy states. Similarly, in purely probabilistic settings, when few constraints are imposed, the system’s “natural” distribution tends to be the one that maximizes entropy.

Maxent distributions have practical modeling use:

- **Conservative Assumptions:** Using a maxent distribution guards against over-fitting or adding hidden assumptions. It essentially says: “Given only these constraints, let the data spread out in the most uniform (least structured) way consistent with what I know.”

- **Simplicity and Clarity:** It's often easier to justify a maxent model to stakeholders or regulators. If you only know a mean and a variance, a Normal distribution may be the least-biased fit. If you only know a mean and posit that values must be positive, the Exponential distribution is your maxent choice.
- **Built-In Neutrality:** In financial or actuarial contexts, adopting a maxent framework can prevent overly optimistic or pessimistic models. By sticking to the distribution with the fewest assumptions, the risk analysis remains transparent and more robust to model mis-specification.

Some discussion of maximum entropy distributions in the context of risk assessment is available in Duracz (2009).

Table 14.4.: Maximum Entropy Distributions and the conditions under which they are applicable.
For example, if you know that a probability must be continuous and have a positive mean (and is non-negative), then the MED is the Exponential Distribution.

| Constraint | Discrete Distribution | Continuous Distribution |
|--|-----------------------|-------------------------|
| Bounded range | Uniform (discrete) | Uniform (continuous) |
| Bounded range (0 to 1) with information about the mean or variance | | Beta |
| Fixed number of independent binary trials with known mean | Binomial | |
| Mean is finite and positive | Geometric | Exponential |
| Mean and mean of logarithm are finite, range is > zero | | Gamma |
| Mean and Variance are finite | | Gaussian (Normal) |
| Positive and equal mean and variance | Poisson | |

As an example, let's look at processes that behave like the Gaussian (Normal) distribution.

14.3.3.1. Processes that give rise to certain distributions

A random walk can be viewed as the cumulative impact of nudges pushing in opposite directions. This behavior culminates in the random, terminal position being able to be described by a Gaussian distribution. The center of a Gaussian distribution is “thick” because there are many more ways for the cumulative total nudges to mostly cancel out, while it's increasingly rare to end up further and further from the starting point (mean). The distribution then spreads out as flat (randomly) as it can while still maintaining the constraint of having a given, finite variance. Any other continuous distribution that has the same mean and variance has lower entropy than the Gaussian.

14. Statistical Inference and Information Theory

Table 14.5.: Underlying processes create typical probability distributions. That there is significant overlap with the distributions in Section 14.3.3 is not a coincidence.

| Process | Distribution of Data | Examples |
|--|----------------------|--|
| Many <i>additive</i> pluses and minuses that move an outcome in one dimension | Normal | Sum of many dice rolls, errors in measurements, sample means (Central Limit Theorem) |
| Many <i>multiplicative</i> pluses and minuses that move an outcome in one dimension | Log-normal | Incomes, sizes of cities, stock prices |
| Waiting times between independent events occurring at a constant average rate | Exponential | Time between radioactive decay events, customer arrivals |
| Discrete trials each with the same probability of success, counting the number of successes | Binomial | Coin flips, defective items in a batch |
| Discrete trials each with the same probability of success, counting the number of trials until the first success | Geometric | Number of job applications until getting hired |
| Continuous trials each with the same probability of success, measuring the time until the first success | Exponential | Time until a component fails, time until a sales call results in a sale |
| Waiting time until the r-th event occurs in a Poisson process | Gamma | Time until the 3rd customer arrives, time until the 5th defect occurs |

💡 Probability Distributions

There are a *lot* of specialized distributions. There are lists of distributions you can find online or in references such as Leemis and McQueston (2008) which has a full-page network diagram of the relationships.

The information-theoretic and Bayesian perspective on it is to eschew memorization of a bunch of special cases and statistical tests. If you pull up the aforementioned diagram in Leemis and McQueston (2008), you can see just a handful of distributions that have the most central roles in the universe of distributions. Many distributions are simply transformations, limiting instances, or otherwise special cases of a more fundamental distribution. Instead of trying to memorize a bunch of probability distributions, it's better to think critically about:

1. The fundamental processes that give rise to the randomness we are interested in modeling.
2. Transformations of the data to make it nicer to work with, such as translations, scaling, or other non-destructive changes.

Then when you encounter an unusual dataset, you don't need to comb the depths of Wikipedia to find the perfect probability distribution for that situation.

14.3.3.2. Additive and Multiplicative Processes

Table 14.5 describes some examples, let us discuss further what it means to have a process that arises via an additive vs multiplicative effect³. Additive processes result in a normal distribution while multiplicative processes give rise to a log-normal distribution⁴.

An outcome is additive and results in a Normal distribution if it's the sum or difference of multiple independent processes, as described by the **Central Limit Theorem**. Examples include:

- Rolling multiple dice and taking their sum.
- A random walk along the natural numbers wherein with equal probability you take a step left or right.
- Calculating the arithmetic mean of samples (the Central Limit Theorem).

However, many processes are multiplicative in nature. For example, the population density of cities is distributed in a log-normal fashion. If we think about the factors that contribute to choice of place to live, we can see how these factors multiply: an attractive city might make someone 10% more likely to move, a city with water features 15% more likely, high crime 30% less likely, etc. These forces combine in a multiplicative way in the generative process of deciding where to move. In finance, many price processes are considered multiplicative.

Tip 1: Logarithms

The logarithm of a geometric process transforms the outcomes into “log-space”. The information is the same, but is often a more convenient form for the analysis. That is, if:

$$Y = x_1 \times x_2 \times \dots \times x_i$$

Then,

$$\log(Y) = \log(x_1) + \log(x_2) + \dots + \log(x_i)$$

This is effectively the transformation that gives rise to the Normal versus Log-Normal distribution.

In the context of computational thinking:

First, we should think about how to transform data or modeling outcomes into a more convenient format. The log transform doesn't eliminate any information but may map the information into a shape that is easier for an optimizer or Monte Carlo simulation to explore.

Second, per Chapter 5, floating point math is a *lossy* transformation of real numbers into a digital computer representation. Some information (in the literal Shannon information sense) is lost when computing and this tends to be worst with very small real numbers, such as those we encounter frequently in probabilities and likelihoods. Logarithms map very small numbers into negative numbers that don't encounter the same degree of truncation error that tiny numbers do.

Third, modern CPUs are generally much faster at adding or subtracting numbers than multiplying or dividing. Therefore working with the logarithm of processes may be computationally faster than the direct process itself.

³Multiplicative processes are often referred to as “geometric”, as in “geometric Brownian motion” or “geometric mean”. Additive processes are sometimes referred to as “arithmetic”. The root of this confusing terminology appears to be due to the fact that series involving repeated multiplication were solved via geometric (triangles, angles, etc.) methods while those using sums and differences were solved via arithmetic.

⁴See Tip 1.

14.4. Bayes' Rule

With some of the foundational concepts laid down, we now turn to the perpetual challenge of attempting to make inferences and predictions given a set of data. We covered basic information theory, probability distributions, log transformations, and random processes because modern statistical analysis relies heavily on those concepts and techniques. We'll introduce Bayes' Rule, but analysis beyond trivial applications will typically encounter challenges posed by those ideas.

The remainder of this chapter will re-introduce Bayes' Rule and then build up modeling applications that illustrate some core concepts of applying Bayes' Rule to complex, data-intensive problems.

14.4.1. Bayes' Rule Formula

The minister and statistician Thomas Bayes derived a relationship of conditional probabilities that we today know as **Bayes' Rule**. Laplace⁵ furthered the notion, and developed the modern formulation, commonly written as:

$$P(H|D) = \frac{P(D|H) \times P(H)}{P(D)}$$

The components of this are:

- $P(H | D)$ is the conditional probability of event H occurring given that D is true.
- $P(D | H)$ is the conditional probability of event D occurring given that H is true.
- $P(H)$ is the prior probability of event H .
- $P(D)$ is the prior probability of event D .

If we take the following:

- D is the available data
- H is our hypothesis

Then we can draw conclusions about the probability of a hypothesis being true given the observed data. When thought about this way, Bayes' rule is often described as:

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{evidence}}$$

This is a very useful framework, which we'll return to more completely in Section 14.5. First, let's look at combining information theory and Bayes' rule in an applied example.

⁵Laplace actually deserves most of the credit, as it was he who formalized the modern notion of Bayes' rule and cemented the mathematical formulation. Bayes just described it first, in a way that actually had almost no direct impact on math or science. See "The Theory That Would Not Die".

14.4.2. Model Selection via Likelihoods

Let's say that we have competing hypotheses about a data generating process, such as: "given a set of data representing risk outcomes, what distribution best fits the data"? We will not be able to determine an *absolute* probability of a model given the data, but amazingly we can determine *relative* probability of models given the data. This is powerful because often one of the most difficult modeling tasks is to select a model formulation - and Bayes gives us a powerful tool to help choose.

We can compare these models using Bayes' rule by observing the following: Suppose we have two models, H_1 and H_2 , and we want to compare their likelihoods given the observed data, D. We can use Bayes' rule to calculate the posterior probability of each model:

$$P(H_1|D) = \frac{P(D|H_1) \times P(H_1)}{P(D)}$$

$$P(H_2|D) = \frac{P(D|H_2) \times P(H_2)}{P(D)}$$

Where:

- $P(H_1|D)$ and $P(H_2|D)$ are the posterior probabilities of models H_1 and H_2 , respectively, given the data D .
- $P(D|H_1)$ and $P(D|H_2)$ are the likelihoods of the data D under models H_1 and H_2 , respectively.
- $P(H_1)$ and $P(H_2)$ are the prior probabilities of models H_1 and H_2 , respectively.
- $P(D)$ is the marginal likelihood of the data, which serves as a normalizing constant.

To compare the two models, we can calculate the ratio of their posterior probabilities, known as the **posterior odds ratio**:

$$\frac{P(H_1|D)}{P(H_2|D)}$$

Substituting the expressions for the posterior probabilities from Bayes' rule, we get:

$$\frac{P(H_1|D)}{P(H_2|D)} = \frac{P(D|H_1) \times P(H_1)}{P(D|H_2) \times P(H_2)}$$

The marginal likelihood $P(D)$ cancels out since it appears in both the numerator and denominator. The **Bayes factor** (BF) is defined as the ratio of the marginal likelihoods (the likelihood ratio):

$$BF = \frac{P(D|H_1)}{P(D|H_2)}$$

The posterior odds ratio equals the Bayes factor multiplied by the prior odds ratio. If we assume equal prior probabilities for the models, i.e., $P(H_1) = P(H_2)$, then the posterior odds ratio simplifies to the Bayes factor.

The Bayes factor then is a statement about the relative probability of two competing models for the given data. We can interpret the results as:

- If $BF > 1$, the data favor model H_1 over model H_2 .
- If $BF < 1$, the data favor model H_2 over model H_1 .
- If $BF = 1$, the data do not provide evidence in favor of either model.

14. Statistical Inference and Information Theory

In practice, the likelihoods $P(D|H_1)$ and $P(D|H_2)$ are often calculated using the probability density or mass functions of the models, evaluated at the observed data points. The prior probabilities $P(H_1)$ and $P(H_2)$ can be assigned based on prior knowledge or assumptions about the models. By comparing the likelihoods of the models using the Bayes factor, we can quantify the relative support for each model given the observed data, while taking into account the prior probabilities of the models.

Another way of interpreting this is the evaluation of which model has the higher likelihood given the data.

⚠ Null Hypothesis Statistical Test

Null Hypothesis Statistical Tests (NHST) is the idea of trying to statistically support an alternative hypothesis over a null hypothesis. The support in favor of alternative versus the null is reported via some statistical power, such as the **p-value** (the probability that the test result is as, or more extreme, than the value computed). The idea is that there's some objective way to push science towards greater truths, and NHST was seen as a methodology that avoided the subjectivity of the Bayesian approach. However, while pure in intention, the NHST choices of both null hypothesis and model contain significant amounts of subjectivity! There is subjectivity in the null hypothesis, data collection methodologies, study design, handling of missing data, choice of data *not* to include, which statistical tests to perform, and interpretation of relationships.

We might as well call the null hypothesis a prior and stop trying to disprove it absolutely. Instead: focus on model comparison, model structure, and posterior probabilities of the competing theories.

Over 100 statistical tests have been developed in service of NHST (Lewis 2013), but it's widely viewed now that a focus on NHST has led to *worse* science due to a multitude of factors, such as:

- “P-hacking” or trying to find subsets of data which can (often only by chance) support rejecting some null.
- Cognitive anchoring to the importance of a p-value of 0.05 or less – why choose that number versus 0.01 or 0.001 or 0.49?
- Bias in research processes where one may stop data collection or experimentation after achieving a favorable test result.
- Inappropriate application of the myriad of statistical tests.
- Focus on p-values rather than effects that simply matter more or have greater effect.
 - For example, which is of more interest to doctors? A study indicating a 1 in a billion chance of serious side effect, with p-value of 0.0001 or a study indicating a 1 in 3 chance with p-value 0.06? Many journals would only publish the former study, even though the latter study intuitively suggests a potentially more risky drug.
- Difficulty to determine *causal* relationships.

The authors of this book recommend against basic NHST and memorization of statistical tests in favor of principled Bayesian approaches. For the actuarial readers, NHST is analogous to traditional credibility methods (of which the authors also prefer more modern statistical approaches).

14.4.2.1. Example: Rainfall Risk Model Comparison

The example we'll look at relates to the annual rainfall totals for a specific location in California⁶, which could be useful for insuring flood risk or determining the value of a catastrophe bond. Acknowledging that we are attempting to create a geocentric model⁷ instead of a scientifically accurate weather model, we narrow the problem to finding a probability distribution that matches the historical rainfall totals.

Our goal is to recommend a model that best fits the data and justify that recommendation quantitatively. Before even looking at the data, Table 14.6 shows three competing models based on thinking about the real-world outcome we are trying to model. These three are chosen for the increasingly sophisticated thought process that might lead the modeler to recommend them - but which is supportable by the statistics?

Table 14.6.: Three alternative hypothesis about the distribution of annual rainfall totals.

| Hypothesis | Process | Possible Rationale |
|------------|--|---|
| H_1 | A Normal (Gaussian) distribution | The sum of independent rainstorms creates annual rainfall totals that are normally distributed |
| H_2 | A LogNormal distribution | Since it's normal-ish, but skewed and can't be negative |
| H_3 | A Gamma distribution on log-transformed rainfall | Combining the Gamma rationale (sum of exponential events) with the log-transform rationale (multiplicative effects, positive-only data) |

i Note

The distribution we derived for H_3 —fitting a Gamma to log-transformed data—is known in the hydrology literature as the “Log-Pearson Type III” distribution. It is actually recommended by the US Army Corps of Engineers for flood frequency analysis.

We arrived at this same model through first-principles reasoning about the probabilistic processes, without needing to memorize specialty distributions. This illustrates how thinking critically about the underlying mechanics can lead to the same conclusions as domain-specific expertise.

Here's the data:

```
rain = [
    39.51, 42.65, 44.09, 41.92, 28.42, 58.65, 30.18, 64.4, 29.02,
    37.00, 32.17, 36.37, 47.55, 27.71, 58.26, 36.55, 49.57, 39.84,
    82.22, 47.58, 51.18, 32.28, 52.48, 65.24, 51.12, 25.03, 23.27,
    26.11, 47.3, 31.8, 61.45, 94.95, 34.8, 49.53, 28.65, 35.3, 34.8,
    27.45, 20.7, 36.99, 60.54, 22.5, 64.85, 43.1, 37.55, 82.05, 27.9,
    36.55, 28.7, 29.25, 42.32, 31.93, 41.8, 55.9, 20.65, 29.28, 18.4,
    39.31, 20.36, 22.73, 12.75, 23.35, 29.59, 44.47, 20.06, 46.48,
```

⁶<https://data.ca.gov/dataset/annual-precipitation-data-for-northern-california-1944-current>

⁷See Section 3.5.

14. Statistical Inference and Information Theory

```
    13.46, 9.34, 16.51, 48.24  
];
```

Plotted, we see some of the characteristics that align with our prior assumptions and knowledge about the system itself, such as: the data being constrained to positive values and a skew towards having some extreme weather years with lots of rainfall.

```
using CairoMakie  
hist(rain)
```

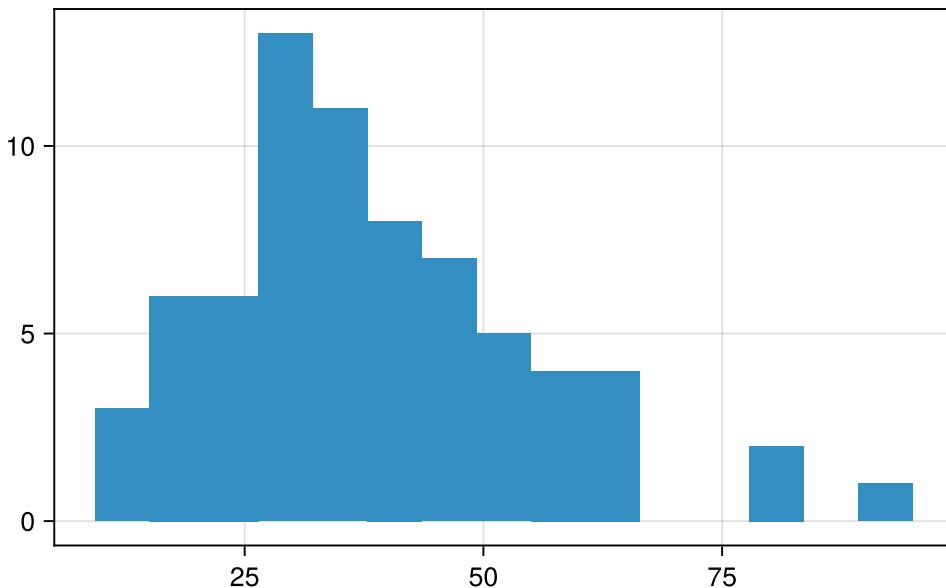


Figure 14.1.: Annual rainfall totals for a specific location in California.

We will show the likelihood of the three models after deriving the **maximum likelihood (MLE)**, which is simply finding the parameters that maximize the calculated likelihood. In general, this can be accomplished by an optimization routine, but here we will just use the functions built into Distributions.jl:

```
using StatsBase  
using Distributions  
  
# H1: Normal  
h1_dist = fit_mle(Normal, rain)  
  
# H2: LogNormal - parameters are from fitting a Normal to the log of the data  
h2_params = fit_mle(Normal, log.(rain))  
h2_dist = LogNormal(params(h2_params)...)  
  
# H3: Log-Pearson Type III - Gamma fitted to log-transformed data  
h3_log_dist = fit_mle(Gamma, log.(rain))
```

```

@show h1_dist
@show h2_dist
@show h3_log_dist;

h1_dist = Distributions.Normal{Float64}( $\mu=38.91442857142857$ ,  $\sigma=16.643603630714306$ )
h2_dist = Distributions.LogNormal{Float64}( $\mu=3.5690550009062663$ ,
    ↵  $\sigma=0.44148379736539156$ )
h3_log_dist = Distributions.Gamma{Float64}( $\alpha=61.58531301458412$ ,
    ↵  $\theta=0.05795302201453571$ )

```

Let's look at the likelihoods by applying the maximum likelihood distribution to the observed data. For the practical reasons described in Tip 1, we will compare the log-likelihoods to maintain convention with what you'd likely see or deal with in practice. Taking the log of the likelihood does not change the ranking of the likelihoods.

```

h1_loglik = sum(log.(pdf.(h1_dist, rain)))
h2_loglik = sum(log.(pdf.(h2_dist, rain)))

# For H3, we fit on the log scale, so we need the Jacobian adjustment
# pdf_Y(y) = pdf_X(log(y)) * (1/y), so log(pdf_Y(y)) = log(pdf_X(log(y))) - log(y)
h3_loglik = sum(logpdf.(h3_log_dist, log.(rain)) .- log.(rain))

@show h1_loglik
@show h2_loglik
@show h3_loglik

h1_loglik = -296.16751566478115
h2_loglik = -291.9265702808178
h3_loglik = -293.6253681269266

-293.6253681269266

```

Note

Instead of `sum(log.(pdf.(distribution,data)))`, we could call `loglikelihood(distribution,data)`. We show the former for the explicitness of the calculation for pedagogy. In real use, `loglikelihood` should be preferred because it has additional logic to improve numerical stability for floating point operations.

The results indicate that the LogNormal and the log-transformed Gamma model for rainfall distribution are superior to the Normal model, consistent with the visual inspection of the quantiles in Figure 14.2. We reach that conclusion by noting how much more likely the latter two are, as the likelihoods of -291.93 and -293.63 are higher (less negative) than -296.17⁸. Higher (less negative) log-likelihood values indicate a better fit.

```

let x = rain
    # Use the fitted distributions from the previous step
    range = 1:0.1:100
    fig, ax, _ = lines(
        range,

```

⁸The values are negative because we are taking the logarithm of a number less than 1. The likelihoods are less than 1 because the likelihood is the joint (multiplicative) probability of observing each of the individual outcomes.

14. Statistical Inference and Information Theory

```

cdf.(h1_dist, range),
label="Normal",
axis=(xgridvisible=false, ygridvisible=false,
)
lines!(ax, range, cdf.(h2_dist, range), label="LogNormal")
# For H3, CDF of Y = exp(X) where X ~ Gamma is CDF_Gamma(log(y))
lines!(ax, range, cdf.(h3_log_dist, log.(range)), label="Gamma (log)")

# Plot empirical CDF of the data
lines!(
    quantile.(Ref(x), 0.01:0.01:0.99),
    0.01:0.01:0.99,
    label="Data",
    color=(:black, 0.6),
    linewidth=3
)
fig[1, 2] = Legend(fig, ax, "Model", framevisible=false)
fig
end

```

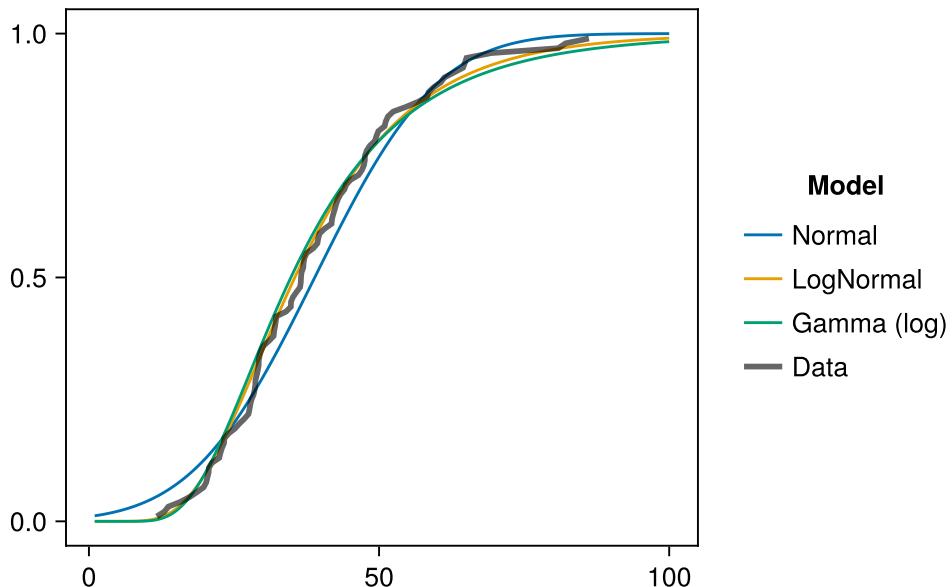


Figure 14.2.: Empirical and fitted CDFs for rainfall data.

We evaluated the likelihood at a single point estimate of the parameters, but a true posterior probability of the parameters of the distributions will be represented by a *distribution* rather than a point. The rest of chapter will describe how to express the posterior probabilities of the parameters for H_1 , H_2 , and H_3 using Bayesian statistical methods.

14.5. Modern Bayesian Statistics

14.5.1. Background

Bayesian statistics is generally *not* taught in undergraduate statistics. Bayes' rule is introduced, basic probability exercises are assigned, and then statistics moves on to a curriculum of regression and NHSTs (of the Frequentist school). Why is the applied practice of statistics currently gravitating towards Bayesian approaches? There are both philosophical and practical reasons why.

14.5.1.1. Philosophical Motivations

Philosophically, one of the main reasons why Bayesian thinking is appealing is its ability to provide straightforward interpretations of statistical conclusions.

For example, when estimating an unknown quantity, a Bayesian probability interval can be directly understood as having a high probability of containing that quantity. In contrast, a Frequentist confidence interval is typically interpreted only in the context of a series of similar inferences that could be made in repeated practice. In recent years, there has been a growing emphasis on interval estimation rather than hypothesis testing in applied statistics. This shift has strengthened the Bayesian perspective since it is likely that many users of standard confidence intervals intuitively interpret them in a manner consistent with Bayesian thinking.

i “The fallacy of placing confidence in confidence intervals”

“The fallacy of placing confidence in confidence intervals” is the title of an article (Morey et al. 2016) describing the issues with confidence intervals, with one of the primary issues being what they call the “Fundamental Confidence Fallacy,” or FCF.

The FCF is the belief that if you have a $X\%$ confidence interval, that the probability that the true value of interest lies within that interval is $X\%$. This is false, and an abbreviated explanation of why is as follows:

- An $X\%$ confidence interval refers to a procedure that results in a range R wherein $X\%$ of the time the true value of interest lies within R .
- However, once the data is observed we can know with higher probability than X whether the true value lies within that range.

Here's an example taken from the article referenced above: we wish to estimate the mean of a continuous random variable. We observe two data points y_1 and y_2 . If $y_1 < y_2$ then we say our interval is $(-\infty, \infty)$ otherwise the interval is empty. Our credibility procedure creates an interval for which 50% of the time it contains the true value. However, once we've observed the data and created the interval, then “post-data” we can tell with certainty whether our interval contains the variable of interest. We can only say “pre-data”, or pre-observations, that the credible interval is really $X\%$ probable to contain the right value. Similar (and other) issues arise with more “real world” examples of confidence intervals.

In contrast, the Bayesian procedure of estimating an interval using the posterior distribution of the data *can* be interpreted as an interval for which you can say that you believe the true value is contained within the interval. Typically, this is referred to as a *credible interval* to distinguish from a *confidence interval*.

Another meaningful way to understand the contrast between Bayesian and Frequentist approaches is through the lens of decision theory, specifically how each view treats the concept of randomness.

14. Statistical Inference and Information Theory

This perspective pertains to whether you regard the data as being random or the parameters as being random.

Frequentist statistics treats parameters as fixed and unknown, and the data as random — that data you collect is but one realization of an infinitely repeatable random process. Consequently, Frequentist procedures, like hypothesis testing or confidence intervals, are generally based on the idea of long-run frequency or repeatable sampling.

Conversely, Bayesian statistics turns this on its head by treating the data as fixed — after all, once you've collected your data, it's no longer random but a fixed observed quantity. Parameters, which are unknown, are treated as random variables. The Bayesian approach then allows us to use probability to quantify our uncertainty about these parameters.

The Bayesian approach tends to align more closely with our intuitive way of reasoning about problems. Often, you are given specific data and you want to understand what that particular set of data tells you about the world. You're likely less interested in what might happen if you had infinite data, but rather in drawing the best conclusions you can from the data you do have.

None of this diminishes the value of Frequentist procedures. Regulatory capital models, credit scorecards, and actuarial reserving processes still lean heavily on generalized linear models, credibility theory, and classical hypothesis tests because they are simple to communicate and often computationally cheaper. In practice most teams blend both mindsets: use Frequentist estimators when they answer the question clearly, and reach for Bayesian workflows when parameter uncertainty, small data, or hierarchical structures matter.

14.5.1.2. Practical Motivations

Practically, recent advances in computational power, algorithm development, and open-source libraries have enabled practitioners to adopt the Bayesian workflow.

For most real-world problems, deriving the posterior distribution is analytically intractable and computational methods must be used. Advances in raw computing power only in the 1990s made non-trivial Bayesian analysis possible, and recent advances in algorithms have made the computations more efficient. For example, one of the most popular algorithms, NUTS, was only published in the 2010s.

Many problems require the use of compute clusters to manage runtime, but if there is any place to invest in understanding posterior probability distributions, it's financial companies trying to manage risk!

Open-source libraries such as Turing.jl, PyMC, and Stan provide access to the core routines in an accessible interface. To get the most out of these tools requires the mindset of computational thinking described in this book - understanding model complexity, model transformations and structure, data types and program organization, etc.

14.5.1.3. Advantages of the Bayesian Approach

The main advantages of this approach over traditional actuarial techniques are:

1. **Focus on distributions rather than point estimates of the posterior's mean or mode.** We are often interested in the distribution of the parameters and a focus on a single parameter estimate will underestimate the risk distribution.
2. **Model flexibility.** A Bayesian model can be as simple as an ordinary linear regression, but as complex as modeling full insurance mechanics.

3. **Simpler mental model.** Fundamentally, Bayes' theorem could be distilled down to an approach where you count the ways that things could occur and update the probabilities accordingly.
4. **Explicit Assumptions:** Enumerating the random variables in your model and explicitly parameterizing prior assumptions avoids ambiguity of the assumptions inside the statistical model.

14.5.1.4. Challenges with the Bayesian Approach

With the Bayesian approach, there are a handful of things that are challenging. Many of the listed items are not unique to the Bayesian approach, but there are different facets of the issues that arise.

1. **Model Construction.** One must be thoughtful about the model and how variables interact. However, with the flexibility of modeling, you can apply (actuarial) science to make better models!
2. **Model Diagnostics.** Instead of R^2 values, there are unique diagnostics that one must monitor to ensure that the posterior sampling worked as intended.
3. **Model Complexity and Size of Data.** The sampling algorithms are computationally intensive - as the amount of data grows and model complexity grows, the runtime demands cluster computing.
4. **Model Representation.** The statistical derivation of the posterior can only reflect the complexity of the world as defined by your model. A Bayesian model won't automatically infer all possible real-world relationships and constraints.

Subjectivity of the Priors?

There are two ways one might react to subjectivity in a Bayesian context: It's a feature that should be embraced or it's a flaw that should be avoided.

14.5.1.5. Subjectivity as a Feature

A Bayesian approach to defining a statistical model is an approach that allows for explicitly incorporating professional judgment. Encoding assumptions into a Bayesian model forces the actuary to be explicit about otherwise fuzzy predilections. The explicit assumption is also more amenable to productive debate about its merits and biases than an implicit judgmental override.

14.5.1.6. Subjectivity as a Flaw

Subjectivity is inherent in all useful statistical methods. Subjectivity in traditional approaches includes how the data was collected, which hypothesis to test, what significance levels to use, and assumptions about the data-generating processes.

In fact, the “objective” approach to null hypothesis testing is so prone to abuse and misinterpretation that in 2016, the American Statistical Association issued a statement intended to steer statistical analysis into a “post $p < 0.05$ era.” That “ $p < 0.05$ ” approach is embedded in most traditional approaches to actuarial credibility⁹ and therefore should be similarly reconsidered.

⁹Note that the approach discussed here is much more encompassing than the Bühlmann-Straub Bayesian approach described in the actuarial literature.

14. Statistical Inference and Information Theory

14.5.2. Implications for Financial Modeling

Like Bayes' Formula itself, another aspect of financial literature that is taught but often glossed over in practice is the difference between process risk (volatility), parameter risk, and model formulation risk. When performing analysis that relies on stochastic results, in practice typically only process/volatility risk is assessed.

Bayesian statistics provides the tools to help financial modelers address parameter risk and model formulation. The posterior distribution of parameters derived is consistent with the observed data and modeled relationships. This posterior distribution of parameters can then be run as an additional dimension to the risk analysis.

Additionally, best practices include skepticism of the model construction itself, and testing different formulations of the modeled relationships and variable combinations to identify models which are best fit for purpose. Tools such as Information Criterion, posterior predictive checks, Bayes factors, and other statistical diagnostics can inform the actuary about trade-offs between different choices of model.

Bayesian Versus Machine Learning

Machine learning (ML) is *fully compatible* with Bayesian analysis - one can derive posterior distributions for the ML parameters like any other statistical model and the combination of approaches may be fruitful in practice.

However, to the extent that actuaries have leaned on ML approaches due to the shortcomings of traditional actuarial approaches, Bayesian modeling may provide an attractive alternative without resorting to notoriously finicky and difficult-to-explain ML models. The Bayesian framework provides an explainable model and offers several analytic extensions beyond the scope of this introductory chapter:

- Causal Modeling: Identifying not just correlated relationships, but causal ones, in contexts where a traditional designed experiment is unavailable.
- Bayes Action: Optimizing a parameter for, e.g., a CTE95 level instead of a parameter mean.
- Information Criterion: Principled techniques to compare model fit and complexity.
- Missing data: Mechanisms to handle the different kinds of missing data.
- Model averaging: Posteriors can be combined from different models to synthesize different approaches.
- Credible Intervals: A posterior representation around the likely range of values for parameters of interest.

14.5.3. Basics of Bayesian Modeling

A Bayesian statistical model has four main components to focus on:

1. **Prior** encoding assumptions about the random variables related to the problem at hand, before conditioning on the data.
2. A **Model** that defines how the random variables give rise to the observed outcome.
3. **Data** that we use to update our prior assumptions.
4. **Posterior** distributions of our random variables, conditioned on the observed data and our model

While this is simply stating Bayes' formula in words, it's also the blueprint for a workflow to implement more advanced Bayesian methods.

Having defined a prior assumption, selected a model, and collected our data, the computation of the posterior can be the most challenging. The workflow involves computationally sampling the posterior distribution, often using a technique called **Markov Chain Monte-Carlo** (MCMC). The result is a series of values that are sampled statistically from the posterior distribution. Introducing this process is the focus of the rest of this chapter.

14.5.4. Markov-Chain Monte Carlo

Computing the posterior distribution for most model parameters is analytically intractable. However, we can probabilistically sample from the posterior distribution and achieve an approximation of the posterior distribution. MCMC samplers, as they are called, do this by moving through the parameter space (the set of possible values for the parameters) in a special way. The statistical marvel is that they travel to different points *in proportion* to the posterior probability. It is a “Markov-Chain” because the probability of the next point’s location is influenced by the prior sampling point’s location.

14.5.4.1. Example: MCMC from Scratch

Here is a simple example demonstrated with one of the oldest MCMC algorithms, called Metropolis-Hastings. The general idea is this:

1. Start at an arbitrary point and make that the `current_state`.
2. Propose a new point which is the `current_state` plus some movement that comes from a random distribution, `proposal_dist`.
3. Calculate the likelihood ratio of the proposed versus current point (`acceptance_ratio` below).
4. Draw a random number - if that random number is less than the `acceptance_ratio`, then move to that new point. Otherwise do not move.
5. Repeat steps 2-4 until the distribution of points converges to a stable posterior distribution.

This gets us what we desire because the resulting distribution of samples has frequency that’s proportional to the posterior distribution.

We will try to find the posterior of an arbitrary set of normally distributed asset returns. We set the true, (unobserved in reality) values for μ and σ and then draw 250 generated observations:

```
# In reality, we don't observe the parameters
# we are interested in determining the values for.
σ = 0.15
μ = 0.1

n_observations = 250
return_dist = Normal(μ,σ)
returns = rand(return_dist,n_observations)

# plot the distribution of returns
μ_range = LinRange(-0.5, 0.5, 400)
σ_range = LinRange(0.0, 3.0, 400)

f = Figure()
ax1 = Axis(f[1,1],title="True Distribution of Returns")
ax2 = Axis(f[2,1],title="Simulated Outcomes", xlabel="Return")
```

14. Statistical Inference and Information Theory

```

plot!(ax1,return_dist)
vlines!(ax1,[μ],color=(:black,0.7))
text!(ax1,μ,0;text="mean ($μ)",rotation=pi/2)
hist!(ax2,returns)
vlines!(ax2,[mean(returns)],color=(:black,0.7))
text!(
    ax2,
    mean(returns),
    0;
    text="mean ($(round(mean(returns);digits=3)))",
    rotation=pi/2
)

linkxaxes!(ax1,ax2)

f

```

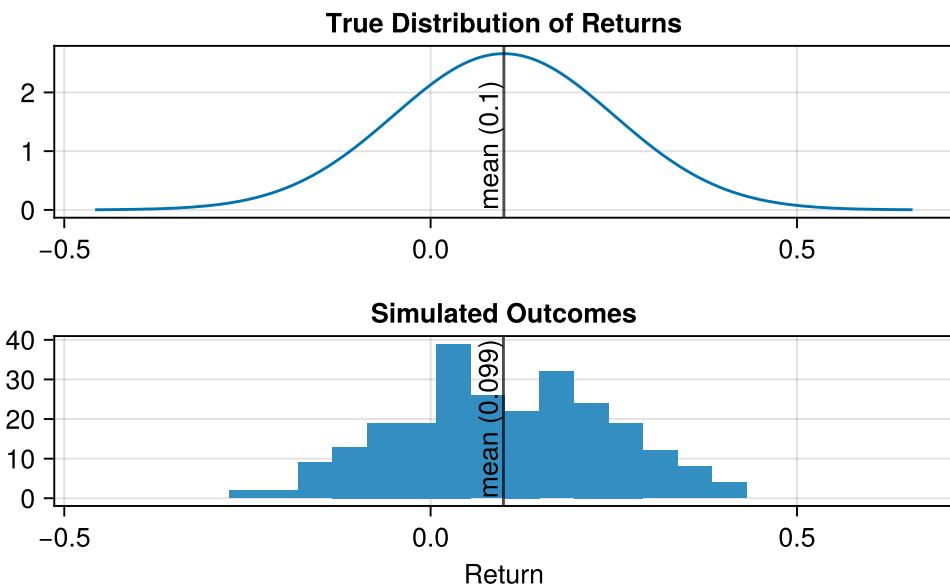


Figure 14.3.: The target probability densities which we will attempt to infer via MCMC.

Having generated sample data, we will next define a probability distribution for the random step that we take from the `current_point` on the Markov chain. We choose a 2D Gaussian for this, since the parameter space to explore is two-dimensional (μ and σ). The `proposal_std` controls how big of a movement is taken at each step.

```

# Define the proposal step distribution
proposal_std = 0.05
proposal_dist = Normal(0,proposal_std)

Distributions.Normal{Float64}(μ=0.0, σ=0.05)

```

We next define how many steps we want the chain to sample for, and implement the algorithm's main loop containing the logic in the steps above.

```

# MCMC parameters
num_samples = 5000
burn_in = 500

# Define priors
μ_prior = Normal(0, 0.25)
σ_prior = Gamma(0.5, 1.0)

# Initialize the Markov chain
μ_current, σ_current = 0.0, 0.25
current_prob = sum(logpdf(Normal(μ_current, σ_current), r) for r in returns)
    +
        logpdf(μ_prior, μ_current) +
        logpdf(σ_prior, σ_current)

# The 'chain' will store all samples, including the burn-in phase.
# We will discard the first 'burn_in' samples later during analysis.
chain = zeros(num_samples, 2)

count = 0

# MCMC sampling loop
while count < num_samples

    # Generate a new proposal
    ḩ, ḍ = μ_current + rand(proposal_dist), σ_current + rand(proposal_dist)
    if ḍ > 0

        # Calculate the acceptance ratio

        proposal_prob = sum(logpdf(Normal(ጀ, ḍ), r) for r in returns) +
                        logpdf(μ_prior, ጀ) + logpdf(σ_prior, ḍ)
        log_acceptance_ratio = proposal_prob - current_prob

        # Accept or reject the proposal
        if log(rand()) < log_acceptance_ratio
            μ_current, σ_current = Ḍ, ḍ
            current_prob = proposal_prob
        end

        # Store the current state as a sample
        count += 1
        chain[count, :] .= μ_current, σ_current
    else
        # skip because σ can't be negative
    end
end

chain

```

5000×2 Matrix{Float64}:

| | |
|-----|------|
| 0.0 | 0.25 |
|-----|------|

14. Statistical Inference and Information Theory

```
0.0      0.25
0.0      0.25
0.0      0.25
0.0      0.25
0.0      0.25
0.0856262 0.229152
0.0856262 0.229152
0.0856262 0.229152
0.146121  0.217932
:
0.102837  0.160074
0.102837  0.160074
0.102837  0.160074
0.102837  0.160074
0.102837  0.160074
0.102837  0.160074
0.102837  0.160074
0.102837  0.160074
0.102837  0.160074
0.102837  0.160074
```

The resulting chain contains a list of points that the algorithm has moved along during the sampling process. Note that there is a burn-in parameter. This is because we want the chain to iterate long enough to be effectively independent of both (1) the starting point for the sample, and (2) that different chains are effectively independent.

After having performed the sampling, we can now visualize the chain versus the `target_distribution`. A few things to note:

1. The red line indicates the “warm up” or “burn-in” phase and we do not consider that as part of the sampled chain because those values are too correlated with the arbitrary starting point.
2. The blue line indicates the path traveled by the Metropolis-Hastings algorithm. Long asides into low-probability regions are possible, but in general the path will traverse areas in proportion to the probability of interest.

```
# Plot the chain
let
    f = Figure()

    # μ lines
    ax1 = Axis(f[1, 1], ylabel="σ", xticklabelsvisible=false)

    # burn in lines
    scatterlines!(ax1, chain[1:burn_in,1],
                chain[1:burn_in,2],
                color=(:red,0.1),
                markercolor=(:red,0.1))

    # sampled lines
    scatterlines!(ax1, chain[burn_in+1:end,1],
                chain[burn_in+1:end,2],
                color=(:blue,0.1),
                markercolor=(:blue,0.1))

    # μ histogram
```

```

ax2 = Axis(f[2,1], xlabel=" $\mu$ ", yticklabelsvisible=false)
hist!(ax2, chain[burn_in+1:end,1], color=:blue)
linkxaxes!(ax1, ax2)

#  $\sigma$  histogram
ax3 = Axis(f[1,2], xticklabelsvisible=false)
hist!(ax3, chain[burn_in+1:end,2], color=:blue, direction=:x)
linkyaxes!(ax1, ax3)

f
end

```

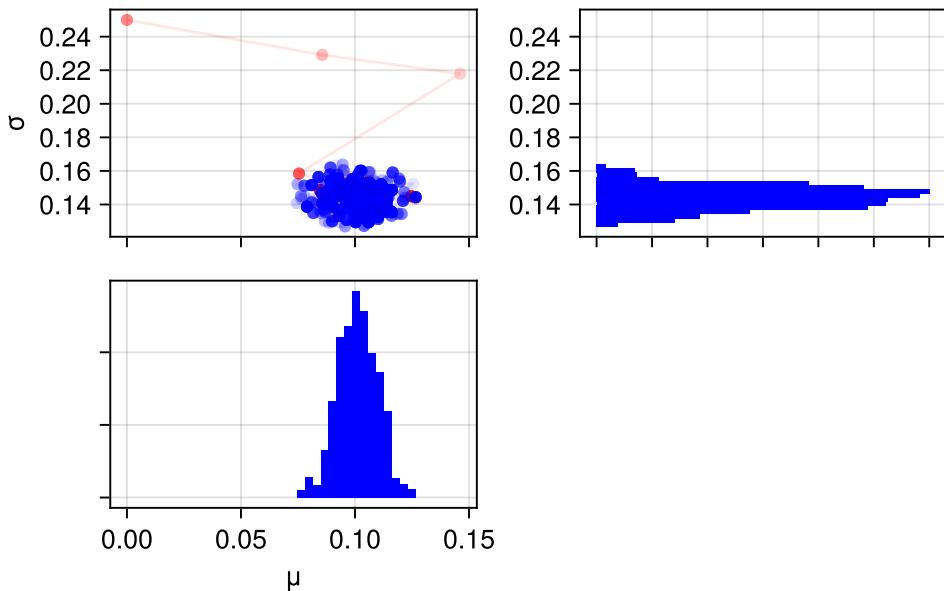


Figure 14.4.: The blue lines of the MCMC chain explore the posterior density of interest (after discarding the burn-in samples in red). Note that locations where the sampler remained longer (rejected more proposals) show up as darker points.

In this example, μ and σ are independent, but if there were a correlation (such as when μ were higher, σ were also higher) then the sampler would pick up on this, and we would see a skew in the plotted chain.

The point of this short, ground-up introduction to MCMC is that the technique is not magic by demonstrating that we could do-it-from-scratch with small amounts of code. The challenge is that it's just computationally intensive. Modern libraries perform the sampling for you with more advanced algorithms than Metropolis-Hastings.

14.5.5. MCMC Algorithms

The Metropolis-Hastings algorithm is simple, but somewhat inefficient. Some challenges with MCMC sampling are both mathematical and computational:

14. Statistical Inference and Information Theory

1. Oftentimes the algorithm will back-track (take a “U-Turn”), wasting steps in regions already explored.
2. The algorithm can have a very high rate of rejecting proposals if the proposal mechanism generates steps that would move the current state into a low-probability region.
3. The choice of proposal distribution and parameters can greatly influence the speed of convergence. Too large of movement and key regions can be entirely skipped over, while small movements can take much longer than necessary to explore the space.
4. As the number of parameters grows, the dimensionality of the parameter space to explore also grows making posterior exploration much harder.
5. The shape of the posterior space can be more or less difficult to explore. Complex models may have regions of density that are not nicely “round” - regions may be curved, donut shaped, or disjointed.

The issues above mean that MCMC sampling is very computationally expensive for more complex examples. Compared with Metropolis-Hastings, modern algorithms (such as the No-U-Turn (NUTS)) algorithm explore the posterior distribution more efficiently by avoiding back-tracking to already explored regions and dynamically adjusting the proposals to adaptively fit the posterior. Many of them take direct influence from particle physics, with the algorithm keeping track of the energy of the current state as it explores the posterior space.

Algorithms have only brought so much relief to the modeler with finite resources and compute. There is still a lot of responsibility for modeler to design models that are computationally efficient, transformed to eliminate oddly-shaped density regions, or find the right simplifications to the analysis in order to make the problem tractable.

Note

What does it mean to transform the parameter space?

An example will be shown in Chapter 31 where we want to ensure that a binomial variable is constrained to the region $[0, 1]$ but the underlying factors are allowed to vary across the entire real numbers. We use a logit (or inverse logit, a.k.a. logistic) to transform the parameters to the required probability range for the binomial outcome.

Another common transform is “Normalizing” the data to center the data around zero and to scale the outcomes such that the sample standard deviation is equal to one.

14.5.6. Rainfall Example (Continued)

We will construct a Bayesian model using the `Turing.jl` library. Using a battle-tested library allows us to step back from the intricacies of defining our own sampler and routine and focus on the models and analysis. The goal is to fit the parameters of one of the competing models from above in order to demonstrate an MCMC analysis workflow and essential concepts.

The first thing that we will do is use `Turing`'s `@model` macro to define a model. This has a few components:

1. The “model” is really just a Julia function that takes in data and relates the data to the statistical outcomes modeled.
2. The `~` is the syntax to relate a parameter to a prior assumption.
3. A loop (or broadcasted `.~`) that ties specific data observations to the random process.

Think of the `@model` block really as a model *constructor*. It isn’t until we pass data to the model that you get a fully instantiated `Model` type¹⁰.

¹⁰Specifically: a `DynamicPPL.Model` type (PPL = Probabilistic Programming Language).

Here's what defining the LogNormal model looks like in Turing. We have to specify prior distributions for LogNormal parameters.

```
using Turing

@model function rain_lognormal(data)
    μ ~ Normal(4.0, 1.0)                                ①
    σ ~ Exponential(2.0)                                 ②
    for i in eachindex(data)
        data[i] ~ LogNormal(μ, σ)                         ③
    end
end

m = rain_lognormal(rain)
```

- ① Defining the model uses the `@model` macro from Turing.
- ② We presume that there will be positive rainfall and 96% of mean annual rainfall will be somewhere between $\exp(2)$ and $\exp(6)$, or 7 and 403 inches.
- ③ In a LogNormal model, 0.5 deviations covers a lot of variation in outcomes.

```
DynamicPPL.Model{typeof(rain_lognormal), (:data,), (), (), Tuple{Vector{Float64}},,
    ↳ Tuple{}, DynamicPPL.DefaultContext}(rain_lognormal, (data = [39.51, 42.65,
    ↳ 44.09, 41.92, 28.42, 58.65, 30.18, 64.4, 29.02, 37.0 ... 12.75, 23.35, 29.59,
    ↳ 44.47, 20.06, 46.48, 13.46, 9.34, 16.51, 48.24],), NamedTuple(),
    ↳ DynamicPPL.DefaultContext())}
```

14.5.6.1. Setting Priors

In the example above, we used “weakly informative” priors. We constrained the prior probability to plausible ranges, knowing enough about the system of study (rainfall) that it would be completely implausible for there to be a $\text{Uniform}(0, \text{Inf})$ distribution of mean log-rainfall total, knowing that rain can't fall in infinite quantities.

Admittedly, we haven't confirmed with a meteorologist that $\exp(20)$ (485 million) inches of rain per year is impossible. But such is the beauty of the transparency of Bayesian analysis that the prior assumption is right there! Front and center and ready to be debated by other modelers! If you think that 485 million inches of rain is possible next year then you can challenge this assumption and propose another explicit alternative.

“Strongly informative” priors are those where we want to encode a stronger assumption about the plausible range of outcomes, such as if we knew enough about the problem domain that we could tell given the location of the rainfall, we'd expect 95% of the rainfall to be between, say, 10 and 30 inches per year. Then we could constrain the prior even more than we did above.

“Uninformative” priors use only maximum entropy or uniform priors to avoid encoding other assumptions into the model.

14.5.6.2. Sampling

Analysis should begin by evaluating the prior assumptions for reasonability and coverage over possible outcomes of the process we are trying to model. The top plot in Figure 14.8 shows the modeled rainfall outcomes taking on a wide range of possible outcomes. If we had more knowledge of the system we could enforce a stronger (narrower) prior assumption to constrain the model to a smaller set of values.

14. Statistical Inference and Information Theory

The object returned is an MCMCChains structure containing the samples as well as diagnostic information. Summary information gets printed below.

```
| chain_prior = sample(m, Prior(), 1000)
```

Chains MCMC chain (1000×5×1 Array{Float64, 3}):

```
Iterations      = 1:1:1000
Number of chains = 1
Samples per chain = 1000
Wall duration    = 0.89 seconds
Compute duration = 0.89 seconds
parameters       = μ, σ
internals        = lp, logprior, loglikelihood
```

Use `describe(chains)` for summary statistics and quantiles.

Figure 14.5.: Model output for the sampled prior. This isn't running an MCMC algorithm, it's simply taking draws from the defined prior assumptions.

Assessment of samples from the prior should include:

- Confirming that the model's behavior is reasonable.
- Confirming that the model covers the range of possible data that might be observed.

The sample outcomes from the modeled prior are shown in Figure 14.8.

Next, we sample the posterior by using the No-U-Turns (NUTS) algorithm and drawing 1000 samples (not including the warm-up phase). This is the primary result we will analyze further.

```
| chain_posterior = sample(m, NUTS(), 1000)
```

Chains MCMC chain (1000×16×1 Array{Float64, 3}):

```
Iterations      = 501:1:1500
Number of chains = 1
Samples per chain = 1000
Wall duration    = 2.02 seconds
Compute duration = 2.02 seconds
parameters       = μ, σ
internals        = n_steps, is_accept, acceptance_rate, log_density,
                   ↵ hamiltonian_energy, hamiltonian_energy_error, max_hamiltonian_energy_error,
                   ↵ tree_depth, numerical_error, step_size, nom_step_size, lp, logprior,
                   ↵ loglikelihood
```

Use `describe(chains)` for summary statistics and quantiles.

Figure 14.6.: Model output for the sampled posterior.

14.5.6.3. Diagnostics

Before analyzing the result itself, we should check a few things to ensure the model and sampler were well behaved. MCMC techniques are fundamentally stochastic and randomness can cause an

errant sampling path. Or a model may be mis-specified such that the parameter space to explore is incompatible with the current algorithm (or any known so far).

A few things we can check:

First, the `ess` or **effective sample size** which adjusts the number of samples for the degree of autocorrelation in the chain. Ideally, we would be able to draw independent samples from the posterior, but due to the Markov-Chain approach the samples can have autocorrelation between neighboring samples. We collect less information about the posterior in the presence of positive autocorrelation.

An `ess` greater than our sample indicates that there was less (negative) autocorrelation than we would have expected for the chain. An `ess` much less than the number of samples indicates that the chain isn't sampling very efficiently but, aside from needing to run more samples, isn't necessarily a problem.

```
ess(chain_posterior)

ESS

parameters      ess    ess_per_sec
Symbol      Float64      Float64

μ    764.6229      378.1518
σ    467.9178      231.4134
```

Second, the `rhat` (\hat{R}) is the Gelman-Rubin convergence diagnostic and its value should be very close to 1.0 for a chain that has converged properly. Even a value of 1.01 may indicate an issue and quickly gets worse for higher values.

```
rhat(chain_posterior)

R-hat

parameters      rhat
Symbol      Float64

μ    1.0020
σ    1.0061
```

Next, we can look at the “trace” plots for the parameters being sampled (Figure 14.7). These are sometimes called “hairy caterpillar” plots because in a healthy chain sample, we should see a series without autocorrelation and that the values bounce around randomly between individual samples.

```
let
  f = Figure()
  ax1 = Axis(f[1,1], ylabel="μ")
  lines!(ax1, vec(get(chain_posterior,:μ).μ.data))
  ax2 = Axis(f[2,1], ylabel="σ")
  lines!(ax2, vec(get(chain_posterior,:σ).σ.data))
  f
end
```

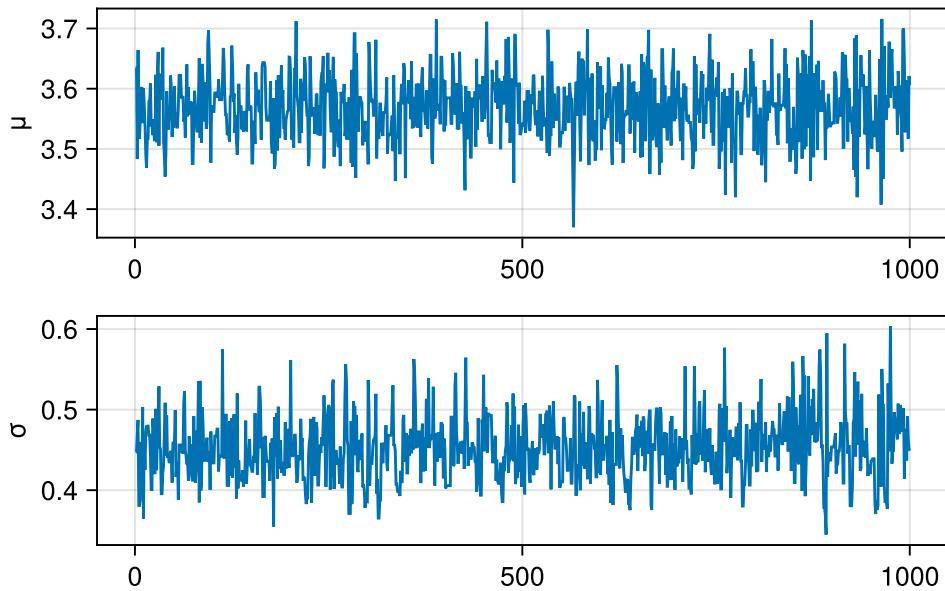


Figure 14.7.: The trace plots indicate low autocorrelation which is desirable for an MCMC sample.

The `ess`, `rhat`, and trace plots all look good for our sampled chain so next we will analyze the results in the context of our rainfall problem.

14.5.6.4. Analysis

Let's see how it looks compared to the data first. Figure 14.8 shows 200 samples from the prior and posterior. The prior (top) shows how wide the range of possible rainfall outcomes could be using our weakly informative prior assumptions. The bottom shows that after having learned from the data, the posterior probability of rainfall has narrowed considerably.

```
function chn_cdf!(axis,chain,rain)
    n = 200
    s = sample(chain, n)
    vals = get(sample(chain, 200), [:μ, :σ])
    ds = LogNormal.(vals.μ, vals.σ)
    xs = range(1.0, stop=200.0, length=200)
    for d in ds
        lines!(axis, xs, cdf.(d, xs), color=(:gray, 0.3))
    end

    # plot the actual data
    percentiles= 0.01:0.01:0.99
    lines!(axis,quantile.(Ref(rain),percentiles),percentiles,linewidth=3)
end

let
    f = Figure()
```

```

ax1 = Axis(
    f[1,1],
    title="Prior",
    xgridvisible=false, ygridvisible=false,
    ylabel="Quantile"
)

chn_cdf!(ax1,chain_prior,rain)

ax2 = Axis(
    f[2,1],
    title="Posterior",
    xgridvisible=false, ygridvisible=false,
    xlabel="Annual Rainfall (inches)",
    ylabel="Quantile"
)

chn_cdf!(ax2,chain_posterior,rain)

linkxaxes!(ax1, ax2)

f
end

```

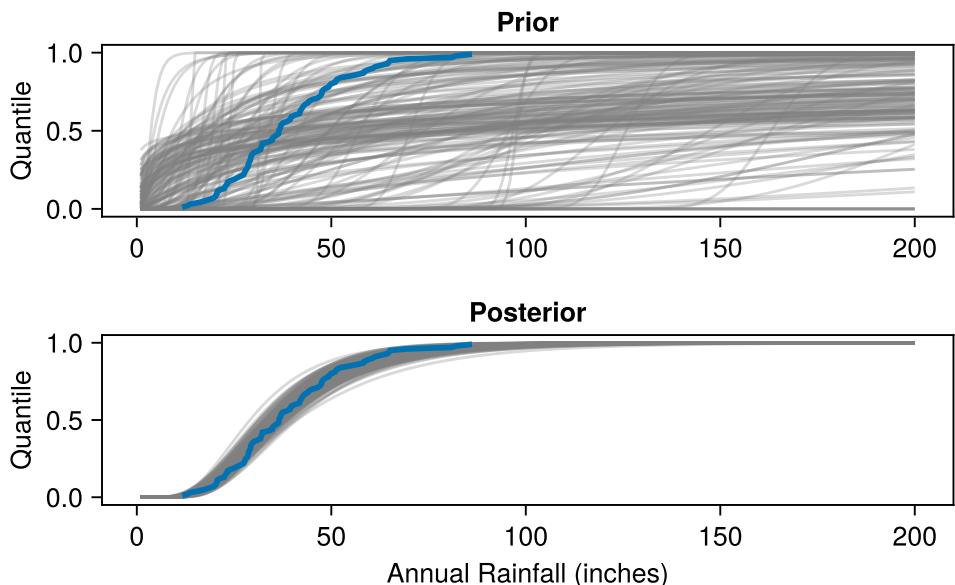


Figure 14.8.: The prior model shows a wide range of possible outcomes, and the shape of the distribution is reasonable: there's a nice 'S' shape to the CDF, indicating a dense region where most outcomes would fall in the PDF. The fitted posterior model (bottom) has good coverage of the observed data (shown in blue).

Comparing to the maximum likelihood analysis from before by plotting the MLE point estimate

14. Statistical Inference and Information Theory

onto the marginal densities in Figure 14.9. The peak of the posterior is referred to as the **maximum a posteriori** (MAP) and would be the point estimate proposed by this Bayesian analysis. However, the Bayesian way of thinking about distributions of outcomes rather than point estimates is one of the main aspects we encourage for financial modelers. Using the posterior distribution of the parameters, we can assess parameter uncertainty directly instead of ignoring it as we tend to do with point estimates.

```
let
    # get the parameters from the earlier MLE approach
    p = params(h2_dist)

    f = Figure()

    # plot μ posterior
    ax1 = Axis(f[1,1],title="μ posterior",xgridvisible=false)
    hideydecorations!(ax1)
    d = density!(ax1,vec(get(chain_posterior,:μ).μ.data))
    l = vlines!(ax1,[p[1]],color=:red)

    # plot σ posterior
    ax2 = Axis(f[2,1],title="σ posterior", xgridvisible=false)
    hideydecorations!(ax2)
    density!(ax2,vec(get(chain_posterior,:σ).σ.data))
    vlines!(ax2,[p[2]],color=:red)

    Legend(f[1,2],[d,l],["Posterior Density", "MLE Estimate"])

    f
end
```

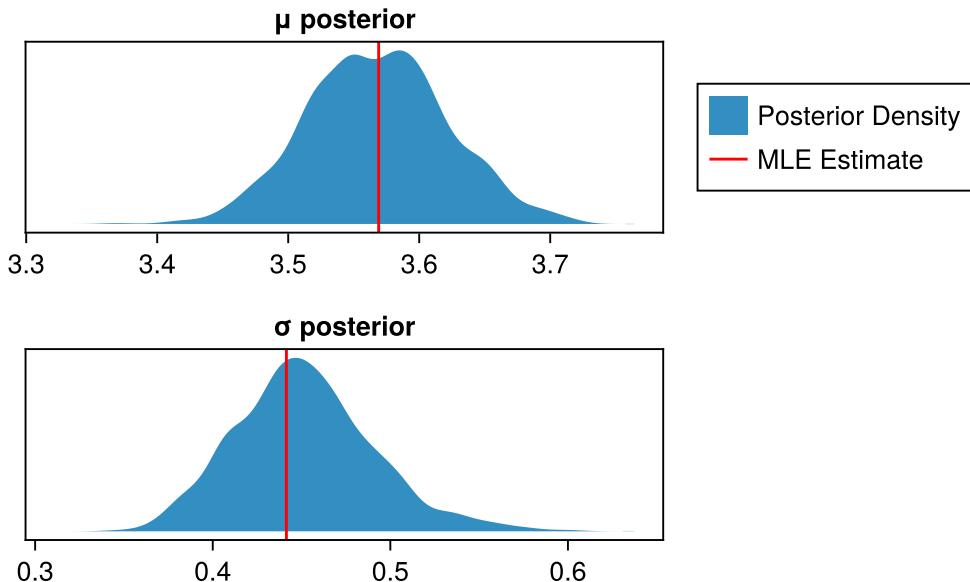


Figure 14.9.: The MLE point estimate need not necessarily align with the peak or center of posterior densities (e.g. in the case of a bimodal distribution).

14.5.6.5. Model Limitations

We have built and assessed a simple statistical model that could be used in the estimation of risk for a particular location. Nowhere in our model did we define a mechanism to capture a more sophisticated view of the world. There is no parameter for changes over time due to climate change, or inter-annual seasonality for El Niño or La Niña cycles, or any of a multitude of other real-world factors that can influence the forecasting. All we've defined is that there is a LogNormal process generating rainfall in a particular location. This may or may not be sufficient to capture the dynamics of the problem at hand.

Part of the benefits of the Bayesian approach is that it allows us to extend the statistical model to be arbitrarily complex in order to capture our intended dynamics. We are limited by the availability of data, computational power and time, and our own expertise in the modeling. Regardless of the complexity of the model, the same fundamental techniques and ideas apply in the Bayesian approach.

14.5.6.6. Continuing the Analysis

Like any good model, you can often continue the analysis in any number of directions, such as: collecting more data, evaluating different models, creating different visualizations, making predictions about future events, creating a multi-level model that predicts rainfall for multiple related locations simultaneously, among many other threads of analysis.

Earlier we discussed model comparison. To compute a real Bayes factor in comparing the different models, we would take the average likelihood across the posterior samples instead of just comparing the maximum likelihood points as we did earlier. There are more sophisticated tools for estimating out-of-sample performance of the model, or measures that evaluate a model for

14. Statistical Inference and Information Theory

over-fitting by penalizing the diagnostic statistic for the model having too many free parameters. See LOO (leave-one-out) cross-validation and various “information criteria” in the resources listed in Section 14.5.8.

14.5.7. Conclusion

This chapter introduced the foundations of statistical inference and modern Bayesian tools. The underlying information-theoretic and mathematical concepts can be challenging, especially when traditional finance and actuarial curricula don’t emphasize the computational foundations. Most importantly: moving beyond single ‘best estimate’ values to embrace full probability distributions leads to richer financial analysis and more comprehensive risk assessment.

14.5.8. Further Reading

Bayesian approaches to statistical problems are rapidly changing the professional statistical field. To the extent that the actuarial profession incorporates statistical procedures, financial professionals should consider adopting the same practices. The benefits of this are a better understanding of the distribution of risk and return, results that are more interpretable and explainable, and techniques that can be applied to a wider range of problems. The combination of these things would serve to enhance best practices related to understanding and communicating about financial quantities.

Textbooks recommended by the author are:

- Statistical Rethinking (McElreath)
- Bayes Rules! (Johnson, Ott, Dogucu)
- Bayesian Data Analysis (Gelman, et al.)

Chi Feng has an interactive demonstration of different MCMC samplers available at: <https://chi-feng.github.io/mcmc-demo/>.

15. Stochastic Modeling

CHAPTER AUTHORED BY: YUN-TIEN LEE, ALEC LOUDENBACK

The Monte Carlo Method: (i) A last resort when doing numerical integration, and (ii) a way of wastefully using computer time. - Malvin H. Kalos^a (c. 1960)

^aKalos was a pioneer in Monte Carlo techniques, quoted via https://doi.org/10.1007/978-3-540-74686-7_3

15.1. Chapter Overview

Brief overview of stochastic modeling concepts. Random inputs vs. embedded randomness in stepwise models. Key components: distributions, time horizons, state space. Evaluating outputs with expectation, variance, and risk measures. Special properties like Markov, stationarity, and SDEs. Practical finance examples with Julia code for scenario generation and macroeconomic simulations. An overview of pseudo random number generators (PRNGs).

15.2. Introduction

Stochastic modeling is the technique of running many similar scenarios through a model in order to evaluate different aspects of a given model. Some contexts in which stochastic modeling arises in the financial context include:

- **Valuation.** Market consistent pricing often involves calibrating a set of stochastic economic scenarios to market-observed prices. Those calibrated scenarios are then used to determine the value of another set of contracts for which market prices are not directly observable.
- **Scenario Analysis.** Many financial models have a high degree of complexity and interactions between components not well understood *a priori*. Using a set of calibrated or otherwise arbitrary scenarios can illuminate unexpected or interrelated behavior that arises from the model. Sometimes financial assets and liabilities have a path-dependency, meaning that the overall behavior or value is a function of the entire path taken (e.g. an Asian option).
- **Risk and Capital Analysis.** Whether model parameters are point estimates ("an asset has a 1% probability of default") or distributions ("an asset has a probability of default distributed as $\text{Beta}(1, 99)$ "), stochastic modeling can be utilized to determine projected capital levels for a company or line of business at certain security levels.

The reliability of modeled outcomes is fundamentally dependent on the quality of both the underlying scenarios and the model framework itself. For example, if your model of interest rates produces a range of short rates that is very narrow (relative to historical values or implied volatilities), and you then run risk metrics using scenarios with unrealistically narrow short rates, you may find false comfort in the small world (Section 3.2.1) of your model.

Recall the kinds of uncertainties in Table 4.2 - stochastic modeling implementations often mechanically only address the aleatory (process) volatility component of outcome uncertainty. Through techniques like developing a posterior distribution for model parameters (Section 14.4) we can also explicitly attempt to model epistemic (parameter) uncertainty as well.

15.3. Fundamentals of Stochastic Modeling

Stochastic modeling explicitly incorporates randomness and uncertainty—a significant departure from deterministic approaches. First, the fundamental concepts.

15.3.1. Random Variables and Probability

At the core of stochastic modeling is the concept of random variables—quantities whose values depend on random outcomes. In financial contexts, random variables might represent:

- Future interest rates
- Stock price movements
- Insurance claim frequencies
- Default probabilities
- Mortality rates

These random variables are characterized by probability distributions that describe the likelihood of different outcomes. For example, stock returns might follow a normal distribution, while insurance claims might follow a Poisson distribution for frequency and a LogNormal distribution for severity.

15.3.2. The Role of Monte Carlo Simulation

Despite the opening quote's humorous skepticism, **Monte Carlo methods** have become indispensable in modern financial modeling. Named after the famous casino, these techniques use repeated random sampling to obtain numerical results when analytical solutions are impractical or impossible.

The basic procedure involves:

1. Generating random inputs based on specified probability distributions
2. Running the model with these inputs
3. Collecting and analyzing the outputs
4. Repeating until sufficient samples are obtained for statistical significance

As computational power has increased, Monte Carlo methods have moved from “last resort” to standard practice in many financial applications. In addition to this conceptual chapter, Chapter 25 demonstrates this process to generate stochastic economic scenarios.

15.3.3. Time Dimensions in Stochastic Models

Stochastic models can be categorized based on how they treat time:

- Discrete-time models: The system evolves in distinct time steps (daily, monthly, annually)
- Continuous-time models: The system evolves continuously, often described by stochastic differential equations

Financial applications frequently use both approaches. For example, option pricing might use continuous-time models (Black-Scholes), while asset-liability management might use discrete-time models with monthly or quarterly steps.

15.3.4. State Spaces and Transitions

A stochastic model's **state space** represents all possible states the system can occupy. This might be:

- Discrete: Limited number of possible states (e.g., credit ratings: AAA, AA, A, etc.)
- Continuous: Infinite possible states (e.g., stock prices can take any positive real value)

The transitions between states are governed by probability distributions or transition matrices, depending on the model type. Understanding the state space is crucial for properly formulating and solving stochastic problems in finance.

15.3.5. Aleatory vs Epistemic Uncertainty

Stochastic models address two fundamental types of uncertainty:

- Aleatory uncertainty: The inherent randomness in the system (e.g., market volatility)
- Epistemic uncertainty: Uncertainty due to limited knowledge about model parameters

Most basic stochastic models focus on aleatory uncertainty, assuming model parameters are known with certainty. More sophisticated approaches incorporate epistemic uncertainty by treating parameters themselves as random variables, often using Bayesian methods.

15.3.6. Calibration and Validation

Before a stochastic model can be applied, it must be properly calibrated to match observed data or market prices. This typically involves:

1. Establishing initial parameters via professional judgment or statistical means.
2. Adjusting parameters to match desired calibration data: e.g., minimizing a loss function with respect to the predicted outcomes versus the calibration data (see Chapter 17).
3. Validating the model against out-of-sample data.

For financial applications, models are often calibrated to be "market-consistent," meaning they reproduce the prices of traded instruments before being used to value more complex products.

15.3.7. Computational Considerations

Stochastic modeling is computationally intensive, particularly when:

- Many scenarios are required for convergence
- The model has high dimensionality
- Path dependencies require storing entire trajectories

Modern implementations leverage parallel processing, variance reduction techniques, and sometimes machine learning approximations to make complex stochastic models computationally feasible.

With these fundamental concepts in mind, we can now explore specific types of stochastic models and their applications in finance and actuarial science.

15.3.8. Kinds of Stochastic Models

15.3.8.1. Input Ensemble

In this approach, you first generate many sets of random inputs (scenarios) and then run the same model repeatedly—one run per scenario¹:

- **Interest Rate Scenarios for Bond Portfolios**

Generate thousands of yield-curve scenarios (e.g., parallel shifts, twists) to test how a bond portfolio might perform over time. By averaging the results over all simulations, you can estimate the portfolio's expected return or risk metrics.

- **Commodity Price Paths for Hedging**

Create multiple possible paths for commodity prices under different macro conditions. Use these to evaluate various hedging strategies and see which one consistently mitigates risk across a wide range of market conditions.

Generally, models programmed in code can easily map across a large number of stochastic inputs.

15.3.8.2. Random State

Here, the model itself “rolls the dice” at each step. Instead of pre-generating input scenarios, the randomness is embedded in the logic that evolves the state over time²:

- **Property & Casualty Claims Simulation**

In each simulated month, randomly draw the number of claims (from a Poisson or negative binomial distribution), then randomly assign the severity of each claim (often a lognormal or gamma distribution). The insurer's financial state (reserves, surplus) evolves based on these simulated losses.

- **Call-Center (Queueing) Model**

Model the random arrival of customer calls following a Poisson process, and at each arrival decide if the call is answered immediately or queued. Service times for each call can be drawn from an exponential distribution. By simulating many days, you can estimate average waiting times or the probability that a caller has to wait more than a certain threshold.

 Note

Many financial models project outcomes using *expected values* of random variables. Many actuarial models utilize this approach to determine expected outcomes given an assumption (such as expected present value of life insurance claims). This is different than evaluating many possible discrete realized scenarios wherein each insured life either dies or survives each period, and models designed for projected expected payments need to be adapted to handle a random state approach.

¹An extended example of this approach is discussed in Chapter 25.

²A worked example of this approach is illustrated in Chapter 30.

15.3.8.3. Closed-Form

Some stochastic processes admit analytical solutions or formulas for key quantities, so you do not need to run a simulation for each scenario. For example, here are a couple of common models that have closed form solutions to the underlying stochastic differential equation.

- **Black-Scholes Option Price**

The Black-Scholes formula provides an analytical price for European options despite the underlying theory being based on a stochastic evolution of asset prices. The related put-call parity is a model-free no-arbitrage relationship that holds regardless of the assumed price dynamics.

- **Ornstein-Uhlenbeck Interest Rate Model**

An Ornstein-Uhlenbeck process is often used to model mean-reverting interest rates. Despite being fundamentally stochastic, it has closed-form expressions for zero-coupon bond prices and other interest-rate derivatives, saving you from running thousands of random simulations for valuation.

The focus of this section will be on the other kinds of computationally driven stochastic models.

15.3.8.4. Special Cases or Properties

Stochastic processes often combine several structural or theoretical properties to reflect real-world phenomena in financial and actuarial models. Before diving into each property—Markov, stationarity, martingales, and SDEs—it can be useful to see how they serve as building blocks. For example, Markov processes allow today's decisions or valuations to depend only on the current state, whereas stationarity implies that the underlying statistical characteristics (such as mean or variance) do not change over time. Martingales capture a notion of "fairness" in gambling and risk-neutral pricing, and SDEs inject random shocks in a continuous-time setting. By understanding these properties, one can better see which assumptions are embedded in derivatives pricing, credit migration models, or mortality forecasts.

15.3.8.4.1. Markov Models

A stochastic process has the Markov property if the future state depends only on the current state and not on the past states (memorylessness). Markov chains and hidden Markov models contain this property. The Markov property simplifies modeling and computation by reducing dependencies on past states. For instance, a credit rating model might assume that the probability of going from BBB to BB depends only on today's rating, without direct reference to prior years' ratings.

15.3.8.4.2. Stationary Models

A stochastic process is stationary if its statistical properties, such as mean and variance, do not change over time. Strict stationarity means all moments of the process are time-invariant, while weak stationarity means the mean is constant, the variance is finite, and the autocovariance depends only on the lag (not on time itself). Stationarity simplifies the analysis and modeling of stochastic processes, especially for time series data. Sometimes a non-stationary model can be made stationary by transformation (such as removing a constant trend component).

15. Stochastic Modeling

15.3.8.4.3. Martingales

A martingale is a stochastic process where the expected future value, given all prior information, is equal to the current value. Examples include fair gambling games or financial asset prices in efficient markets. Martingales are important in financial modeling and risk-neutral pricing, especially for derivatives.

15.3.8.4.4. Stochastic Differential Equations (SDEs)

These are differential equations that incorporate stochastic terms (typically driven by Brownian motion or other noise sources). The well-known Black-Scholes equation for option pricing is one example of how SDEs can be applied in the real world. SDEs are essential for modeling systems where both deterministic and random factors drive behavior over time.

A fundamental building block of SDEs is **Brownian Motion** (also called a Wiener process). This is a continuous-time stochastic process where changes over time are independent and normally distributed. This is extensively used in financial models for asset price movements (e.g., the Black-Scholes model). In addition to financial markets, Brownian motion is also widely used in modeling physical systems (diffusion).

In practice, these properties often occur together in financial models.

15.3.9. Components of Stochastic Models

Stochastic modeling has additional terminology to introduce.

15.3.9.1. Random variables

A random variable represents a quantity whose value is determined by the outcome of a random event. It can be discrete or continuous. It is essentially a mapping from event space to numerical values. Examples include stock prices, waiting time in queues, and number of claims in insurance. Random variables form the basis of stochastic models by introducing uncertainty into the model.

15.3.9.2. Probability distributions

A probability distribution describes the likelihood of different outcomes for a random variable. Examples include Normal distribution, Poisson distribution, and Exponential distribution. Probability distributions help in modeling the behavior of random variables and in defining how likely different events or outcomes are.

15.3.9.3. State space

The state space represents all possible states or values that a stochastic process can take. For example, in a Markov process, the state space might be the set of all possible values that the system can occupy at any given time. The state space helps in defining the scope of the model by specifying possible outcomes.

15.3.9.4. Stochastic processes

A stochastic process is a collection of random variables indexed by time (or some other variable) that evolve in a probabilistic manner. Examples include Brownian motion, Markov chains and Poisson processes. Stochastic processes model how random variables change over time, which is essential for understanding dynamic systems influenced by randomness.

15.3.9.5. Transition probabilities

These represent the probabilities of transitioning from one state to another in a stochastic process. For example, in a Markov chain, the transition matrix contains the probabilities of moving from one state to another. Transition probabilities determine how the system evolves from one time step to the next, reflecting the underlying randomness.

15.3.9.6. Time horizon

The time horizon refers to the period over which the stochastic process is observed. It can be discrete (e.g., steps in a Markov chain) or continuous (e.g., continuous-time models like Brownian motion). The time horizon helps in determining how the process behaves over short or long periods.

15.3.9.7. Initial conditions

These are the starting points or initial values of the random variables or the system at time zero. They may be the initial price of a stock, the initial number of customers in a queue, etc. The starting condition influences the future evolution of the process, and different initial conditions can lead to different outcomes.

15.3.9.8. Noise (random shocks)

Random noise represents unpredictable random fluctuations that can affect the outcome of a stochastic process. This can be market volatility, measurement errors, or environmental variations. Noise is a critical element in stochastic models as it introduces randomness and uncertainty into otherwise deterministic systems.

15.4. Evaluating and Applying Stochastic Models

15.4.1. Evaluating Stochastic Results

These types of models are evaluated simply by running them many times until the measure of interest converges on a stable result: for example, we might run a model until the mean of the results no longer varies materially as we add more scenarios.

In practice, we watch the **standard error** of the estimate,

$$\text{SE}(\hat{\mu}) = \frac{s}{\sqrt{N}},$$

where s is the sample standard deviation and N is the scenario count. Doubling accuracy therefore requires roughly four times as many runs. Plotting running estimates (mean, VaR, CTE) versus

15. Stochastic Modeling

scenario count is an easy convergence diagnostic; once the curve flattens relative to your materiality threshold, stop the Monte Carlo.

15.4.1.1. Expectation and variance

The expected value (mean) represents the average or mean outcome of a random variable over many trials or realizations. The variance measures the spread or variability of outcomes around the expected value. These statistical measures provide insights into the central tendency and the uncertainty or risk in a stochastic model.

15.4.1.2. Covariance and correlation

Covariance measures how two random variables change together. Positive covariance indicates that the variables tend to increase together. Correlation is the standardized version of covariance that measures the strength of the linear relationship between two variables. Understanding how different random variables interact helps in building more complex models, especially in multivariate stochastic processes.

15.4.1.3. Risk Measures

Risk measures encompass the set of functions that map a set of outcomes to an output value characterizing the associated riskiness of those outcomes. As is usual when attempting to compress information (e.g., condensing information into a single value), there are multiple ways we can characterize this riskiness.

15.4.1.3.1. Coherence & Other Desirable Properties

Further, it is desirable that a risk measure has certain properties, and risk measures that meet the first four criteria are called “coherent” in the literature. From “An Introduction to Risk Measures for Actuarial Applications” ((Hardy 2006)), she describes four properties. Using H as a risk measure and X as the associated risk distribution:

15.4.1.3.1.1. 1. Translation Invariance

For any non-random c

$H(X + c) = H(X) + c$ This means that adding a constant amount (positive or negative) to a risk adds the same amount to the risk measure. It also implies that the risk measure for a non-random loss, with known value c , say, is just the amount of the loss c .

15.4.1.3.1.2. 2. Positive Homogeneity

For any non-random $\lambda > 0$:

$$H(\lambda X) = \lambda H(X)$$

This axiom implies that changing the units of loss does not change the risk measure.

15.4.1.3.1.3. 3. Subadditivity

For any two random losses X and Y ,

$$H(X + Y) \leq H(X) + H(Y)$$

It should not be possible to reduce the economic capital required (or the appropriate premium) for a risk by splitting it into constituent parts. Or, in other words, diversification (i.e., consolidating risks) cannot make the risk greater, but it might make the risk smaller if the risks are less than perfectly correlated.

15.4.1.3.1.4. 4. Monotonicity

If $Pr(X \leq Y) = 1$ then $H(X) \leq H(Y)$.

If one risk is always bigger than another, the risk measures should be similarly ordered.

15.4.1.3.1.5. Other Properties

In “Properties of Distortion Risk Measures” (Balbás, Garrido, and Mayoral 2009) also note other properties of interest:

Complete

Completeness is the property that the distortion function associated with the risk measure produces a unique mapping between the original risk’s survival function $S(x)$ and the distorted $S^*(x)$ for each x . See the Distortion Risk Measures note below for more detail on this.

In practice, this means that a non-complete risk measure ignores some part of the risk distribution (e.g. CTE and VaR don’t use the full distribution).

Exhaustive

A risk measure is “exhaustive” if it is coherent and complete.

Adaptable

A risk measure is “adapted” or “adaptable” if its distortion function satisfies the following conditions (see the Distortion Risk Measures note below):

1. g is strictly concave, that is g' is strictly decreasing.
2. $\lim_{u \rightarrow 0^+} g'(u) = \infty$ and $\lim_{u \rightarrow 1^-} g'(u) = 0$.

Adaptable risk measures give more weight to larger losses. They are exhaustive but the converse is not true.

| Measure | Coherent | Complete | Exhaustive | Adaptable |
|---------|----------|----------|------------|-----------|
|---------|----------|----------|------------|-----------|

15.4.1.4. Summary of Risk Measure Properties

| Measure | Coherent | Complete | Exhaustive | Adaptable |
|--------------------------------------|----------|----------|------------|-----------|
| VaR | No | No | No | No |
| CTE | Yes | No | No | No |
| DualPower
($y > 1$) | Yes | Yes | Yes | No |
| Proportional-Hazard ($\gamma > 1$) | Yes | Yes | Yes | No |
| WangTransform | Yes | Yes | Yes | Yes |

i Distortion Risk Measures

Distortion Risk Measures are a way of remapping the probabilities of a risk distribution in order to compute a risk measure H on the risk distribution X .

Adapting (Wang 2002), there are two key components:

15.4.1.5. Distortion Function $g(u)$

This remaps values in the $[0,1]$ range to another value in the $[0,1]$ range, and in H below, operates on the survival function S and $F = 1 - S$.

Let $g : [0, 1] \rightarrow [0, 1]$ be an increasing function with $g(0) = 0$ and $g(1) = 1$. The transform $F^*(x) = g(F(x))$ defines a distorted probability distribution, where “ g ” is called a distortion function.

Note that F^* and F are equivalent probability measures if and only if $g : [0, 1] \rightarrow [0, 1]$ is continuous and one-to-one. We define a family of distortion risk-measures using the mean-value under the distorted probability $F^*(x) = g(F(x))$:

15.4.1.6. Risk Measure Integration

To calculate a risk measure H , we integrate the distorted F across all possible values in the risk distribution (i.e. $x \in X$):

$$H(X) = E^*(X) = - \int_{-\infty}^0 g(F(x))dx + \int_0^{+\infty} [1 - g(F(x))]dx$$

That is, the risk measure (H) is equal to the expected value of the distortion of the risk distribution ($E^*(X)$).

15.4.1.7. Risk Measures: Examples

15.4.1.7.1. Basic Comparison

We won't re-implement the logic here, but here's a very simple example demonstrating the relative values of VaR, CTE, and a Wang Transform at the 90% level. These functions come from the public library `ActuaryUtilities.jl`.

```
using ActuaryUtilities
using Random
using Distributions

Random.seed!(1234)
let outcomes = rand(LogNormal(0.0, 0.5), 10_000)

@show VaR(0.9)(outcomes)
@show CTE(0.9)(outcomes)
@show WangTransform(0.9)(outcomes)
end

(VaR(0.9))(outcomes) = 1.9071039587647827
(CTE(0.9))(outcomes) = 2.495594005714287
(WangTransform(0.9))(outcomes) = 2.183978490896952

2.183978490896952
```

15.4.1.8. Plotting a Range of Values

We will generate a random outcome and show how the risk measures behave:

```
using Distributions, Random, ActuaryUtilities, CairoMakie

# the assumed distribution of outcomes
outcomes = Weibull(1, 5)

Random.seed!(2024)

αs = range(0.0, 0.99; length=100)

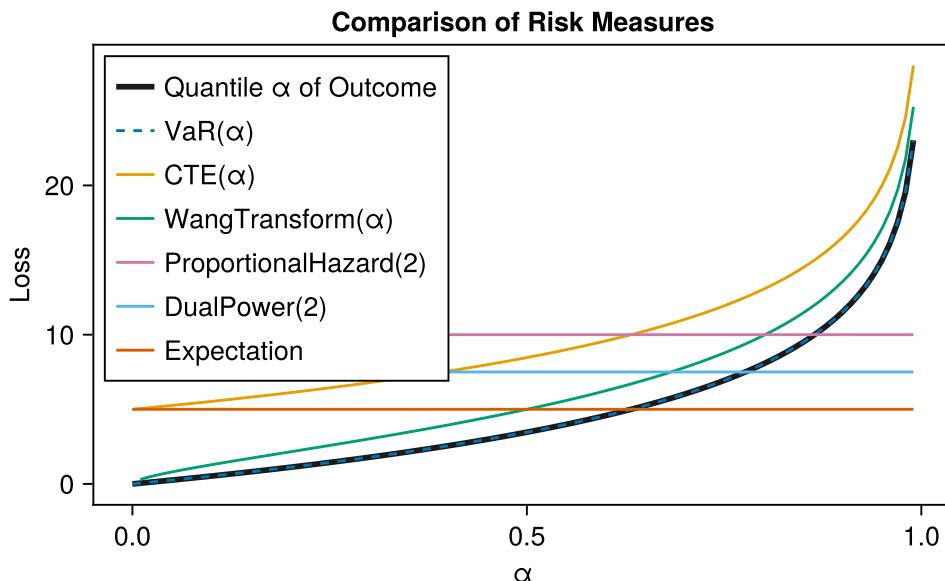
let
    f = Figure()
    ax = Axis(
        f[1, 1],
        xlabel="α",
        ylabel="Loss",
        title="Comparison of Risk Measures",
        xgridvisible=false,
        ygridvisible=false,
    )
    lines!(
        ax,
        αs,
        [quantile(outcomes, α) for α in αs],
        label="Quantile α of Outcome",
        color=:grey10,
        linewidth=3,
    )
end
```

```

    lines!(
        ax,
        αs,
        [VaR(α)(outcomes) for α in αs],
        label="VaR(α)",
        linestyle=:dash
    )
    lines!(
        ax,
        αs,
        [CTE(α)(outcomes) for α in αs],
        label="CTE(α)",
    )
    lines!(
        ax,
        αs[2:end],
        [WangTransform(α)(outcomes) for α in αs[2:end]],
        label="WangTransform(α)",
    )
    lines!(
        ax,
        αs,
        [ProportionalHazard(2)(outcomes) for α in αs],
        label="ProportionalHazard(2)",
    )
    lines!(
        ax,
        αs,
        [DualPower(2)(outcomes) for α in αs],
        label="DualPower(2)",
    )
    lines!(
        ax,
        αs,
        [RiskMeasures.Expectation()(outcomes) for α in αs],
        label="Expectation",
    )
    axislegend(ax, position=:lt)

f
end

```



15.4.2. Variance Reduction Techniques

While Monte Carlo simulation is a powerful approach for modeling complex financial or actuarial systems, it can require a very large number of random draws for the results to converge. When there has been a limited number of runs, the variance of the sampled results can be high and vary materially between different sets of stochastic runs. This can be addressed by simply running more scenarios, but can be computationally expensive, particularly when path-dependent processes—or high-dimensional models—are involved. For this reason, several “variance reduction techniques” have been established to improve efficiency without reducing accuracy. They work by designing the way we draw samples (or process them) so that desired statistics converge more quickly.

15.4.2.1. Control Variates

A control variate is a random variable whose expected value is already known (or can be estimated with high precision). By simulating the main process alongside the control variate and leveraging the differences in outcomes, the overall variance of the estimator is reduced. In finance, for instance, one might use a simpler instrument whose price is analytically known as a control variate when simulating a more complex derivative.

15.4.2.2. Antithetic Variates

In antithetic variates, each random draw (X) is paired with a corresponding draw ($-X$) or another symmetric transformation. The idea is that positive and negative deviations tend to cancel out, thereby reducing overall variability. For example, if simulating geometric Brownian motion paths, each path with a particular series of random shocks could be matched with a path using the same shocks negated. Updating estimates with the average result of these pairs generally yields a lower variance for the same number of total draws.

15. Stochastic Modeling

15.4.2.3. Stratified Sampling

In stratified sampling, instead of sampling purely at random across the entire probability distribution, the distribution is divided into “strata” (or sub-intervals), and draws are forced to fall into each of these strata with a pre-specified frequency. By covering the distribution more systematically, the resulting estimates often converge faster. A related approach is **quasi-Monte Carlo**, which uses deterministic low-discrepancy sequences (such as Sobol or Halton sequences) to span the sample space more uniformly than random sampling. Julia has a package for this called QuasiMonteCarlo.jl.

Together, these techniques reflect a crucial principle of computational thinking: by leveraging structure or known properties of the simulation, you can achieve more accurate estimates with fewer overall scenarios. This translates into faster runs, less strain on computational resources, and better performance when embedding these methods in large-scale models or iterative processes.

15.4.3. Stochastic Modeling: Examples

15.4.3.1. Macroeconomic Analysis

Here we show still another stochastic process in macroeconomic analysis. Stochastic macroeconomic analysis often involves modeling random shocks and their effects on macroeconomic variables such as output, consumption, inflation, and employment. One common approach is through Dynamic Stochastic General Equilibrium (DSGE) models, which are widely used in macroeconomic analysis. These models incorporate randomness (stochastic elements) to capture real-world uncertainty in economic systems.

```
using Random, CairoMakie, Distributions

# Parameters
α = 0.33                      # Capital share of output
δ = 0.05                        # Depreciation rate
s = 0.2                          # Savings rate
n = 0.01                        # Population growth rate
g = 0.02                        # Technology growth rate
σ = 0.01                        # Standard deviation of productivity shocks
T = 100                         # Number of periods to simulate
K₀ = 1.0                         # Initial capital stock
A₀ = 1.0                         # Initial productivity

# Shock distribution (normal distribution for productivity shocks)
shock_distribution = Normal(0, σ)

# Random number generator (Xoshiro as default with a seed to ensure
# reproducibility)
rng_gen = Xoshiro(1234)

# Function to simulate the model
function simulate_stochastic_solow(rng, T, α, δ, s, n, g, σ, K₀, A₀)
    K = zeros(T)                  # Capital over time
    Y = zeros(T)                  # Output over time
    A = zeros(T)                  # Productivity shocks over time
    A[1] = A₀                     # Initial productivity
    K[1] = K₀                     # Initial capital
```

```

for t in 1:(T-1)
    # Apply random productivity shock
    ε_t = rand(rng, shock_distribution)
    A[t+1] = A[t] * exp(ε_t) # Productivity evolves stochastically

    # Output based on Cobb-Douglas production function
    Y[t] = A[t] * K[t]^α

    # Capital accumulation equation
    K[t+1] = s * Y[t] + (1 - δ) * K[t]

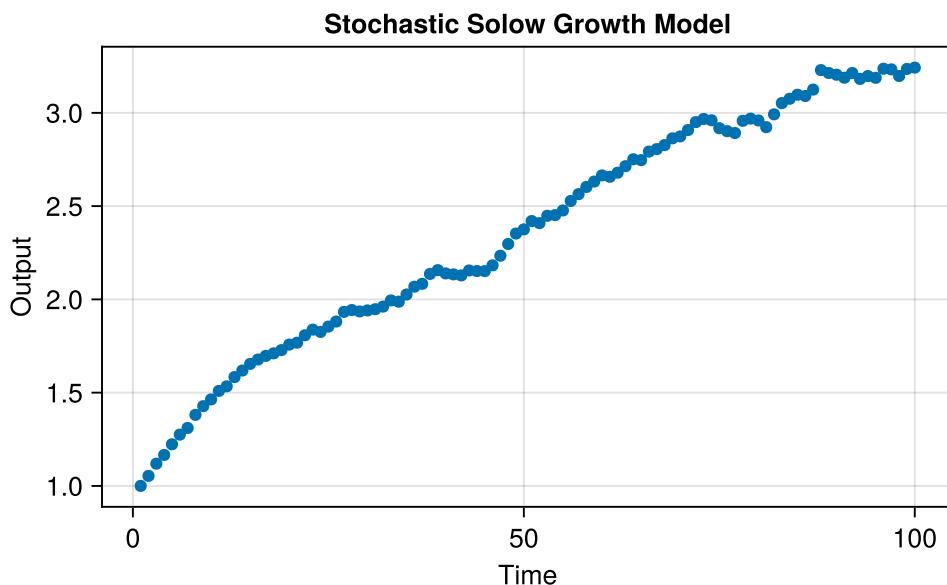
    # Population and technology growth
    K[t+1] *= (1 + n) * (1 + g)
end

Y[T] = A[T] * K[T]^α # Final output
return K, Y, A
end

# Simulate the model
K, Y, A = simulate_stochastic_solow(rng_gen, T, α, δ, s, n, g, σ, K₀, A₀)

# Plot the results
f = Figure()
axis = Axis(
    f[1, 1],
    xlabel="Time",
    ylabel="Output",
    title="Stochastic Solow Growth Model"
)
scatter!(1:T, Y, label="Output (Y)")
f

```



15.4.3.2. Other Examples

Stochastic examples can be found in many other sections of this book, such as:

- Portfolio optimization applications (Chapter 27).
- Genetic algorithms for certain optimization problems (Chapter 17).
- Stochastic state-based mortality projections (Chapter 30).

15.5. Pseudo-Random Number Generators

Modern computers use pseudo-random number generators (PRNGs)—algorithms that produce sequences of numbers that appear random but are actually determined by an initial seed. Given the same seed, you get the same sequence. PRNGs also have a finite period: after enough generated values, the sequence repeats. Choose PRNGs with periods that are long enough for your application.

Financial modelers should understand how PRNGs work because many financial models rely on Monte Carlo simulations, risk analysis, and other stochastic modeling techniques that require random sampling. A good PRNG is essential for robust financial modeling. Choosing the right PRNG ensures results with high statistical quality and efficiency, which are critical when making financial decisions. The ability to specify and control the seed of a random number generator is a fundamental requirement in computational modeling, as it enables exact reproducibility of results. By using a fixed seed, simulations can be rerun with identical random sequences, allowing researchers and stakeholders to verify outcomes, conduct consistent sensitivity analyses, and perform rigorous debugging. This reproducibility is critical for transparency, auditability, and compliance with regulatory standards in fields such as quantitative finance, risk management, and scientific computing. Without a seedable PRNG, the inherent randomness of simulations would preclude the possibility of replicating results precisely, undermining confidence in the validity and reliability of the modeling process.

Different choices of random number generators typically have low impact on the mean outputs of stochastic models, since most generators are designed to approximate uniform distributions adequately. However, they can have substantial effects on risk measures such as standard deviation, Value-at-Risk (VaR), and tail quantiles, particularly when the PRNG has a short period or exhibits hidden correlations. In such cases, the variability and uncertainty estimates around risk measures can be severely biased or understated, leading to misleading conclusions about the model's risk profile. For this reason, it is essential to rerun the stochastic model multiple times with different seeds—or even different PRNGs—to test the stability and convergence of the estimated risk measures and ensure that the results are robust to the choice of random number source.

15.5.1. Common PRNGs

15.5.1.1. Mersenne Twister

For many years, the Mersenne Twister was a standard and highly recommended PRNG for financial modeling purposes. One of its greatest strengths is its exceptionally long period of $2^{19937} - 1$, which is crucial for applications requiring a large number of independent random numbers. It is also known for its good statistical properties, passing many standard tests for randomness. Moreover, it was designed with features for creating multiple streams, though ensuring their statistical independence in parallel applications requires careful management.

15.5.1.2. Xorshift

Xorshift is a family of PRNGs known for their simplicity and extremely fast operation. The name “Xorshift” comes from the bitwise XOR (exclusive or) and bit-shifting operations that are the core of the algorithm. Xorshift generators are often used in applications where speed is a priority and cryptographic-strength randomness is not a strict requirement. One of the main advantages of xorshift is that its core operations can be efficiently implemented in hardware. However, a typical xorshift generator has a relatively short period compared to the Mersenne Twister.

15.5.1.3. Xoshiro

Xoshiro is a family of high-performance PRNGs with excellent statistical properties. The name “Xoshiro” is a portmanteau of the core operations it uses: XOR, shift, and rotate. Xoshiro algorithms, including the highly-regarded **Xoshiro256++** variant, use a combination of bitwise XOR, bit-shifting, and addition/rotation operations. They generally have more complex update rules and longer periods than basic Xorshift algorithms.

When selecting seeds at random to initialize multiple instances of the Mersenne Twister generator, there is a significant likelihood of producing streams that are statistically correlated in parallel computations. This arises because the default seeding mechanism may not sufficiently de-correlate the internal states across different instances. In contrast, generators such as Xoshiro256++ offer markedly improved suitability for parallel environments, with carefully managed methodologies for ensuring stream independence.

15.5.1.4. Comparing PRNGs

15. Stochastic Modeling

| Generator | Typical Period | Relative Speed | Parallel Safety |
|------------------|--------------------------------|----------------|-----------------------------|
| Mersenne Twister | Very Long
$(2^{19937} - 1)$ | Good | Poor (risk of correlation) |
| Xorshift | Shorter | Very Fast | Poor (not designed for it) |
| Xoshiro256++ | Very Long ($2^{256} - 1$) | Excellent | Excellent (designed for it) |

Note

Since Julia 1.7, the default random number generator for the language has been Xoshiro256++, which replaced an implementation of Mersenne Twister.

Importantly, Julia's implementation of Xoshiro256++ is thread-safe (meaning you can use the RNG in multiple threads without losing RNG quality).

15.5.2. Consistent Interface

Julia offers a consistent interface for random numbers due to its design and multiple dispatch principles. Consider the following random numbers in different data types.

```
using Random

rng = MersenneTwister(1234)
rand(rng, Int, (2, 3))

2×3 Matrix{Int64}:
7022509616038853369  -483142612144439943  -9189440298257092269
4512210966853297707  3016561889232165636  -2139829978692188285

using Random

rng = MersenneTwister(1234)
rand(rng, Float64, (2, 3))

2×3 Matrix{Float64}:
0.538321  0.0275419  0.115857
0.997355  0.0703632  0.905544

using Random

rng = Xoshiro(1234)
rand(rng, Bool, (2, 3))

2×3 Matrix{Bool}:
1  0  1
0  1  1
```

Passing the RNG explicitly is a good habit; it guarantees reproducibility, keeps multi-threaded simulations thread-safe, and makes it trivial to swap PRNG families when performing robustness checks.

15.6. Conclusion

Stochastic modeling studies systems influenced by random factors and uncertainty. By combining random variables, probability distributions, and processes like Markov chains or Brownian motion, these models provide insights into systems that can't be described deterministically.

15. Stochastic Modeling

16. Automatic Differentiation

CHAPTER AUTHORED BY: ALEC LOUDENBACK

One machine can do the work of fifty ordinary men. No machine can do the work of one extraordinary man. - Elbert Hubbard, *Thousand and One Epigrams* (1911)

16.1. Chapter Overview

Harnessing the chain rule to compute derivatives not just of simple functions, but of complex programs.

16.2. Motivation for (Automatic) Derivatives

Derivatives are one of the most useful analytical tools we have. Determining the rate of change with respect to an input is effectively sensitivity testing. Knowing the derivative lets you optimize things faster (see Chapter 17). You can test properties and implications (monotonicity, maxima/minima). These applications make derivatives foundational for machine learning, generative models, scientific computing, and more.

We will work up concepts on computing derivatives, from the most basic (finite differentiation) to automatic differentiation (forward mode and then reverse mode). These tools can be used in many modeling applications. Automatic Differentiation ("AD" or "autodiff") refers to innovative techniques to get computers to perform differentiation of complex algorithms (code) that would be intractable for a human to perform manually.

16.3. Finite Differentiation

Finite differentiation is evaluating a function $f(x)$ at a value x and then at a nearby value $x + \epsilon$. The line drawn through these two points effectively estimates the line that is tangent to the function f at x - effectively the derivative has been found by approximation. That is, we are looking to approximate the derivative using the property:

$$f'(x) = \lim_{\epsilon \rightarrow 0} \frac{f(x_0 + \epsilon) - f(x_0)}{\epsilon}$$

We can approximate the result by simply choosing a small ϵ .

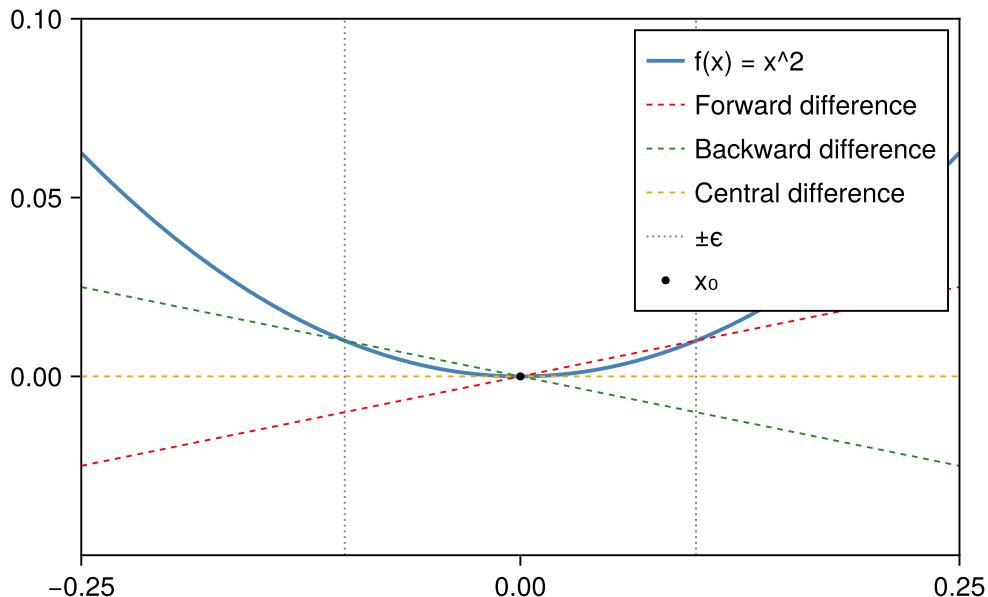
There are three common finite-difference formulas to be aware of:

16. Automatic Differentiation

Table 16.1.: Three different methods for calculating a finite difference.

| Method | Notes |
|----------------------------|--|
| Forward difference | $\frac{f(x_0 + \epsilon) - f(x_0)}{\epsilon}$, where the step is taken above x_0 . |
| Backward difference | $\frac{f(x_0) - f(x_0 - \epsilon)}{\epsilon}$, where the step is taken below x_0 . |
| Central difference | $\frac{f(x_0 + \epsilon) - f(x_0 - \epsilon)}{2\epsilon}$, which averages the two one-sided slopes. |

The central difference often balances the information from both sides of the evaluation point and avoids some issues in areas where the derivative is changing rapidly. The plot below compares the three estimates for $f(x) = x^2$ around $x_0 = 0$:



Central differences usually beat forward or backward differences in accuracy, but the extra evaluation can matter when the underlying function is expensive. Take, for example, the process of optimizing a function to find a maximum or minimum.

Maximum-finding algorithms usually involve guessing an initial point, evaluating the function at that point, and determining what the derivative of the function is at that point. Both items are used to update the guess to one that's closer to the solution. This approach is used in many optimization algorithms such as Newton's Method. At each step you need to evaluate the function three times: at x , $x + \epsilon$, and $x - \epsilon$. With forward (or backward) finite differences, you can often reuse the prior evaluation at x from the previous iteration, thereby saving an expensive computation.

There are additional challenges with the finite differentiation method. In practice, we are often interested in much more complex functions than x^2 . For example, we may actually be interested in the sum of a series that is many elements long or contains more complex operations than basic

algebra. In the prior example, the ϵ is set unusually wide for demonstration purposes. As ϵ grows smaller, the accuracy of all three finite difference methods generally improves. However, it is not always the case: for sufficiently small ϵ the subtraction $f(x_0 + \epsilon) - f(x_0)$ can lose significant digits due to floating-point round-off, corrupting the derivative estimate.

To demonstrate, here is a more complex example using an arbitrary function

$$f(x) = \exp(x)$$

for this example we'll show the results of the three methods calculated at different values of ϵ :

```

f(x) = exp(x)
ε = 10 .^ (range(-16, stop=0, length=100))
x₀ = 1
estimate = @. (f(x₀ + ε) - f(x₀ - ε)) / 2ε
actual = f(x₀)

fig = Figure()
ax = Axis(fig[1, 1], xscale=log10, yscale=log10, xlabel="ε", ylabel="absolute
    ↵ error")
scatter!(ax, ε, abs.(estimate .- actual))
fig

```

①

- ① The derivative of $f(x) = \exp(x)$ is itself. That is $f'(x) = f(x)$ in this special case.

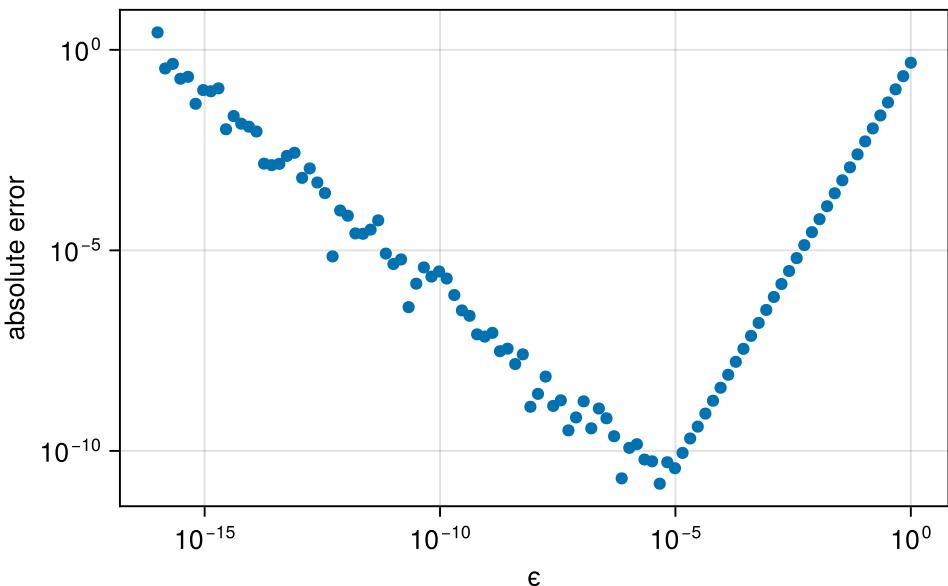


Figure 16.1.: A log-log plot showing the absolute error of the finite differences. Further to the left, roundoff error dominates while further to the right, truncation error dominates.

i Note

The `@.` in the code example above is a macro that broadcasts each function call to its right. `(f(x0 + ε) - f(x0 - ε)) / 2ε` is the same as `(f.(x0 .+ ε) -. f.(x0 .- ε)) ./ (2 .* ε)`

A few observations:

1. At virtually every value of ϵ we observe some error from the true derivative.
2. That error is the sum of two parts:
 1. **Truncation error** is inherent in that we are using a given non-zero value for ϵ and not determining the limiting analytic value as $\epsilon \rightarrow 0$. The larger ϵ is, the larger the truncation error.
 2. **Roundoff error**, which arises due to the limited precision of floating point math.

The implications of this are that we need to often be careful about the choice of ϵ , as the optimal choice will vary depending on the function and the point we are attempting to evaluate. This presents a number of practical difficulties in various algorithms.

Additionally, when computing the finite difference we must evaluate the function multiple times to determine a single estimate of the derivative. When performing something like optimization, the process typically involves iteratively making many guesses requiring many evaluations of the approximate derivative. Further, the efficiency of the algorithm usually depends on the accuracy of computing the derivative!

Despite the accuracy and computational overhead, finite differences can be very useful in many circumstances. However, a more appealing alternative approach will be covered next.

16.4. Automatic Differentiation

Automatic differentiation is essentially the practice of defining algorithmically what the derivatives of a function should be. We are able to do this through a creative application of the chain rule. Recall that the **chain rule** allows us to compute the derivative of a composite function using the derivatives of the component functions:

$$\begin{aligned} h(x) &= f(g(x)) \\ h'(x) &= f'(g(x))g'(x) \end{aligned}$$

Using this rule, we can define how elementary operations act when differentiated. Combined with the fact that most computer code is built up from a bunch of elementary operations, we can get a very long way in differentiating complex functions.

16.4.1. Dual Numbers

To understand where we are going, let's remind ourselves about complex numbers. Complex numbers have a form which has a real part (r) and an imaginary part (iq):

$$r + iq$$

By definition we say that $i^2 = -1$. This is useful because it allows us to perform certain types of operations (e.g. finding a square root of a negative number) that are otherwise unsolvable with just

the real numbers¹. After defining how the normal algebraic operations (addition, multiplication, etc.) work for the imaginary number, we are able to utilize the imaginary numbers for a variety of practical mathematical tasks.

What is meant by extending the algebraic operations for imaginary numbers? For example, stating how addition should work for imaginary numbers:

$$(r + iq) + (s + iu) = (r + s) + i(q + u)$$

In a similar fashion as extending the Real (\mathbb{R}) numbers with an *imaginary* part, for automatic differentiation we will extend them with a *dual* part. A **dual number** is one of the form:

$$a + \epsilon b$$

Where $\epsilon^2 = 0$ and $\epsilon \neq 0$ by definition. While a represents the function value, b carries its derivative. An example should make this clearer. First let's define a DualNumber:

```
struct DualNumber{T,U}
    a::T
    b::U
    function DualNumber(a::T, b::U=zero(a)) where {T,U}
        return new{T,U}(a, b)
    end
end
```

- ① We define this type parametrically to handle all sorts of `<:Real` types and allow `a` and `b` to have different types in case a mathematical operation causes a type change (e.g. as in the case of integers becoming a floating point number like `10/4 == 2.5`)
- ② In the constructor, we set the default value of `b` to be `zero(a)`. `zero(a)` is a generic way to create a value equal to zero with the same type of the argument `a`. E.g. `zero(12.0) == 0.0` and `zero(12) == 0`.

Now let's define how dual numbers work under addition. The mathematical rule is:

$$(a + \epsilon b) + (c + \epsilon d) = (a + c) + (b + d)\epsilon$$

We then need to define how it works for the combinations of numbers that we might receive as arguments to our function (this is an example where multiple dispatch greatly simplifies the code compared to object oriented single dispatch!):

```
Base.:+(d::DualNumber, e::DualNumber) = DualNumber(d.a + e.a, d.b + e.b)
Base.:+(d::DualNumber, x) = DualNumber(d.a + x, d.b)
Base.:+(x, d::DualNumber) = d + x
```

And here's how we would get the derivative of a very simple function:

```
f1(x) = 5 + x
f1(DualNumber(10, 1))
```

```
DualNumber{Int64, Int64}(15, 1)
```

¹Richard Feynman has a wonderful, short lecture on algebra here: https://www.feynmanlectures.caltech.edu/I_22.html

16. Automatic Differentiation

That's not super interesting though - the derivative of f_1 is just 1 and we supplied that in the construction of `DualNumber`. We did at least prove that we can add the 10 and 5!

Let's make this more interesting by also defining the multiplication operation on dual numbers. We'll follow the product rule:

$$(uv)' = u'v + uv'$$

```
Base.*(d::DualNumber, e::DualNumber) = DualNumber(d.a * e.a, d.b * e.a + d.a
    ↵ * e.b)
Base.*(x, d::DualNumber) = DualNumber(d.a * x, d.b * x)
Base.*(d::DualNumber, x) = x * d
```

Now what if we evaluate this function:

```
f2(x) = 5 + 3x
f2(DualNumber(10, 1))

DualNumber{Int64, Int64}(35, 3)
```

We have found that the second component is 3, which is indeed the derivative of $5 + 3x$ with respect to x . And in the first part we have the value of f_2 evaluated at 10.

Note

When calculating the derivative, why do we start with 1 in the dual part of the number? Because the derivative of a variable with respect to itself is 1. From this unitary starting point, the various operations applied accumulate the derivative of the various operations in the b part of $a + \epsilon b$.

We can also define this for things like transcendental functions:

```
Base.exp(d::DualNumber) = DualNumber(exp(d.a), exp(d.a) * d.b)
Base.sin(d::DualNumber) = DualNumber(sin(d.a), cos(d.a) * d.b)
Base.cos(d::DualNumber) = DualNumber(cos(d.a), -sin(d.a) * d.b)
exp(DualNumber(1, 1))

DualNumber{Float64, Float64}(2.718281828459045, 2.718281828459045)

sin(DualNumber(0, 1))

DualNumber{Float64, Float64}(0.0, 1.0)

cos(DualNumber(0, 1))

DualNumber{Float64, Float64}(1.0, -0.0)
```

And finally, to put it all together in a more usable wrapper, we can define a function which will calculate the derivative of another function at a certain point. This function applies f to an initialized `DualNumber` and then returns the b component from the result:

```
derivative(f, x) = f(DualNumber(x, one(x))).b
derivative (generic function with 1 method)
```

And then evaluating it on a more complex function like $f(x) = 5e^{\sin(x)} + 3x$ at $x = 0$, we would analytically derive 8, which matches what we calculate next:

```
f3(x) = 5 * exp(sin(x)) + 3x
derivative(f3, 0)
```

8.0

We have demonstrated that through the clever use of dual numbers and the chain rule that complex expressions can be automatically differentiated by a computer to an exact level, limited only by the same machine precision that applies to our primary function of interest as well.

💡 Tip

The demonstration above re-implements dual numbers for pedagogical purposes. In production code you would typically rely on `ForwardDiff.Dual` (from `ForwardDiff.jl`), which already provides these algebraic rules along with extensive coverage of transcendental functions.

Libraries exist (such as `ChainRules.jl`) which define large numbers of predefined rules for many more operations, even beyond basic algebraic functions. This allows complex programs to be differentiated automatically.

16.5. Performance of Automatic Differentiation

Recall that in the finite difference method, we generally had to evaluate the function two or three times to *approximate* the derivative. Here we have a single function call that provides both the value and the derivative at that value. How does this compare performance-wise to simply evaluating the function a single time? Let's check how long it takes to compute a `Float64` versus a `DualNumber`:

```
using BenchmarkTools
@btime f3(x) setup = x = rand()

4.833 ns (0 allocations: 0 bytes)

5.595846563630646

@btime f3(DualNumber(x, 1)) setup = x = rand()

9.467 ns (0 allocations: 0 bytes)

DualNumber{Float64, Float64}(14.12462805659582, 9.562103591186982)
```

The dual number version takes somewhat longer than the plain function evaluation — but for this additional cost, we get both the function value *and* its exact derivative. This is typically less than twice the cost of the base evaluation alone. Compare this to finite differences, which require two or three separate function evaluations just to *approximate* the derivative. As the function gets more complex, the overhead does increase but is still generally preferred over finite differentiation. This advantage becomes more pronounced as we contemplate derivatives with respect to many variables at once or for higher-order derivatives.

i Note

In fact, it's largely due to the advances in applications of automatic differentiation that have led to the explosion of machine learning and artificial intelligence techniques in the 2010s/2020s. The "learning" process relies on solving parameter weights and would be too computationally expensive if using finite differences.

These applications of AD in specialized C++ libraries underpin the libraries like PyTorch, TensorFlow, and Keras. These libraries specialize in allowing for AD on a limited subset of operations. Julia's available AD libraries are more general and can be applied to many more scenarios.

16.6. Automatic Differentiation in Practice

We have, of course, not defined an exhaustive list of operations, covering only `+`, `*`, `exp`, `sin`, and `cos`. There are only a few more arithmetic (`-`, `/`) and transcendental (`log`, more trigonometric functions, etc.) before we would have a very robust set of algebraic operations defined for our `DualNumber`. In fact, it's possible to go even further and to define the behavior through conditional expressions and iterations to differentiate fairly complex functions or to extend the mechanism to partial derivatives and higher-order derivatives as well.

```
import Distributions
import ForwardDiff

N(x) = Distributions.cdf(Distributions.Normal(), x)

function d1(S, K, τ, r, σ, q)
    return (log(S / K) + (r - q + σ^2 / 2) * τ) / (σ * √(τ))
end

function d2(S, K, τ, r, σ, q)
    return d1(S, K, τ, r, σ, q) - σ * √(τ)
end

"""
eurocall(parameters)

Calculate the Black-Scholes implied option price
for a European call where 'parameters' is a vector
with the following six elements:

- 'S' is the current asset price
- 'K' is the strike or exercise price
- 'τ' is the time remaining to maturity (can be typed with \\tau[tab])
- 'r' is the continuously compounded risk free rate
- 'σ' is the (implied) volatility (can be typed with \\sigma[tab])
- 'q' is the continuously paid dividend rate
"""

function eurocall(parameters)
    S, K, τ, r, σ, q = parameters
    iszero(τ) && return max(zero(S), S - K)
    d1 = d1(S, K, τ, r, σ, q)
```

①

```

d2 = d2(S, K, τ, r, σ, q)
return S * exp(-q * τ) * N(d1) - K * exp(-r * τ) * N(d2)
end

```

- ① We put the various variables inside a single parameters vector to allow calling a single gradient call instead of multiple derivative calls for each parameter.

Main.Notebook.eurocall

```

S = 1.0
K = 1.0
τ = 30 / 365
r = 0.05
σ = 0.2
q = 0.0
params = [S, K, τ, r, σ, q]
eurocall(params)

```

0.024933768194037365

💡 Tip

Some terminology in differentiation:

- **Scalar-valued function:** A function whose output is a single scalar.
- **Vector-valued (array-valued) function:** A function whose output is a vector or array.
- **Derivative (or partial derivative):** The (instantaneous) rate of change of the output with respect to one input variable. In a multivariate context, these are partial derivatives, e.g., $\frac{\partial}{\partial x} f(x, y, z)$.
- **Gradient:** For a scalar-valued function of several variables, the gradient is the vector of all first partial derivatives, e.g. $\nabla f(x, y, z) = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right]^T$.
- **Jacobian:** For a vector-valued function, the Jacobian is the matrix of first partial derivatives. If $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, its Jacobian is an $m \times n$ matrix.
- **Hessian:** For a scalar-valued function of several variables, the Hessian is the matrix of all second partial derivatives (i.e., it's an $n \times n$ matrix).

With the above code, now we can get the partial derivatives with respect to each parameter. The first, third, fourth, fifth, and sixth correspond to the common “Greeks” *delta*, *theta*, *rho*, *vega*, and *phi* respectively. The second term is the partial derivative with respect to the strike price:

```

ForwardDiff.gradient(eurocall, params)

6-element Vector{Float64}:
 0.5399635456230847
 -0.515029774290475
 0.16420676980838977
 0.04233121458320943
 0.11379886104405815
 -0.044380565393678295

```

16. Automatic Differentiation

- Entry 1: delta $\partial C / \partial S$
- Entry 2: sensitivity to strike $\partial C / \partial K$ (no standard Greek)
- Entry 3: $\partial C / \partial \tau$, which is the negative of the market convention theta ($\Theta = \partial C / \partial t = -\partial C / \partial \tau$)
- Entry 4: rho $\partial C / \partial r$
- Entry 5: vega $\partial C / \partial \sigma$
- Entry 6: dividend rho (often denoted phi) $\partial C / \partial q$

We can also get the second order Greeks with another call. This includes many uncommon second order partial derivatives, but the popular *gamma* is in the [1,1] position for example:

```
ForwardDiff.hessian(eurocall, params)
```

```
6x6 Matrix{Float64}:
 6.92276   -6.92276    0.242297   0.568994   -0.0853491   -0.613375
 -6.92276    6.92276   -0.07809   -0.526663    0.199148    0.568994
 0.242297   -0.07809   -0.846846    0.521448    0.685306   -0.559878
 0.568994   -0.526663    0.521448    0.0432874   -0.0163683   -0.0467667
 -0.0853491    0.199148    0.685306   -0.0163683    0.00245525    0.007015
 -0.613375    0.568994   -0.559878   -0.0467667    0.007015    0.0504144
```

i Note

Tip from practice: desks that run daily risk charge calculations often rely on AD to refresh Greeks after every market data cut. A forward finite-difference grid of six inputs would require at least seven full revaluations of the pricing stack. With AD you obtain the same sensitivities (and, if needed, the Hessian) in a single pass, freeing up time for scenario analysis and stress testing.

16.6.1. Performance

Earlier we assessed the impact on performance for the derivatives using `DualNumber` on a very basic function. What about if we take a more realistic example like `eurocall`?

```
@btime eurocall($params)
17.034 ns (0 allocations: 0 bytes)
0.024933768194037365

let
    g = similar(params)
    cfg = ForwardDiff.GradientConfig(eurocall, params)
    @btime ForwardDiff.gradient!($g, eurocall, $params, $cfg)
end
```

- ① Pre-allocate the output array so the benchmark measures only the gradient computation.
② Pre-allocate the `GradientConfig` to avoid `ForwardDiff` re-creating internal buffers on each call — without this, the overhead is dominated by allocations rather than actual derivative computation.

```
98.509 ns (0 allocations: 0 bytes)
```

```
6-element Vector{Float64}:
0.5399635456230847
-0.5150297774290475
0.16420676980838977
0.04233121458320943
0.11379886104405815
-0.044380565393678295
```

We can compare this against a naïve finite-difference gradient that requires seven evaluations of the pricing function (one base evaluation plus one per input):

```
function fd_gradient!(g, f, x; h=1e-8)
    fx = f(x)
    for i in eachindex(x)
        xi = x[i]
        x[i] = xi + h
        g[i] = (f(x) - fx) / h
        x[i] = xi
    end
    g
end

let
    g = similar(params)
    @btime fd_gradient!($g, eurocall, $params)
end
```

128.941 ns (0 allocations: 0 bytes)

```
6-element Vector{Float64}:
0.5399635849556716
-0.5150297410771998
0.1642067704032968
0.042331216310032005
0.11379887032703095
-0.044380565888957335
```

Forward-mode AD computes all six first-order derivatives faster than finite differences and returns exact results — no step-size tuning and no truncation error (recall Figure 16.1).

16.7. Forward Mode and Reverse Mode

The approach outlined above is forward-mode AD, where derivatives are propagated alongside values. Reverse-mode AD evaluates the function, records a computation graph, and then accumulates sensitivities backwards.

Reverse mode requires more book-keeping because the derivative needs to be carried backwards through the computation graph, rather than being propagated forward as with the `DualNumber` approach.

16.8. Practical Tips for Automatic Differentiation

Here are a few practical tips to keep in mind.

16.8.1. Choosing between Reverse Mode and Forward Mode

Forward mode is more efficient when the number of inputs is small relative to the number of outputs, because each forward pass computes the derivative with respect to one input. Reverse mode is more efficient when the number of outputs is small relative to the number of inputs, because each reverse pass computes the derivative with respect to one output. Examples of reverse mode being preferred include loss functions in statistical analysis where many features or parameters are used to predict a single outcome variable.

- Use forward mode when the number of inputs is small relative to the number of outputs.
- Use reverse mode when the number of inputs is large and the number of outputs is small (e.g., loss functions).

16.8.2. Mutation

Auto-differentiation works through most code, but a particularly tricky part to get right is when values within arrays are mutated (changed). It's possible to do so but may require a little bit more boilerplate to set up. Enzyme.jl has the best support for functions with mutation inside of them.

16.8.3. Custom Rules

Custom rules for new or unusual functions can be defined, but this is an area that should be explored equipped with a bit of calculus and a deeper understanding of both forward-mode and reverse-mode. ChainRules.jl provides an interface for defining additional rules that hook into the AD infrastructure in Julia as well as provide a good set of documentation on how to extend the rules for your custom function.

16.8.4. Available Libraries in Julia

- **ForwardDiff.jl** provides robust forward-mode AD.
- **Zygote.jl** is a reverse-mode package with the innovations of being able to differentiate structs in addition to arrays and scalars.
- **Enzyme.jl** is a newer package which allows for both forward and reverse mode, but has the advantage of supporting array mutation. Additionally, Enzyme works at the level of LLVM code (an intermediate level between high level Julia code and machine code) which allows for different, sometimes better, optimizations.
- **DifferentiationInterface.jl** is a wrapper library providing a consistent API while being able to exchange different backends.

In the authors' experience, they would probably recommend DifferentiationInterface.jl as a starting point, and diving into specific libraries if certain features are needed.

16.9. References

- Rackauckas (2020a)
- Center (2019)

17. Optimization

CHAPTER AUTHORED BY: ALEC LOUDENBACK, YUN-TIEN LEE

Far better an approximate answer to the right question, which is often vague, than an exact answer to the wrong question, which can always be made precise. - John Tukey, 1962

17.1. Chapter Overview

Optimization lies at the heart of every quantitative finance workflow: calibrating pricing models, allocating portfolios, fitting yield curves, or matching liability cash flows all boil down to choosing inputs that minimize (or maximize) a well-defined metric. This chapter surveys the objective functions we typically optimize, contrasts gradient-based and gradient-free techniques, and illustrates how Julia's differentiable programming ecosystem accelerates financial calibration and model fitting.

17.2. Introduction

Local and global optimization: A **local optimum** value refers to a solution where the objective function (or cost function) has the best possible value in a neighborhood surrounding that solution. A **global optimum** value, on the other hand, is the best possible value of the objective function across the entire feasible domain. For smooth and convex functions, any local minimum is also a global minimum, and the negative gradient always points in a descent direction, making gradient-based methods extremely efficient for finding the optimal solution. Even for non-convex functions, the negative gradient provides valuable information about the direction to move toward improving the objective function value locally.

```
using CairoMakie
let
    f(x) = x^8 - 3x^4 + x
    xs = range(-1.5, 1.5; length=1001)
    ys = f.(xs)

    fig = Figure()
    ax = Axis(fig[1, 1],
              xlabel="x",
              ylabel="f(x)",
              title="Function with Local and Global Optima (over [-1.5, 1.5])",
              limits=(-1.5, 1.5, minimum(ys) - 0.5, maximum(ys) + 0.5),
    )
    lines!(ax, xs, ys, color=:blue)

    # Global minimum on the sampled grid
```

17. Optimization

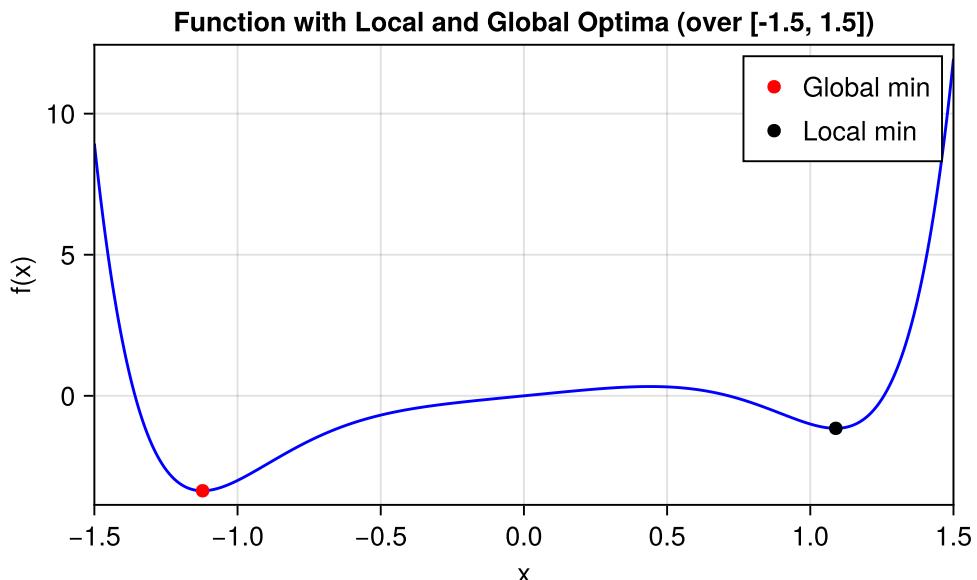
```

i_global = argmin(ys)
x_global_min = xs[i_global]
y_global_min = ys[i_global]
scatter!(ax, [x_global_min], [y_global_min],
    color=:red, markersize=10, label="Global min")

# Detect local minima by comparing neighbors (simple discrete test)
left_lower = ys[2:end-1] .< ys[1:end-2]
right_lower = ys[2:end-1] .< ys[3:end]
local_min_inds = findall(left_lower .& right_lower) .+ 1
# Exclude the global index to highlight a different local minimum if
# present
local_min_inds = filter(i -> i != i_global, local_min_inds)
if !isempty(local_min_inds)
    i_local = first(local_min_inds)
    scatter!(ax, [xs[i_local]], [ys[i_local]],
        color=:black, markersize=10, label="Local min")
end

axislegend(ax, position=:rt)
fig
end

```



Optimization techniques are incredibly important and find uses in many areas:

- Machine learning relies on “training”, which is essentially optimizing parameters to match target data.
- Determining parameters for market consistent models where the modeled price is fit to market-observed prices.
- Minimizing a certain risk outcome by optimizing asset allocations.
- Maximizing risk-adjusted yield in a portfolio.

We will introduce some basic concepts and categories of optimization techniques in this chapter:

- Objective and Loss Functions
- Optimization Techniques
 - Nonlinear Optimization
 - * Gradient Based Techniques
 - * Bracketing Methods
 - * Other Techniques
 - Linear Optimization

17.3. Objective and Loss Functions

Any optimization algorithm needs to know what is being optimized. We call this function the **objective function**. Usually, the objective function is a function we want to minimize — though maximizing a function is mathematically equivalent to minimizing its negative. Concretely, if you want to maximize $f(x)$, you can minimize $-f(x)$. The **optimal point**, or **argmin** (**argmax**) is simply the input value that produces the smallest (largest) objective value.

In many practical situations — especially those with noisy or uncertain data — the function we want to optimize is called a **loss function**. This term is common in statistical modeling and machine learning, where the aim is to measure how far a model's predictions deviate from real observations. For instance, the following squared loss function computes the sum of squared errors between a model's predictions and the targets:

```
loss(f, guess, target) = sum((f.(guess) .- target).^2)
```

Here, `guess` might represent model parameters (or input values), and `target` is the observed data. Minimizing this loss aligns the model's predictions with reality as closely as possible.

17.4. Optimization Techniques

17.4.1. Nonlinear Optimization

Nonlinear optimization refers to problems where the relationship between the inputs and outputs is not constrained to be a linear relationship, which is incredibly common in financial modeling.

In many financial settings, variables must obey certain bounds or relationships (e.g., portfolios cannot exceed a total budget, a particular risk measure must remain below a threshold). Handling constraints often involves methods like Lagrange multipliers, projected gradient, or analytic penalty functions. We won't cover these in depth here, but be aware that most general optimization libraries can handle constraints in nonlinear problems as well.

17.4.1.1. Gradient-Based Optimization

Gradient-based optimization algorithms tend to utilize the gradient (i.e., multivariable derivative) in order to make the optimization substantially more efficient. The gradient is useful because it can tell you if you are at a stationary point (when the derivative is zero, the function may be at a maximum, minimum, or saddle point), but also because algorithms can be smarter about searching for a solution using the additional information.

17. Optimization

The **gradient** provides the direction of the steepest ascent of a function. Optimization algorithms often iteratively update parameters in the direction opposite to the gradient (for minimization problems), which tends to converge towards a local minimum (or maximum for maximization problems). Besides, computing the gradient is often computationally feasible and relatively inexpensive compared to other methods for determining function behavior, such as higher-order derivatives or finite-difference methods. Beyond just the direction, the magnitude (or norm) of the gradient also indicates how steep the function change is in that direction. This information is used to adjust step sizes in optimization algorithms, balancing between convergence speed and stability.

Calculating gradients in the context of computer algorithms is discussed at length in Chapter 16. Here is a quick recap of the main approaches to computing derivatives:

- Finite differences: evaluate the function at nearby points and determine the rates of change associated with each change in direction.
- Analytic derivatives: A human-derived or computer tool (such as Mathematica) is able to analytically determine a derivative that is coded into the optimization algorithm.
- Automatic differentiation (AD): elementary rules are applied to decompose elementary code operations into derivatives, allowing for very efficient computation of complex algorithms.

To compare the approaches, here is an example of determining the derivative of a simple function at a certain point:

```
using Zygote

# Define a differentiable function
f(x) = 3x^2 + 2x + 1
# Define an input value
x = 2.0
h = 1e-3

finite_diff = (f(x + h) - f(x - h)) / 2h
auto_diff = gradient(f, x)[1]
analytic_diff = 6x + 2

println("Value of f(x) at x=", f(x))
println("Gradient (finite diff) at x=", finite_diff)
println("Gradient (auto diff) at x=", auto_diff)
println("Gradient (analytic) at x=", analytic_diff)
```

```
Value of f(x) at x=17.0
Gradient (finite diff) at x=13.9999999999757
Gradient (auto diff) at x=14.0
Gradient (analytic) at x=14.0
```

1. `finite_diff` uses finite differences. It needs multiple calls to `f` and is sensitive to the choice of `h` (too small amplifies round-off, too large smears the slope).
2. `auto_diff` comes from automatic differentiation and matches machine precision for smooth code without symbolic work.
3. `analytic_diff` is a closed-form derivative. When the algebra is manageable this remains the gold standard, but AD gives similar accuracy with far less manual effort.

17.4.1.1. Root finding

Root finding, also known as root approximation or root isolation, is the process of finding the values of the independent variable (usually denoted as x) for which a given function equals zero.

In mathematical terms, if we have a function $f(x)$, root finding involves finding values of x such that $f(x) = 0$.

There are various algorithms for root finding, each with its own advantages and disadvantages depending on the characteristics of the function and the requirements of the problem. One notable approach is Newton's method, an iterative method that uses the derivative or gradient of the function to approximate the root with increasing accuracy in each iteration.

We will again use a simple function to illustrate the process:

```
using Zygote

# Define a differentiable function
f(x) = 2x^2 - 3x + 1
# Define an initial value
x = 0.0
# tolerance of difference in value
tol = 1e-6
# maximum number of iteration of the algorithm
max_iter = 1000
iter = 0
while abs(f(x)) > tol && iter < max_iter
    x -= f(x) / gradient(f, x)[1]
    iter += 1
end
if iter == max_iter
    println("Warning: Maximum number of iterations reached.")
else
    println("Root found after ", iter, " iterations.")
end
print("Approximate root: ", x)
```

Root found after 5 iterations.
Approximate root: 0.499999998835846

💡 Tip

Newton's method converges quadratically near a simple root, but it can diverge if the initial guess is far away or if the derivative becomes very small. In practice we often bracket the solution first (to guarantee a sign change) and then switch to Newton once we are in a safe neighborhood.

Although it might look different, **root-finding** (i.e. finding x such that $g(x) = 0$) can be cast as a minimization problem by defining an objective function such as $g(x)^2$. In this view, driving $g(x)^2$ to zero compels $g(x)$ itself to be zero, so the methods and algorithmic ideas from minimization apply naturally to root-finding scenarios as well.

17.4.1.1.2. BFGS

BFGS — named for Broyden, Fletcher, Goldfarb, and Shanno — is a popular member of the quasi-Newton family of optimization algorithms. Although it does require first-order gradient information, BFGS does not need the exact **Hessian** (i.e., second derivatives). Instead, it updates an approximation to the inverse Hessian at each step using the gradients from previous iterations. This extra curvature information allows BFGS to converge much faster than steepest descent in practice. Moreover, because it never explicitly forms the full Hessian matrix, it remains efficient

17. Optimization

for moderately sized problems. In finance and actuarial settings, BFGS can be especially useful for model calibration or parameter estimation tasks where one needs to handle nonlinear functions relatively quickly but cannot afford the computational overhead of second derivatives.

The following example uses the `Optim` package available in Julia to do the BFGS optimization. We don't illustrate the complete algorithm as it is a bit longer, but wanted to ensure that this workhorse of an algorithm was mentioned.

```
using Optim

# Define the objective function to minimize
function objective_function(x)
    return sum(abs2, x)
end

# Initial guess for the minimization
initial_x = [1.0]
# Perform optimization using BFGS method
result = optimize(objective_function, initial_x, BFGS())
# Extract the optimized solution
solution = result.minimizer
minimum_value = result.minimum

# Print the result
println("Optimized solution: x = ", solution)
println("Minimum value found: ", minimum_value)
```

Optimized solution: x = [-6.359357485052897e-13]
Minimum value found: 4.0441427622698303e-25

17.4.1.2. Gradient-Free Optimization

This category includes algorithms that do not rely on gradients or derivative information. They often explore the objective function using heuristics or other types of probes to guide the search.

17.4.1.2.1. Bracketing Methods

A bracketed search algorithm is a technique used in optimization and numerical methods to confine or “bracket” a minimum or maximum of a function within a specified interval. The primary goal is to reduce the search space systematically until a satisfactory solution or range containing the optimal value is found.

For 1D minimization with bracketing, golden-section search and Brent’s method are standard. The bisection code below is for root finding, not minimization.

```
"""
bisection_method(f, a, b; tol=1e-6, max_iter=100)

Bisection method to find a root of the function 'f(x)' within the interval
`[a, b]`.

# Arguments
- 'f': Function to find the root of.
- 'a', 'b': Initial interval '[a, b]' where the root is expected to be.
```

```

- 'tol': Tolerance for the root (default is '1e-6').
- 'max_iter': Maximum number of iterations allowed (default is '100').

# Returns
- 'root': Approximate root found within the tolerance.
- 'iterations': Number of iterations taken to converge.
"""
function bisection_method(f, a, b; tol=1e-6, max_iter=100)
    fa = f(a)
    fb = f(b)
    if fa * fb > 0
        error("The function values at the endpoints must have opposite
              ↵ signs.")
    end
    iterations = 0
    while (b - a) / 2 > tol && iterations < max_iter
        c = (a + b) / 2
        fc = f(c)
        if fc == 0
            return c, iterations
        end
        if fa * fc < 0
            b = c
            fb = fc
        else
            a = c
            fa = fc
        end
        iterations += 1
    end
    root = (a + b) / 2
    return root, iterations
end

# Define the function we want to find the root of
function f(x)
    return x^3 - 6x^2 + 11x - 6.1
end

# Initial interval [a, b] and tolerance
a = 0.5
b = 10
tolerance = 1e-6
# Apply the bisection method
root, iterations = bisection_method(f, a, b, tol=tolerance)

# Print results
println("Approximate root: ", root)
println("Iterations taken: ", iterations)
println("Function value at root: ", f(root))

```

Approximate root: 3.046680122613907
Iterations taken: 23
Function value at root: -9.356632642010254e-7

A popular practical algorithm is called Brent's Method, which uses additional heuristics to accel-

17. Optimization

erate the optimization routine in most cases.

17.4.1.3. Other Non-Gradient Based Optimization Techniques

17.4.1.3.1. Nelder-Mead simplex method

The Nelder-Mead simplex method is a popular optimization algorithm used for minimizing (or maximizing) nonlinear functions that are not necessarily differentiable. It's particularly useful when gradient-based methods cannot be applied. It is often used in low-dimensional problems due to its simplicity and robustness. We will use the Rosenbrock function, which can be useful in certain portfolio optimization problems, to illustrate the process.

```
using Optim

# Define the Rosenbrock function
rosenbrock(x) = (1 - x[1])^2 + 100 * (x[2] - x[1]^2)^2

# Initial guess for (x, y)
initial_guess = [-1.5, 2.0]

# Perform optimization using the Nelder-Mead method
result = optimize(rosenbrock, initial_guess, NelderMead())

# Extract results
optimal_point = Optim.minimizer(result)
minimum_value = Optim.minimum(result)

println("Optimal Point: ", optimal_point)
println("Minimum Value: ", minimum_value)
```

Optimal Point: [0.9999913430783984, 0.9999847519696222]
Minimum Value: 5.016695917763382e-10

17.4.1.3.2. Simulated annealing

Simulated Annealing (SA) is a probabilistic optimization technique inspired by the annealing process in metallurgy. It is used to find near-optimal solutions to optimization problems, particularly in cases where traditional gradient-based methods may get stuck in local minima/maxima. SA accepts worse solutions with a certain probability, allowing it to explore the search space more broadly initially and then gradually narrow down toward better solutions as it progresses. In this section the Rastrigin function is used to illustrate the process. The function can be useful for asset modeling.

```
using Random

Random.seed!(1234)

# Parameters
max_iterations = 1000 # Number of iterations
T₀ = 100.0 # Starting temperature
cooling_rate = 0.99 # Cooling rate (temperature multiplier)
bounds = (-5.12, 5.12) # Bounds for the search space
dimension = 5 # Number of dimensions in the search space
```

```

# Objective function: Rastrigin function
rastrigin(x) = 10length(x) + sum(xi * xi - 10cos(2π * xi) for xi in x)

# Random initialization within bounds
function initialize_solution()
    bounds[1] .+ (bounds[2] - bounds[1]) .* rand(dimension)
end

# Random perturbation within bounds
function perturb_solution(solution; step=0.1)
    perturbed = solution .+ step .* randn(length(solution))
    return clamp.(perturbed, bounds[1], bounds[2])
end

# Simulated Annealing main function
function simulated_annealing_min()
    cur_sol = initialize_solution()
    cur_f = rastrigin(cur_sol)
    best_sol = copy(cur_sol)
    best_f = cur_f
    cur_t = T₀

    for iteration in 1:max_iterations
        # Generate new candidate solution by perturbation
        cand_sol = perturb_solution(cur_sol)
        cand_f = rastrigin(cand_sol)

        # Acceptance probability (Metropolis criterion)
        ΔE = cand_f - cur_f
        if ΔE < 0 || rand() < exp(-ΔE / max(cur_t, eps()))
            cur_sol = cand_sol
            cur_f = cand_f
        end

        # Update best solution found so far
        if cur_f < best_f
            best_sol = copy(cur_sol)
            best_f = cur_f
        end

        # Decrease temperature
        cur_t *= cooling_rate
        if iteration % 100 == 0
            println("Iter $iteration: Best=$best_f, Temp=$cur_t")
        end
    end

    return best_sol, best_f
end

# Run the simulated annealing algorithm
best_solution, best_value = simulated_annealing_min()
println("Best Solution: ", best_solution)
println("Best Value (Minimum): ", best_value)

```

17. Optimization

```
Iter 100: Best=50.20499770763644, Temp=36.60323412732294
Iter 200: Best=50.20499770763644, Temp=13.397967485796167
Iter 300: Best=35.53682213967214, Temp=4.9040894071285726
Iter 400: Best=35.53682213967214, Temp=1.7950553275045138
Iter 500: Best=34.870094171001725, Temp=0.6570483042414603
Iter 600: Best=34.870094171001725, Temp=0.24050092913110663
Iter 700: Best=34.870094171001725, Temp=0.08803111816824594
Iter 800: Best=34.870094171001725, Temp=0.0322223628802339
Iter 900: Best=34.870094171001725, Temp=0.011794380589564411
Iter 1000: Best=34.870094171001725, Temp=0.004317124741065788
Best Solution: [-3.050932850238516, -0.9876054393249603, -2.006173796601313,
    ↵ 3.9588921514884654, -1.9772040188427922]
Best Value (Minimum): 34.870094171001725
```

Simulated annealing is a stochastic algorithm, so any single run may or may not land near the global minimum at the origin. In practice we experiment with the number of iterations and cooling rate to balance runtime against solution quality; slower cooling schedules explore more broadly at the expense of speed.

17.4.1.3.3. Particle swarm optimization (PSO)

Particle swarm optimization (PSO) is a meta-heuristic optimization algorithm inspired by the social behavior of birds flocking or fish schooling. It is used to solve optimization problems by iteratively improving a candidate solution based on the velocity and position of particles (potential solutions) in the search space. The PSO algorithm differs from other methods in a key way, that instead of updating a single candidate solution at each iteration, we update a population (set) of candidate solutions, called a swarm. Each candidate solution in the swarm is called a particle. We think of a swarm as an apparently disorganized population of moving individuals that tend to cluster together while each individual seems to be moving in a random direction. The PSO algorithm aims to mimic the social behavior of animals and insects.

```
using Random, CairoMakie

# Define the Rosenbrock function
rosenbrock(x) = (1 - x[1])^2 + 100 * (x[2] - x[1]^2)^2

# PSO Implementation
function particle_swarm_optimization(
    objective, n_particles, n_iterations, bounds, dim; vmax=0.5)
    # Initialize particles
    span = bounds[2] - bounds[1]
    positions = [bounds[1] .+ span .* rand(dim) for _ in 1:n_particles]
    # small initial velocities
    velocities = [0.1 .* randn(dim) for _ in 1:n_particles]
    personal_best_positions = deepcopy(positions)
    personal_best_scores = [objective(p) for p in positions]
    best_idx = argmin(personal_best_scores)
    global_best_position = personal_best_positions[best_idx]
    global_best_score = minimum(personal_best_scores)

    # PSO parameters
    w = 0.5          # Inertia weight
    c1, c2 = 2.0, 2.0 # Cognitive and social learning factors
```

```

# Optimization loop
for iter in 1:n_iterations
    for i in 1:n_particles
        # Update velocity
        r1, r2 = rand(), rand()
        velocities[i] .= clamp.(w .* velocities[i] .+
                               c1 .* r1 .* (personal_best_positions[i]
        ↵ .- positions[i]) .+
                               c2 .* r2 .* (global_best_position .-
        ↵ positions[i]),
                               -vmax, vmax)

        # Update position
        positions[i] .+= velocities[i]

        # Clamp positions within bounds
        positions[i] .= clamp.(positions[i], bounds[1], bounds[2])

        # Evaluate fitness
        score = objective(positions[i])
        if score < personal_best_scores[i]
            personal_best_positions[i] = deepcopy(positions[i])
            personal_best_scores[i] = score
        end

        if score < global_best_score
            global_best_position = deepcopy(positions[i])
            global_best_score = score
        end
    end
    if iter % 100 == 0
        println("Iteration $iter: Best Score = $global_best_score")
    end
end
return global_best_position, global_best_score
end

# Parameters
n_particles = 30
n_iterations = 100
bounds = (-2.0, 2.0)
dim = 2

# Run PSO
best_position, best_score = particle_swarm_optimization(
    rosenbrock, n_particles, n_iterations, bounds, dim)

println("Best Position: $best_position")
println("Best Score: $best_score")

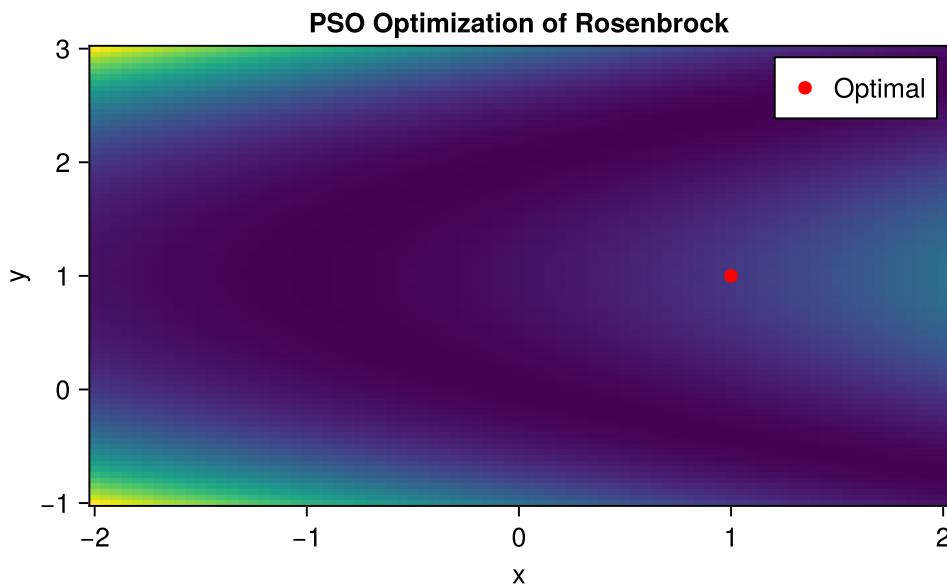
# Visualization
x = -2.0:0.05:2.0
y = -1.0:0.05:3.0
Z = [(1 - x_i)^2 + 100 * (y_i - x_i^2)^2 for y_i in y, x_i in x]

```

17. Optimization

```
# Heatmap and Scatter Plot
fig = Figure()
ax = Axis(fig[1, 1], title="PSO Optimization of Rosenbrock", xlabel="x",
          ylabel="y")
heatmap!(ax, x, y, Z, colormap=:viridis)
scatter!(ax, [best_position[1]], [best_position[2]],
          color=:red, markersize=10, label="Optimal")
axislegend(ax)
fig
```

```
Iteration 100: Best Score = 1.6855704183315996e-20
Best Position: [1.000000000891616, 1.0000000001877603]
Best Score: 1.6855704183315996e-20
```



The swarm converges close to the valley ridge of the Rosenbrock function. Plotting both the surface and the swarm's best point makes it easy to sanity-check whether particles collapsed onto a plausible solution.

17.4.1.3.4. Evolutionary Algorithm

An evolutionary algorithm (EA) is a family of optimization algorithms inspired by the principles of biological evolution. They are particularly useful for solving complex optimization problems where traditional gradient-based methods may struggle due to nonlinearity, multimodality, or high dimensionality of the search space.

The following shows an example to maximize population fitness in terms of an objective function, with common crossover and mutation processes throughout all generations.

```
using Random
```

```

Random.seed!(1234)

# Parameters
population_size = 50 # Number of individuals in the population
chromosome_length = 5 # Number of genes in each individual (dimensionality)
generations = 100 # Number of generations
mutation_rate = 0.1 # Probability of mutation
crossover_rate = 0.7 # Probability of crossover
bounds = (-5.12, 5.12) # Boundaries for each gene

# Target function: Rastrigin function
rastrigin(x) = 10length(x) + sum(xi * xi - 10cos(2π * xi) for xi in x)

# Initialize population randomly within bounds
function initialize_population()
    span = bounds[2] - bounds[1]
    [bounds[1] .+ span .* rand(chromosome_length)
        for _ in 1:population_size]
end

# Fitness function (negative because we are minimizing)
function fitness(individual)
    return -rastrigin(individual)
end

# Selection: Tournament selection
function tournament_selection(population, fitnesses)
    c1, c2 = rand(1:population_size, 2)
    if fitnesses[c1] > fitnesses[c2]
        return population[c1]
    else
        return population[c2]
    end
end

# Crossover: Single-point crossover
function crossover(parent1, parent2)
    if rand() < crossover_rate
        point = rand(1:chromosome_length)
        child1 = vcat(parent1[1:point], parent2[point+1:end])
        child2 = vcat(parent2[1:point], parent1[point+1:end])
        return child1, child2
    else
        return parent1, parent2
    end
end

# Mutation: Randomly change genes with some probability
function mutate(individual)
    for i in eachindex(individual)
        if rand() < mutation_rate
            individual[i] = bounds[1] + (bounds[2] - bounds[1]) * rand()
        end
    end
end

```

17. Optimization

```
    individual
end

# Main Genetic Algorithm loop
function genetic_algorithm()
    population = initialize_population()
    best_individual = nothing
    best_fitness = -Inf

    for gen in 1:generations
        # Evaluate fitness
        fitnesses = [fitness(ind) for ind in population]

        # Find best individual in current population
        current_best = argmax(fitnesses)
        if fitnesses[current_best] > best_fitness
            best_fitness = fitnesses[current_best]
            best_individual = population[current_best]
        end

        # Generate new population
        new_population = Vector{Vector{Float64}}()
        while length(new_population) < population_size
            # Selection
            parent1 = tournament_selection(population, fitnesses)
            parent2 = tournament_selection(population, fitnesses)

            # Crossover
            child1, child2 = crossover(parent1, parent2)

            # Mutation
            child1 = mutate(child1)
            child2 = mutate(child2)

            # Add children to new population
            push!(new_population, child1, child2)
        end
        population = new_population[1:population_size]

        if gen % 100 == 0
            println("Generation $gen: Best Fitness = ", best_fitness)
        end
    end

    return best_individual, -best_fitness
end

# Run the genetic algorithm
best_solution, best_value = genetic_algorithm()
println("Best Solution: ", best_solution)
println("Best Value (Minimum): ", best_value)
```

```
Generation 100: Best Fitness = -2.7903233905305527
Best Solution: [-0.01511189328016016, -0.9890363750908149, -0.041651785419857035,
                ↵ 0.01868237710632581, -1.0362290520785473]
```

```
Best Value (Minimum): 2.7903233905305527
```

Despite their simplicity, evolutionary algorithms can navigate rugged search spaces where gradients are unavailable or uninformative. For financial applications they appear in stress-test design and agent-based simulations where payoff surfaces are discontinuous.

17.4.1.3.5. Bayesian optimization

Bayesian Optimization (BO) is a powerful technique for global optimization of expensive-to-evaluate black-box functions. It leverages probabilistic models to predict the objective function's behavior across the search space and uses these models to make informed decisions about where to evaluate the function next. This approach efficiently balances exploration (searching for promising regions) and exploitation (exploiting regions likely to yield optimal values), making it particularly suitable for optimization problems where function evaluations are costly, such as optimizing parameters of complex simulations.

```
using Random
using Surrogates
using CairoMakie

# reproducible seed
Random.seed!(12345)

# Forrester-like, multimodal objective function
f(x) = (6x - 2)^2 * sin(12x - 4)
lb, ub = 0.0, 1.0

# Sobol initialization
n_init = 6
x_init = sample(n_init, lb, ub, SobolSample())
y_init = f.(x_init)

# Kriging surrogate model
krig = Kriging(x_init, y_init, lb, ub; p=1.9)

# Surrogate optimization loop (Expected Improvement)
maxiters = 20
num_new_samples = 200
best_x, best_f = surrogate_optimize(
    f, EI(), lb, ub, krig, SobolSample();
    maxiters=maxiters, num_new_samples=num_new_samples
)

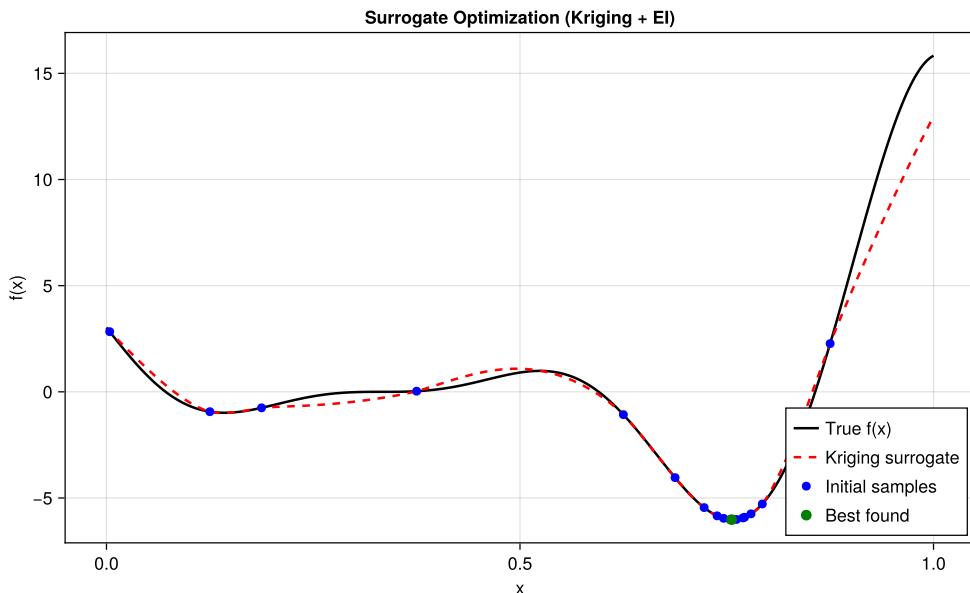
# Plotting
xs = collect(range(lb, ub, length=400))
ys_true = f.(xs)
ys_pred = krig.(xs)

fig = Figure(size=(800, 500))
ax = Axis(
    fig[1, 1],
    title="Surrogate Optimization (Kriging + EI)",
    xlabel="x", ylabel="f(x)"
)
```

17. Optimization

```
# True function
lines!(ax, xs, ys_true, color=:black, linewidth=2, label="True f(x)")
# Surrogate prediction
lines!(ax, xs, ys_pred, color=:red, linestyle=:dash, linewidth=2,
      label="Kriging surrogate")
# Initial samples
scatter!(ax, x_init, y_init, color=:blue, markersize=10, label="Initial
          samples")
# Best found point
scatter!(ax, [best_x], [best_f], color=:green, markersize=12, label="Best
          found")
axislegend(ax, position=:rb)
fig
```

Termination tolerance reached.



While gradient-free methods help optimize complex nonlinear or noisy functions, some financial and operational problems are best modeled with linear relationships. The next section shows how to set up such problems, including constraints, using linear and integer programming.

17.4.2. Linear optimization

Linear optimization, also known as linear programming (LP), is a mathematical method for finding the best outcome in a mathematical model with linear relationships. It involves optimizing a linear objective function subject to a set of linear equality and inequality constraints. Linear programming has a wide range of applications across various fields, including operations research, economics, engineering, and logistics.

We will use linear optimization to solve the following problem, with n the number of elements in b :

$$\begin{aligned} & \max_x \quad c \cdot x \\ \text{subject to} \quad & x \geq 0 \\ & A_i \cdot x \leq b_i \quad \forall i \in \{1, 2, \dots, n\} \end{aligned}$$

```
using JuMP, GLPK, LinearAlgebra

# Define the objective coefficients
c = [1.0, 2.0, 3.0]
# Define the constraint matrix (A) and right-hand side (b)
A = [1.0 1.0 0.0;
      0.0 1.0 1.0]
b = [10.0, 20.0]
# Create a JuMP model
linear_model = Model(GLPK.Optimizer)
# Define decision variables
@variable(linear_model, x[1:3] >= 0)
# Define objective function
@objective(linear_model, Max, dot(c, x))
# Add constraints
@constraint(linear_model, constr[i=1:2], dot(A[i, :], x) <= b[i])
# Solve the optimization problem
optimize!(linear_model)

# Print results
println("Objective value: ", objective_value(linear_model))
println("Optimal solution:")
for i in 1:3
    println("\tx[$i] = ", value(x[i]))
end
```

Objective value: 70.0

Optimal solution:

```
x[1] = 10.0
x[2] = 0.0
x[3] = 20.0
```

17.4.2.1. Integer programming

Integer Programming (IP) is a type of optimization problem where some or all of the variables are restricted to be integers. Although the problem definition seems similar to an LP, the complexity of solving an IP increases hugely as the solution space is not continuous but discrete.

Let us use IP to solve this problem. A factory produces two types of products x_1 and x_2 with the following details:

$$\begin{aligned} & \max_x \quad 40x_1 + 50x_2 \\ \text{subject to} \quad & x_1, x_2 \in \mathbb{Z} \\ & 4x_1 + 3x_2 \leq 200 \quad (\text{labor}) \\ & x_1 + 2x_2 \leq 40 \quad (\text{material}) \end{aligned}$$

17. Optimization

```
# Import necessary packages
using JuMP, GLPK, MathOptInterface

# Create a model with the GLPK solver
model = Model(GLPK.Optimizer)

# Define decision variables (x1 and x2 are integers)
@variable(model, x1 >= 0, Int)
@variable(model, x2 >= 0, Int)

# Define the objective function (maximize profit)
@objective(model, Max, 40 * x1 + 50 * x2)

# Add constraints
@constraint(model, 4x1 + 3x2 <= 200) # Labor constraint
@constraint(model, x1 + 2x2 <= 40) # Material constraint

# Solve the model
optimize!(model)

# Check the solution status
if termination_status(model) == MathOptInterface.OPTIMAL
    println("Optimal solution found!")
    println("x1 (Product x1 units): ", value(x1))
    println("x2 (Product x2 units): ", value(x2))
    println("Maximum Profit: ", objective_value(model))
else
    println("No optimal solution found. Status: ", termination_status(model))
end

Optimal solution found!
x1 (Product x1 units): 40.0
x2 (Product x2 units): 0.0
Maximum Profit: 1600.0
```

17.5. Example: Model Fitting

In model fitting, the “best fitting curve” refers to the curve or function that best describes the relationship between the independent and dependent variables in the data. The goal of model fitting is to find the parameters of the chosen curve or function that minimize the difference between the observed data points and the values predicted by the model.

The process of finding the best fitting curve typically involves:

- Choosing a model: Based on the nature of the data and the underlying relationship between the variables, a suitable model or family of models is selected.
- Estimating parameters: Using the chosen model, one estimates the parameters that best describe the relationship between the variables. This is often done using optimization techniques such as least squares regression, maximum likelihood estimation, or Bayesian inference.
- Evaluating the fit: Once the parameters are estimated, one evaluates the goodness of fit of the model by comparing the predicted values to the observed data. Common metrics

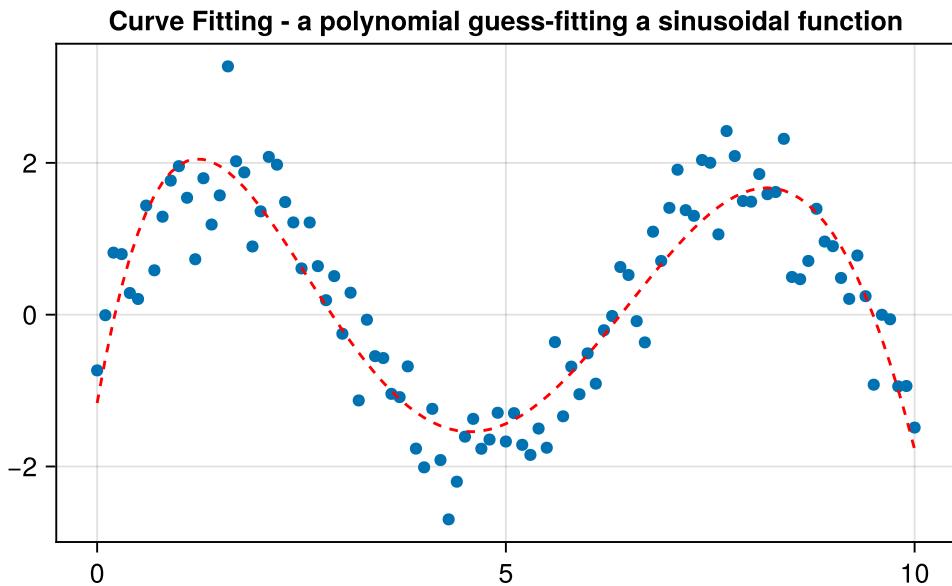
for evaluating fit, or error functions, include the residual sum of squares, the coefficient of determination (R-squared), and visual inspection of the residuals.

- Iterating if necessary: If the fit is not satisfactory, one may need to iterate on the model or consider alternative models until one finds a satisfactory fit to the data.

While we have shown general approaches like BFGS or gradient-free schemes, libraries such as LsqFit wrap these concepts into convenient functions. Under the hood, these packages may employ gradient-based methods (including automatic differentiation) to refine parameters. Below is a demonstration:

```
using Random, LsqFit, CairoMakie

x_data = 0:0.1:10
y_data = 2 .* sin.(x_data) .+ 0.5 .* randn(length(x_data))
# Define the model function, using a polynomial function
# to fit sinusoidal data for illustration purposes
curve_model(x, p) = p[1] * x .^ 5 + p[2] * x .^ 4 .+ p[3] * x .^ 3 .+
    p[4] * x .^ 2 .+ p[5] * x .+ p[6]
# Initial parameter guess
p₀ = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
# Fit the model to the data
fit_result = curve_fit(curve_model, x_data, y_data, p₀)
# Extract the fitted parameters
params = coef(fit_result)
# Evaluate the model with the fitted parameters
y_fit = curve_model(x_data, params)
# Plot the data and the fitted curve
fig = Figure()
Axis(fig[1, 1],
    title="Curve Fitting - a polynomial guess-fitting a sinusoidal function")
scatter!(x_data, y_data, label="Data")
lines!(x_data, y_fit, label="Fitted Curve", linestyle=:dash, color=:red)
fig
```



The recovered parameters are the coefficients of the degree-5 polynomial. In market calibration these parameters would translate to, for example, seasonal components in demand or cyclical risk drivers in factor models.

17.6. More Resources

The textbook “Algorithms for Optimization” (by Kochenderfer and Wheeler) is a comprehensive introduction to optimization and uses Julia for its examples.

In Julia, Optimization.jl provides a unified front-end for all kinds of general optimization problems. JuMP.jl provides a unified front-end in a specialized optimization mini-domain specific language.

18. Visualizations

CHAPTER AUTHORED BY: YUN-TIEN LEE, ALEC LOUDENBACK

Graphical excellence is that which gives to the viewer the greatest number of ideas in the shortest time with the least ink in the smallest space. - Edward Tufte, 2001

18.1. Chapter Overview

The evolved brain and pattern recognition, a general guide for creating and iterating on visualizations, and principles for creating good visualizations while avoiding common mistakes.

18.2. Introduction

The human visual system is astonishingly good at detecting structure. We can spot a trend, an outlier, or a cluster in a scatter plot almost instantly — the kind of pattern recognition that would take minutes of staring at a table of numbers, if we noticed it at all. Good visualization leverages this hardware. It turns data into something your eyes can reason about directly.

Consider the following example of tabular data, with four sets of paired x and y coordinates.

```
using DataFrames

# Define the Anscombe Quartet data
anscombe_data = DataFrame(
    x1=[10.0, 8.0, 13.0, 9.0, 11.0, 14.0, 6.0, 4.0, 12.0, 7.0, 5.0],
    y1=[8.04, 6.95, 7.58, 8.81, 8.33, 9.96, 7.24, 4.26, 10.84, 4.82, 5.68],
    x2=[10.0, 8.0, 13.0, 9.0, 11.0, 14.0, 6.0, 4.0, 12.0, 7.0, 5.0],
    y2=[9.14, 8.14, 8.74, 8.77, 9.26, 8.10, 6.13, 3.10, 9.13, 7.26, 4.74],
    x3=[10.0, 8.0, 13.0, 9.0, 11.0, 14.0, 6.0, 4.0, 12.0, 7.0, 5.0],
    y3=[7.46, 6.77, 12.74, 7.11, 7.81, 8.84, 6.08, 5.39, 8.15, 6.42, 5.73],
    x4=[8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 19.0, 8.0, 8.0, 8.0],
    y4=[6.58, 5.76, 7.71, 8.84, 8.47, 7.04, 5.25, 12.50, 5.56, 7.91, 6.89]
)
```

18. Visualizations

| | x1 | y1 | x2 | y2 | x3 | y3 | x4 | y4 |
|----|---------|---------|---------|---------|---------|---------|---------|---------|
| | Float64 |
| 1 | 10.0 | 8.04 | 10.0 | 9.14 | 10.0 | 7.46 | 8.0 | 6.58 |
| 2 | 8.0 | 6.95 | 8.0 | 8.14 | 8.0 | 6.77 | 8.0 | 5.76 |
| 3 | 13.0 | 7.58 | 13.0 | 8.74 | 13.0 | 12.74 | 8.0 | 7.71 |
| 4 | 9.0 | 8.81 | 9.0 | 8.77 | 9.0 | 7.11 | 8.0 | 8.84 |
| 5 | 11.0 | 8.33 | 11.0 | 9.26 | 11.0 | 7.81 | 8.0 | 8.47 |
| 6 | 14.0 | 9.96 | 14.0 | 8.1 | 14.0 | 8.84 | 8.0 | 7.04 |
| 7 | 6.0 | 7.24 | 6.0 | 6.13 | 6.0 | 6.08 | 8.0 | 5.25 |
| 8 | 4.0 | 4.26 | 4.0 | 3.1 | 4.0 | 5.39 | 19.0 | 12.5 |
| 9 | 12.0 | 10.84 | 12.0 | 9.13 | 12.0 | 8.15 | 8.0 | 5.56 |
| 10 | 7.0 | 4.82 | 7.0 | 7.26 | 7.0 | 6.42 | 8.0 | 7.91 |
| 11 | 5.0 | 5.68 | 5.0 | 4.74 | 5.0 | 5.73 | 8.0 | 6.89 |

Something not obvious by looking at the tabular data above is that each set of data has the same summary statistics. That is, the four sets of data are all described by the same linear features.

```
using Statistics, Printf
let d = anscombe_data
    map([[:x1, :y1], [:x2, :y2], [:x3, :y3], [:x4, :y4]]) do pair
        x, y = eachcol(d[:, pair])

        # calculate summary statistics
        mean_x, mean_y = mean(x), mean(y)
        intercept, slope = ([ones(size(y)) x] \ y) # a one-line OLS
    ↵ regression
        correlation = cor(x, y)

        (; mean_x, mean_y, intercept, slope, correlation)
    end ▷ DataFrame
end
```

| | mean_x | mean_y | intercept | slope | correlation |
|---|---------|---------|-----------|----------|-------------|
| | Float64 | Float64 | Float64 | Float64 | Float64 |
| 1 | 9.0 | 7.50091 | 3.00009 | 0.500091 | 0.816421 |
| 2 | 9.0 | 7.50091 | 3.00091 | 0.5 | 0.816237 |
| 3 | 9.0 | 7.5 | 3.00245 | 0.499727 | 0.816287 |
| 4 | 9.0 | 7.50091 | 3.00173 | 0.499909 | 0.816521 |

Analytical summarization alone is not enough to understand the data. We need to visualize the data to see the patterns emerge, wherein each of the four datasets tells a very different story:

```
using CairoMakie

fig = Figure()

pairs = [(:x1, :y1), (:x2, :y2), (:x3, :y3), (:x4, :y4)]
titles = ["Dataset 1", "Dataset 2", "Dataset 3", "Dataset 4"]

for (i, (xsym, ysym)) in enumerate(pairs)
    row = i ≤ 2 ? 1 : 2
    col = i % 2 == 1 ? 1 : 2
```

```

ax = Axis(fig[row, col], title=titles[i], xlabel="x", ylabel="y")

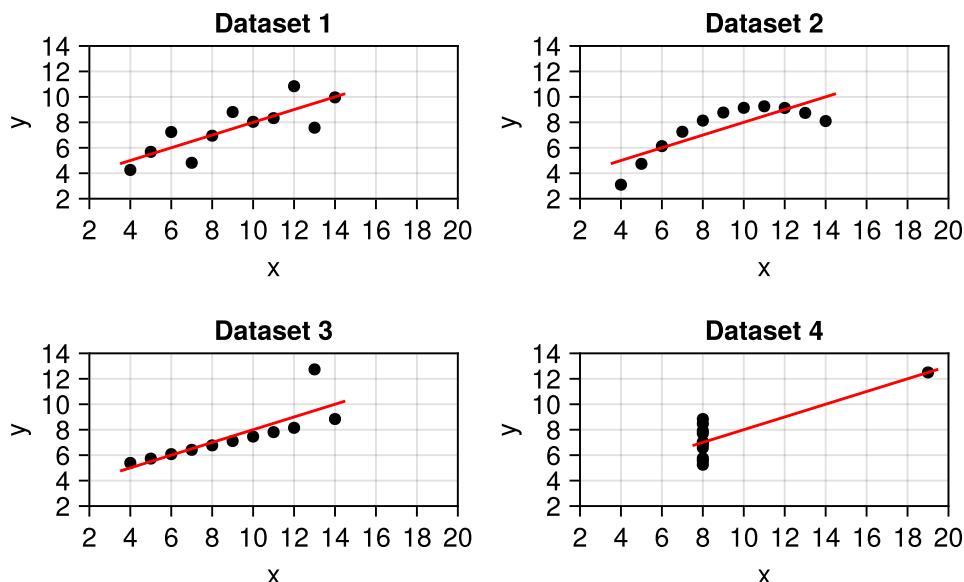
x = anscombe_data[!, xsym]
y = anscombe_data[!, ysym]
scatter!(ax, x, y, color=:black)

intercept, slope = ([ones(size(y)) x] \ y)
xs = range(minimum(x) - 0.5, stop=maximum(x) + 0.5, length=100)
ys = intercept .+ slope .* xs
lines!(ax, xs, ys, color=:red)

# Keep comparable axes to highlight differences
xlims!(ax, 2, 20)
ylims!(ax, 2, 14)
end

fig

```



This is Anscombe's Quartet, a famous demonstration that summary statistics can hide wildly different data-generating processes. Four datasets with identical means, slopes, correlations — and completely different stories once you actually look at them. The lesson generalizes: Table 18.1 summarizes some of the reasons visualization belongs in every modeler's workflow.

Table 18.1.: A list of reasons to practice the art and science of data visualization.

| Purpose | Description |
|---------------------|--|
| Simplify Complexity | Raw data can be overwhelming, especially with large datasets or many variables. A single visualization can condense thousands of data points into a clear picture. |

| Purpose | Description |
|-----------------------------------|--|
| Reveal Patterns and Relationships | Some insights are hidden in plain sight until you visualize them, such as the relationships in Anscombe's Quartet example. |
| Support Better Decisions | Understanding patterns and relationships can then translate into better decision making, such as highlighting trends or risks at a glance. |
| Communicate Effectively | Conveying information to others in a visual manner is one of the most effective ways of aiding understanding. The best visualizations don't just inform — they tell a story that's useful for understanding and decision making. |
| Encourage Exploration | Visual exploration is at the heart of understanding data, uncovering distributions, relationships, or unusual patterns before diving into formal models. |

If you find yourself explaining a table to someone — “see, the third column is growing faster than the second” — that’s a sign a chart would do the job better.

18.3. Developing Visualizations

With the *why* established, let’s turn to the *how*. The principles below aren’t comprehensive — entire careers are built around information design — but they cover the mistakes we see most often in practice.

18.3.1. Define Your Message

Before you write any plotting code, ask: *what am I trying to show?* A visualization that tries to show everything usually shows nothing. If you’re comparing portfolio returns across strategies, that’s a different chart than if you’re illustrating the distribution of a single strategy’s returns. The clearer you are about the one thing the reader should take away, the easier every subsequent design decision becomes.

It also helps to know your audience. A board presentation calls for a different level of detail than an internal model review — not because one audience is smarter, but because their questions are different.

18.3.2. Emphasize Accuracy and Integrity

The most common sin in data visualization is distortion. Truncated axes make small differences look dramatic. Non-uniform scales hide trends or manufacture them. If you’re plotting growth or quantities that span orders of magnitude, a logarithmic axis is often more honest than a linear one.

Human perception introduces its own distortions. We are poor at comparing arc lengths, which is why pie charts are almost always inferior to a simple bar chart. Area is even trickier: it scales as the square of the radius, so a circle that *looks* four times as large may only represent twice the value. These perceptual traps are well-documented, and yet they appear constantly in professional reports. Every visual element should serve the data, not decorate it.

18.3.3. Prioritize Clarity Over Complexity

Tufte coined the term “chartjunk” for decorative elements that add no information — gratuitous gridlines, rainbow color palettes, 3D effects on 2D data. These aren’t just aesthetic complaints; they actively interfere with reading the data. A clean chart with precise labels and a legible font does more work than an elaborate one. When in doubt, remove an element and see if anything is lost. Usually it isn’t.

18.3.4. Organize Data Thoughtfully

When the data has many variables or a long time axis, resist the urge to cram everything into one plot. Small multiples — the same chart repeated across subsets of the data — are one of the most powerful tools in visualization. They let the viewer make comparisons without overloading any single panel. If you do layer multiple datasets onto one plot, make sure each layer is visually distinct. A chart where you can’t tell which line is which has failed at its job.

18.3.5. Enhance Readability

Label your axes. It sounds obvious, but unlabeled or ambiguously labeled axes are among the most common problems in practice. Annotate directly on the plot where possible — the reader shouldn’t have to cross-reference a distant legend to understand what they’re looking at. Keep your color scheme consistent across related charts: if blue means “portfolio A” in one figure, it should mean the same thing in the next.

18.3.6. Validate and Iterate

Show your visualization to someone who wasn’t involved in making it. If they misread it, that’s useful information — the chart is communicating something different from what you intended. Visualization is an iterative process, much like writing: the first draft is rarely the final one.

 Tip

Most financial modelers are familiar with putting together plots in Excel — quick charts, ad-hoc smoothing, and a few formatted tables to paste into presentations. In Julia you get all of that and reproducibility, parameterization, and automation. Instead of manual clicks, a short script can produce consistent charts for every scenario, embed them in reports, save high-resolution files, and be re-run whenever inputs change.

18.3.7. Example: Improving a Disease Funding Visualization

Let’s take a visualization (Figure 18.1) with several issues and apply the principles above to improve it.

This example was found via (Schwarz 2016), which identifies several issues with the graphic. The data is fundamentally two-dimensional (deaths and funding), yet it is presented with four degrees of visual variation: color, vertical ranking, horizontal categorization, and bubble size. That’s twice as many visual channels as data dimensions, which means the chart is working against itself.

The circles compound the problem. People intuit comparisons of area, not diameter — so the breast cancer funding circle *appears* nearly four times as large as the prostate cancer one, even

18. Visualizations

though it represents less than twice the money. Disease names should be placed directly on the circles rather than in a separate legend, both for readability and for accessibility (color-blind readers can't rely on hue alone). The numerical labels carry eight digits of precision for dollar amounts, which is noise, not signal. And one label is missing entirely.

In this revised version, we take the data as accurate and simply recast the visualization. We use a simpler 2D scatterplot mirroring the two-dimensional data, eliminate unnecessary color, and let the labels sit directly within the plot so the eye doesn't need to jump between a legend and the datapoints. We also remove decimal-level precision in the axis ticks (unnecessary to tell the story) and strip out unnecessary plot elements like gridlines and axes without tick labels.

```
using CairoMakie

# Data
diseases = ["Breast Cancer", "Prostate Cancer", "Heart Disease", "Motor
    ↵ Neuron/ALS",
    ↵ "HIV / AIDS", "Chronic Obstructive Pulmonary Disease", "Diabetes",
    ↵ "Suicide"]
money_raised = [257, 147, 54.1, 22.9, 14, 7, 4.2, 3.2]
deaths_us = [41.374, 21.176, 596.577, 6.849, 7.683, 142.942, 73.831, 39.518]

# Create the scatter plot
fig = Figure()
ax = Axis(
    fig[1, 1],
    xlabel="Annual Money Raised (\$millions)",
    ylabel="Annual Deaths in US (thousands)",
    limits=(0, 350, -30, 770),
    xgridvisible=false,
    ygridvisible=false,
)
hidespines!(ax, :t, :r)
scatter!(ax, money_raised, deaths_us)

# Annotate each point with the disease name
for (i, disease) in enumerate(diseases)
    # avoid overlapping labels
    offset = if disease == "Motor Neuron/ALS"
        (0, -15)
    else
        (3, 2)
    end
    text!(ax, money_raised[i], deaths_us[i], text=disease, fontsize=12,
        ↵ offset=offset)
end

# Display the plot
fig
```

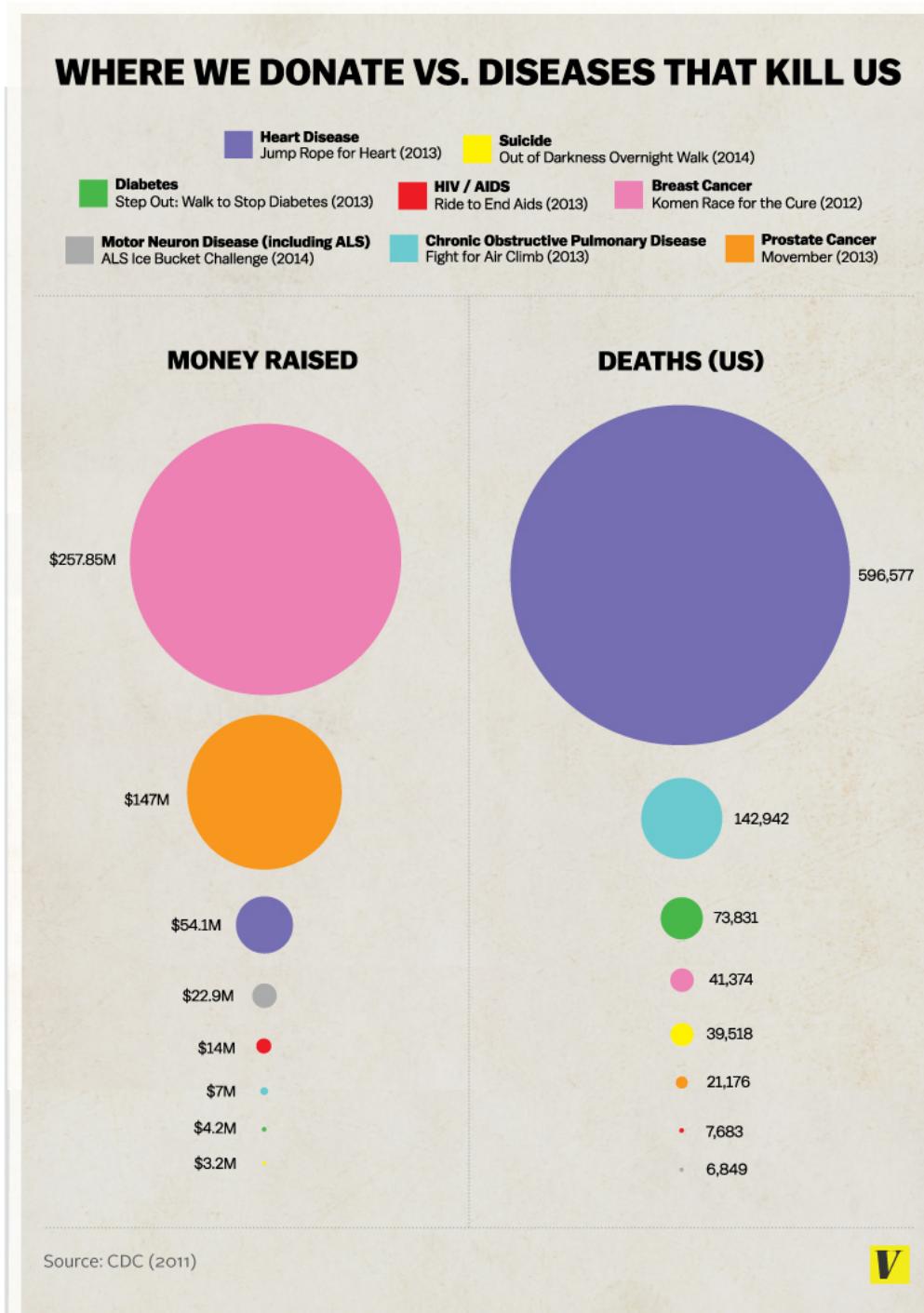
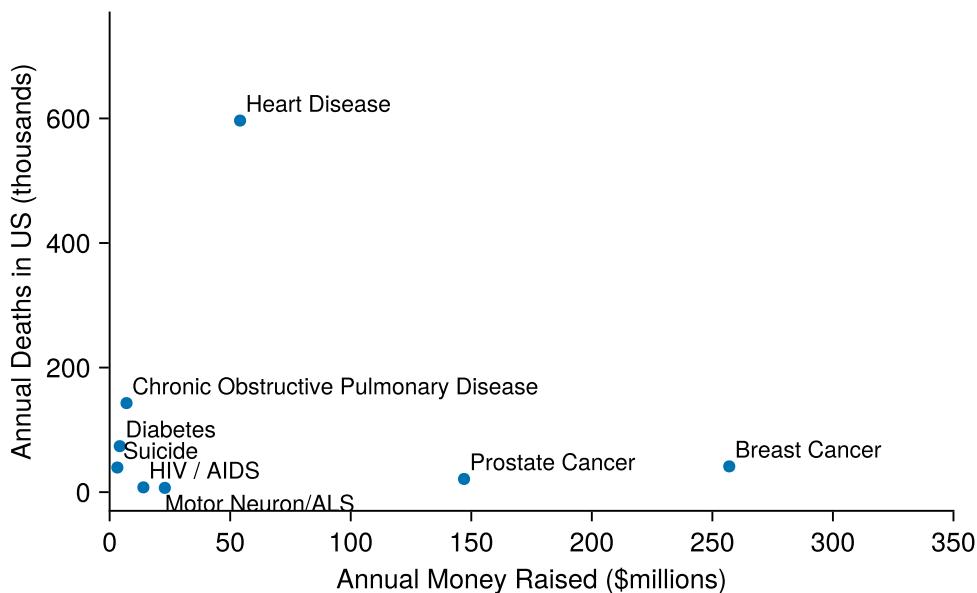


Figure 18.1.: Vox Media Infographic that inappropriately and ineffectively conveys data. From Vox Media (Accessed via Archive.org) (Matthews 2014).



From the revised plot, two things jump out immediately: the cancers receive outsized funding relative to the deaths they cause, and heart disease is an outsized killer relative to everything else on the chart. Neither insight was obvious from the original graphic.

Better still, the clarity of the new chart raises questions the original never could. Is there an inverse relationship between perceived “control” over a disease and how much funding people direct to it? Does funding even correlate with research progress — has money accelerated cancer survival more than it has for heart disease? A good visualization doesn’t just answer the question you started with; it suggests the next question to ask.

18.4. Principles of Good Visualization

The practical advice above is about process. What follows are the underlying principles — most drawn from Tufte’s *The Visual Display of Quantitative Information* (Tufte 2001) — that explain *why* the process works.

First and foremost, represent the data without distortions of size or space. Refrain from clipping axes and do not rely on features such as shape area unless you have fully considered how viewers perceive them.

Use variations of visual features — color, marker style, line weight — to represent data dimensionality with purpose. If colors vary in a plot, they should carry meaning. Rather than jumping to a 3D plot, use variations in marker or line styles, or small multiples, to convey higher dimensions.

Good visualizations encourage the eye to compare different pieces of data and reveal the data at several levels of detail, from a broad overview to the fine structure. Instead of summary statistics alone, try plotting all of the data with reduced transparency and let the viewer draw summary conclusions.

Maintain consistency throughout the exhibit. Any change in font, color, size, or weight can be interpreted as an intentional choice that the viewer will try to interpret — don’t overburden the viewer with unintentional variation. Every visualization should serve a reasonably clear purpose:

description, exploration, tabulation, or decoration. Cut out what's not purposeful and maximize the data-to-ink ratio.

18.5. Types of visualization tools

With principles in hand, let's look at some of the most common plot types and how to build them in Julia. This is far from exhaustive, but it covers the workhorses you'll reach for most often.

 Note

Some of these examples don't fully follow the principles above — we leave in default gridlines, for instance. The priority here is showing how each plot type is constructed in code. In your own work, you'd strip them down further.

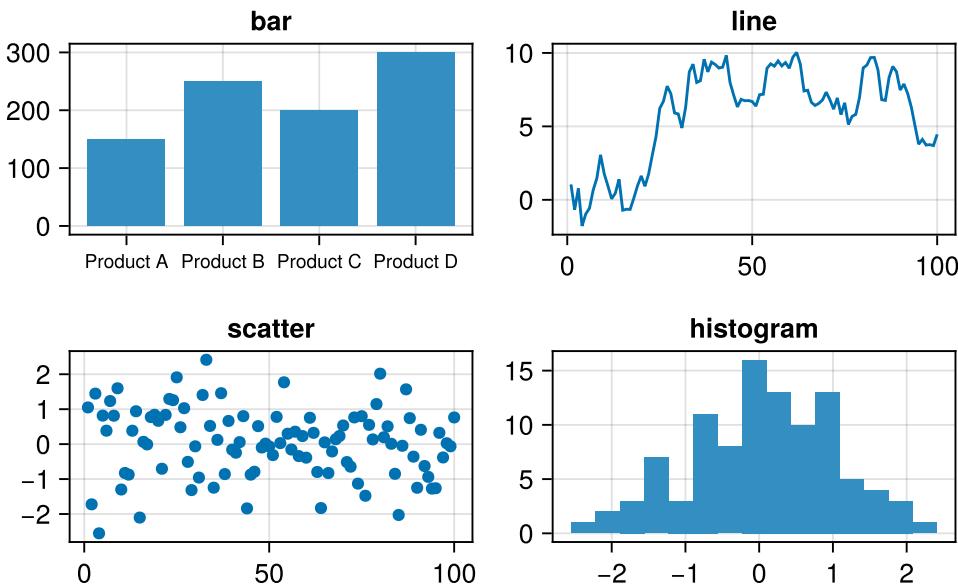
The four workhorses are bar charts, line graphs, scatter plots, and histograms. Bar charts compare categorical or discrete values — product sales, claim counts by region, that sort of thing. Line graphs connect values along a continuous axis (usually time) and make trends and rates of change immediately visible. Scatter plots show the relationship between two continuous variables and are your first stop when looking for correlations, clusters, or outliers. Histograms reveal how a single variable is distributed, which is often more informative than any summary statistic.

```
using Random, CairoMakie

# Data for the plots
categories = ["Product A", "Product B", "Product C", "Product D"]
sales = [150, 250, 200, 300] # For bar chart

x = randn(100) # For scatter plot

# Combine individual plots into a 2x2 layout
f = Figure()
barplot(f[1, 1], 1:4, sales, axis=(xticks=(1:4, categories), title="bar",
    xticklabelsize=10))
axis = Axis(f[1, 2], title="line")
lines!(f[1, 2], cumsum(x))
axis = Axis(f[2, 1], title="scatter")
scatter!(axis, x)
axis = Axis(f[2, 2], title="histogram")
hist!(axis, x)
f
```



When data has more than two dimensions, you need ways to encode the extra information. Heatmaps use color intensity to show the value at each cell of a two-dimensional grid — useful for correlation matrices, transition matrices, or any quantity that varies across two categorical or discretized axes. Bubble charts extend scatter plots by mapping a third variable to the size of each point (though beware the area-perception trap discussed earlier). Parallel coordinates plots draw one vertical axis per variable and connect each observation's values with a line, letting you spot patterns across many dimensions at once. Radar charts do something similar but arrange the axes radially — they work best when comparing a handful of observations, not dozens.

```
using Random, CairoMakie

Random.seed!(1234)

# Data for plots
# For heatmap
xs = range(0, π, length=10)
ys = range(0, π, length=10)
zs = [sin(x * y) for x in xs, y in ys]

bubble_x = rand(10) * 10
bubble_y = rand(10) * 10
bubble_size = rand(10) * 100

# Dummy data for radar chart
radar_data = [0.7, 0.9, 0.4, 0.6, 0.8]

# Dummy data for parallel coordinates plot
parallel_data = rand(10, 5)

f = Figure()
```

```

# Heatmap (1,1)
ax1 = Axis(f[1, 1], title="Heatmap")
heatmap!(ax1, xs, ys, zs)

# Parallel coordinates (1,2)
ax2 = Axis(f[1, 2], title="Parallel Coordinates")
for i in 1:size(parallel_data, 1)
    lines!(ax2, 1:size(parallel_data, 2), parallel_data[i, :])
end

# Radar Chart (2,1)
ax3 = Axis(f[2, 1], title="Radar Chart", aspect=1)
n = length(radar_data)
angles = range(0, 2π, length=n + 1)
r = vcat(radar_data, radar_data[1])

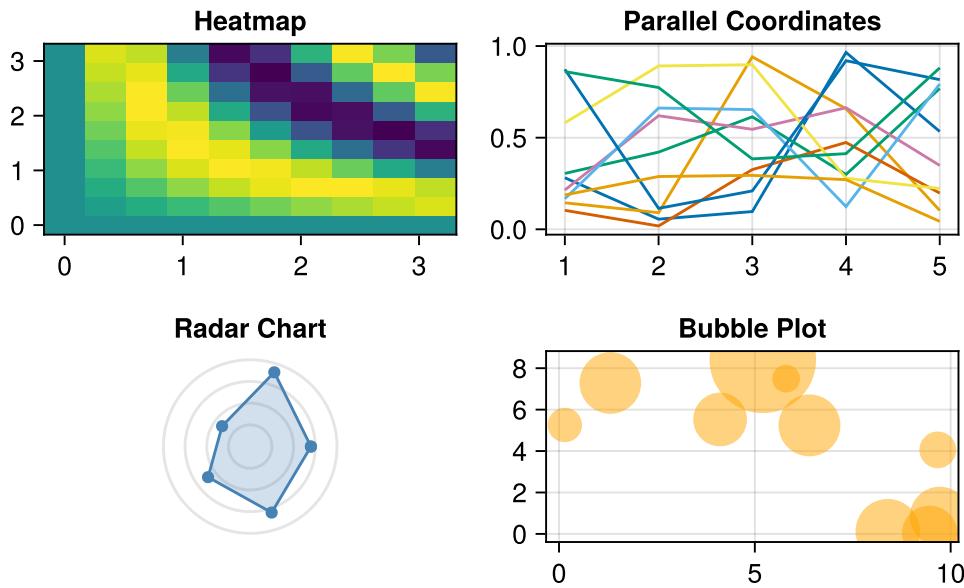
for ρ in (0.25, 0.5, 0.75, 1.0)
    arc!(ax3, Point2f(0), ρ, 0, 2π, color=:grey90) # light radial grid
end

xs = cos.(angles) .* r
ys = sin.(angles) .* r
poly!(ax3, xs, ys, color=(:steelblue, 0.25), strokecolor=:steelblue)
lines!(ax3, xs, ys, color=:steelblue)
scatter!(ax3, xs, ys, color=:steelblue)
hidedecorations!(ax3);
hidespines!(ax3);

# Bubble Plot (2,2)
ax4 = Axis(f[2, 2], title="Bubble Plot")
scatter!(ax4, bubble_x, bubble_y, markersize=bubble_size, color=:orange,
        alpha=0.5)

f

```



Sometimes the data lives in so many dimensions that no single chart type can show it directly. Dimensionality reduction techniques — PCA, t-SNE, UMAP — project high-dimensional data down to two or three dimensions while trying to preserve the structure (clusters, distances, neighborhoods) that matters. The result is a scatter plot you can actually look at. Below, we use t-SNE to project synthetic stock data (five features: volatility, momentum, market cap, P/E ratio, and dividend yield) onto a 2D plane. Stocks with similar financial profiles should land near each other.

```
using TSne, DataFrames, Random, Distributions, CairoMakie, StatsBase

# Generate synthetic financial dataset
Random.seed!(1234)
num_stocks = 100

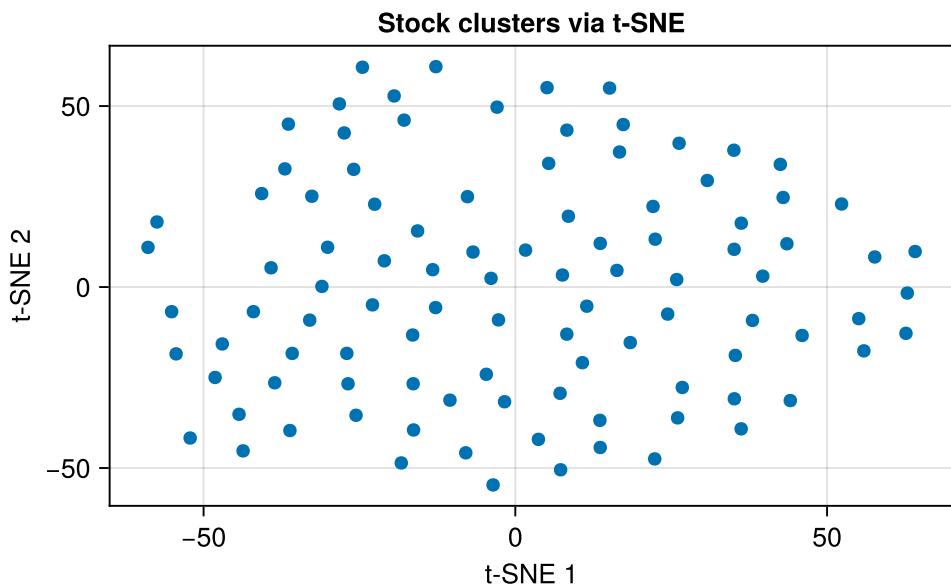
df = DataFrame(
    Stock=["Stock_$(i)" for i in 1:num_stocks],
    Volatility=rand(Uniform(10, 50), num_stocks), # % annualized
    Momentum=rand(Uniform(-10, 30), num_stocks), # 6-month return
    Market_Cap=rand(Uniform(1, 200), num_stocks), # in billion USD
    P_E_Ratio=rand(Uniform(5, 50), num_stocks),
    Dividend_Yield=rand(Uniform(0, 5), num_stocks) # in %
)

# Normalize features
features = [:Volatility, :Momentum, :Market_Cap, :P_E_Ratio, :Dividend_Yield]
X = Matrix(df[:, features])
X = StatsBase.standardize(ZScoreTransform, X, dims=1) # Standardize data

# Apply t-SNE
tsne_result = tsne(X)
```

```
# Add t-SNE components to DataFrame
df.TSNE_1 = tsne_result[:, 1]
df.TSNE_2 = tsne_result[:, 2]

# A graph of somewhat randomly distributed but also small patterns of
# linearity.
fig = Figure()
ax = Axis(fig[1, 1], xlabel="t-SNE 1", ylabel="t-SNE 2", title="Stock
# clusters via t-SNE")
scatter!(ax, df.TSNE_1, df.TSNE_2, strokecolor=:black, markersize=10)
fig
```



Time series have their own visual vocabulary — line charts for trends, area charts for cumulative quantities, decomposition plots for separating signal from seasonality and noise. We cover a worked time series example in Chapter 20.

When the data has a geographic dimension, choropleth maps — maps where regions are shaded by some metric — are a natural choice. Below we build one from a GeoJSON file of US states, coloring each state by a synthetic metric.

```
using GeoJSON, Downloads, DataFrames, CairoMakie, Colors, Random
dl = Downloads.download(
    "https://raw.githubusercontent.com/codeforamerica/" *
    "click_that_hood/master/public/data/united-states.geojson"
)
geo = GeoJSON.read(dl)
state_names = [f.properties[:name] for f in geo.features]
Random.seed!(100)
metric = rand(49)
df_states = DataFrame(name=state_names, value=metric)
```

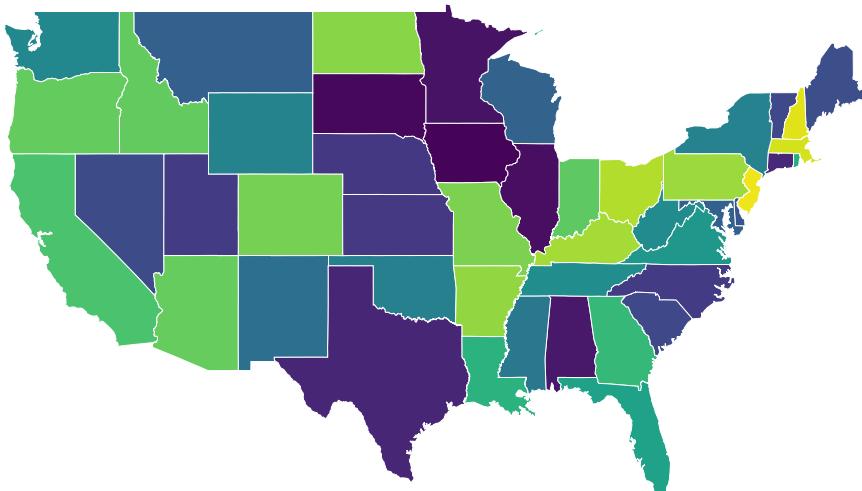
18. Visualizations

```
fig = Figure()
ax = Axis(
    fig[1, 1],
    title="Synthetic state metric",
    xgridvisible=false, ygridvisible=false,
    xlabel="", ylabel="")
)

grad = cgrad(:viridis)
for feature in geo.features
    coords = feature.geometry.coordinates[1][1]
    xs = first.(coords)
    ys = last.(coords)
    name = feature.properties[:name]
    val = df_states[df_states.name==name, :value]
    color = isempty(val) ? :gray80 : get(grad, val[1])
    poly!(ax, xs, ys, color=color, strokecolor=:white, strokewidth=0.5)
end
hidespines!(ax);
hidexdecorations!(ax);
hideydecorations!(ax);

fig
```

Synthetic state metric



Finally, when exploration is the goal rather than presentation, interactive dashboards (Tableau, Power BI, or Julia's own Pluto notebooks) let you filter, zoom, and drill into the data in real time — something static charts can't offer.

18.5.1. Additional Examples

For more kinds of visualizations, see:

- <https://datavizproject.com>
- A Tour Through the Visualization Zoo ((Heer, Bostock, and Ogievetsky 2010))

18.6. Julia Plotting Packages

Julia's plotting ecosystem has matured considerably. The packages below are the ones you're most likely to encounter; which one to reach for depends on whether you need static output, interactivity, or just a quick sanity check in the REPL.

18.6.1. CairoMakie.jl and GLMakie.jl

The Makie family is what we use throughout this book. CairoMakie produces print-quality vector output (PDF, SVG) and is the right choice for publications and reports. GLMakie swaps in a GPU-accelerated backend for interactive or 3D work — same API, different renderer. The defaults are sensible, the customization options are deep, and the code tends to be readable, which is why we chose it here.

18.6.2. Plots.jl

Plots.jl takes a different approach: it provides a single high-level API that can target multiple backends (GR, Plotly, PGFPlotsX, and others). This makes it easy to switch output formats, though the abstraction layer can limit fine-grained control. Its companion package StatsPlots adds statistical chart types — boxplots, violin plots, density plots — and integrates directly with DataFrames.

18.6.3. GraphPlot.jl

For network and graph visualization (social networks, dependency graphs, transition diagrams), GraphPlot.jl works alongside the Graphs.jl ecosystem.

18.6.4. UnicodePlots.jl

Sometimes you just want to see a histogram without leaving the terminal. UnicodePlots renders charts in Unicode characters — no graphics dependencies, no window manager, instant feedback. It's surprisingly useful for quick checks during development.

18.7. References

Much of the material in this chapter is informed by (Tufte 2001).

19. Matrices and Their Uses

CHAPTER AUTHORED BY: YUN-TIEN LEE

"The essence of mathematics is not to make simple things complicated, but to make complicated things simple. — Stan Gudder"

19.1. Chapter Overview

Linear algebra is the workhorse behind portfolio construction, risk decomposition, scenario generation, and many other financial workflows. This chapter revisits essential matrix operations and connects each to a realistic finance task.

19.2. Matrix manipulation

We first review basic matrix manipulation routines before going into more advanced topics.

19.2.1. Addition and subtraction

Think of each matrix as a data grid (like a spreadsheet). Adding or subtracting values element by element is analogous to combining two sets of financial figures—such as merging cash inflows and outflows.

Example: Combining variations in A and B where each element represents a specific scenario's impact when doing cash flow projections. We would like to know the difference in variations between A and B.

```
# Define two matrices with returns across 3 assets over 4 scenarios
A = [0.01 0.02 -0.01;
      0.03 0.01 0.04;
     -0.02 0.01 0.02;
     -0.03 0.03 0.01]

B = [0.02 0.01 -0.01;
      0.02 0.01 -0.01;
     -0.01 0.02 0.01;
      0.01 0.02 -0.02]

# Perform element-wise subtraction
D = A .- B
```

19. Matrices and Their Uses

```
4x3 Matrix{Float64}:
-0.01  0.01  0.0
 0.01  0.0   0.05
-0.01 -0.01  0.01
-0.04  0.01  0.03
```

19.2.2. Transpose

Transposing a matrix is akin to flipping a dataset over its diagonal—turning rows into columns. This operation is useful when aligning data for regression or matching dimensions in financial models.

Example: Converting a time-series (rows as time points) in A into a format suitable for cross-sectional analysis (columns as different variables) in B.

```
# Define a matrix with variable values at all time points
A = [1 2 3;
      4 5 6;
      7 8 9]
# Perform matrix transpose
B = A'
```



```
3x3 adjoint(::Matrix{Int64}) with eltype Int64:
1 4 7
2 5 8
3 6 9
```

Tip

For real-valued data `transpose(A)` and `A'` give the same result. The explicit transpose keeps intent clear and avoids accidentally taking the complex conjugate when your data later includes complex numbers (for example when working with Fourier transforms).

19.2.3. Determinant

The determinant acts as a “volume-scaling” factor. It indicates how much a linear transformation stretches or compresses space. A zero determinant signals that the transformation collapses the space into a lower dimension, implying that the matrix cannot be inverted.

Given a matrix A:

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

the determinant of matrix A can be calculated via Laplace expansion along any row, e.g., row 1 as:

$$\det(\mathbf{A}) = \sum_{j=1}^n (-1)^{1+j} a_{1j} \det(\mathbf{A}_{1j})$$

where \mathbf{A}_{1j} is a sub-matrix obtained by deleting row 1 and column j .

Example: In portfolio theory, a near-zero determinant of a covariance matrix might indicate multicollinearity among assets.

```
using LinearAlgebra

# Define a covariance matrix
A = [2.5e-6 2.5e-6 -1.25e-6;
      2.5e-6 2.5e-6 -1.25e-6;
      -1.25e-6 -1.25e-6 1.25e-6]

# Perform matrix determinant calculation
B = det(A)
```

4.336808689942019e-34

19.2.4. Trace

The trace, being the sum of the diagonal elements, offers a quick summary that can reflect the total variance or influence of a matrix.

Example: In risk analysis, the trace of a covariance matrix may provide insights into the overall market volatility captured by the diagonal elements.

```
using LinearAlgebra

# Define a covariance matrix
A = [2.5e-6 2.5e-6 -1.25e-6;
      2.5e-6 2.5e-6 -1.25e-6;
      -1.25e-6 -1.25e-6 1.25e-6]
# Perform matrix trace calculation
B = tr(A)
```

6.25e-6

19.2.5. Norm

A matrix norm measures the “size” or “energy” of the matrix. It generalizes the concept of vector length to matrices, quantifying the overall magnitude.

The Frobenius norm of a matrix is defined as:

$$\|A\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n |A_{ij}|^2}$$

Example: A common usage of norms includes error analysis, where the norm of the difference between two matrices measures how far an approximation has deviated from the true values. Another common usage in machine learning is during regularization where it allows the training process to know how large an error would be to guide the direction of updating parameters.

```
using LinearAlgebra

# Define an error matrix
A = [-0.01 0.01 0.0;
      0.01 0.0 0.05;
```

19. Matrices and Their Uses

```
-0.01 -0.01 0.01;
-0.04 0.01 0.03]

# Perform matrix norm calculation
B = norm(A)

0.0754983443527075
```

Julia's `norm` defaults to the Frobenius norm for matrices. Use `opnorm(A)` for the operator (spectral) norm if you need the largest singular value instead.

19.2.6. Multiplication

Matrix multiplication (non-element-wise) represents the composition of linear transformations. It's like applying a sequence of financial adjustments — first transforming the data with one factor and then modifying it with another.

Other applications include:

- Transforming asset returns by a matrix representing factor loadings to obtain risk contributions.
- Neural network construction. Matrix multiplication is fundamental for training and using neural networks.
- Systems of linear equations. Many real-world problems reduce to solving systems of linear equations.

```
# Define return and factor matrices
A = [0.01 0.02 -0.01;
      0.03 0.01 0.02;
      -0.02 0.01 0.02;
      -0.03 0.03 0.01]
B = [0.8 0.2;
      0.5 0.5;
      0.3 0.7]

# Perform non-element-wise matrix multiplication
C = A * B

4x2 Matrix{Float64}:
 0.015  0.005
 0.035  0.025
-0.005  0.015
-0.006  0.016
```

On the other hand, element-wise multiplication multiplies corresponding elements directly (like applying a weight matrix).

Example: Adjusting individual cash flow items by their respective risk weights in stress testing.

```
# Define cashflow and weight matrices
A = [100 120 150;
      50 60 70;
      200 180 160;
      80 90 100]
```

```
B = [0.8 0.8 0.7;
     0.5 0.5 0.4;
     0.3 0.2 0.1;
     0.2 0.5 0.8]

# Perform element-wise matrix multiplication
C = A .* B

4x3 Matrix{Float64}:
80.0 96.0 105.0
25.0 30.0 28.0
60.0 36.0 16.0
16.0 45.0 80.0
```

19.2.7. Inversion

Matrix inversion “reverses” a transformation. If a matrix transforms one set of financial assets into another state, its inverse would bring them back.

Example: In solving linear systems for equilibrium pricing, obtaining the inverse of the coefficient matrix allows you to revert to the original asset prices.

That said, production code typically avoids forming the inverse explicitly; instead, we solve systems with the matrix factorization that underlies the inverse.

```
# Define a linear system
A = [1 -0.2 0; -0.1 1 -0.1; 0 -0.3 1]

# Compute the inverse of the matrix
A_inv = inv(A)

3x3 Matrix{Float64}:
1.02105   0.210526  0.0210526
0.105263  1.05263   0.105263
0.0315789 0.315789  1.03158
```

💡 Tip

In numerical work we usually solve systems with $A \setminus b$ instead of computing $\text{inv}(A)$ explicitly. Forming the inverse is slower and amplifies rounding error, whereas the backslash operator reuses a factorization and gives better numerical stability.

ℹ Note

For a matrix to be inverted, it must meet several important criteria:

- **Square Matrix:** The matrix must be square, meaning it has the same number of rows and columns.
- **Non-zero Determinant:** The determinant of the matrix must be non-zero.

19.3. Matrix decomposition

19.3.1. Eigenvalues

Eigenvalue decomposition, also known as eigen decomposition, is a matrix factorization that decomposes a matrix into its eigenvectors and eigenvalues. This technique uncovers the intrinsic “modes” or principal directions in a dataset. The eigenvalues indicate the strength of each mode, while eigenvectors show the direction or pattern associated with that strength. Eigenvalues and eigenvectors are fundamental concepts in linear algebra and play key roles in areas such as:

- Eigenvalues help in analyzing how linear transformations affect vectors in a vector space.
- Eigenvalues facilitate the diagonalization of matrices and simplify the calculations.
- In systems of differential equations, eigenvalues help determine the stability of equilibrium points.
- Identifying the main factors that cause variance in a set of asset returns, which is critical for risk management or stress testing portfolios.
- In graph theory, eigenvalues of the adjacency matrix provide insights into the properties of the graph, such as connectivity, stability, and clustering.
- Many algorithms in data science, like clustering and factorization methods, rely on eigenvalues to identify patterns and reduce dimensionality, which enhances computational efficiency and interpretability.

```
using LinearAlgebra

# Use a symmetric matrix so eigenvalues are real and eigenvectors are
# orthonormal
A = [4.0 1.0 2.0;
      1.0 3.0 0.0;
      2.0 0.0 5.0]

# Perform eigenvalue decomposition
eigen_A = eigen(A)

Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
3-element Vector{Float64}:
1.8548973087995777
3.4760236029181355
6.669079088282288
vectors:
3×3 Matrix{Float64}:
-0.679313 -0.374362 0.631179
 0.593233 -0.786436 0.172027
 0.431981  0.491296 0.75632
```

 Note

For a matrix to have eigenvalues, it must be square, meaning it has the same number of rows and columns.

19.3.2. Singular values

Singular value decomposition (SVD) breaks a matrix into three matrices U , Σ , and V , representing the left singular vectors (analogous to the primary features), the singular values (diagonal matrix capturing the importance), and the right singular vectors (detail on how features interact), respectively. Singular values are key to:

- Matrix factorization, which simplifies many matrix operations, making it easier to analyze and manipulate data.
- Dimensionality reduction. This is particularly useful in high-dimensional data scenarios, where reducing dimensions helps eliminate noise and improve computational efficiency.
- SVD can be used for data compression, particularly in image processing.
- SVD helps filter out noise in data analysis.
- SVD provides a robust method for solving linear equations, particularly when the matrix is ill-conditioned or singular.
- In machine learning, SVD helps extract important features from datasets.
- SVD provides insights into the relationships within data. The singular values indicate the strength of the relationship, while the singular vectors offer a way to visualize and interpret those relationships.

```
using Random, LinearAlgebra

# Create a random matrix
A = rand(4, 3)

4x3 Matrix{Float64}:
0.0571395  0.694474  0.674011
0.488052   0.641082  0.61682
0.506115   0.555514  0.0982501
0.115673   0.98767   0.759523

# Perform Singular Value Decomposition (SVD)
U, Σ, V = svd(A);
```

Left Singular Vectors (U):

```
U

4x3 Matrix{Float64}:
-0.48733   0.398417  0.0725027
-0.507313  -0.3353   0.763219
-0.316449  -0.7789   -0.481863
-0.636399  0.349504  -0.424322
```

Singular Values (Σ):

```
Σ

3-element Vector{Float64}:
1.9365727183978274
0.5551392582030694
0.22617026950523525
```

Right Singular Vectors (V):

19. Matrices and Their Uses

```
v  
3x3 adjoint(::Matrix{Float64}) with eltype Float64:  
-0.262946 -0.891062 0.369955  
-0.758046 -0.0464038 -0.650548  
-0.596847 0.451502 0.663264
```

Reconstructed Matrix:

```
A_reconstructed = U * Diagonal(Σ) * V'  
  
4x3 Matrix{Float64}:  
0.0571395 0.694474 0.674011  
0.488052 0.641082 0.61682  
0.506115 0.555514 0.0982501  
0.115673 0.98767 0.759523
```

Reconstructed Matrix matches Original Matrix:

```
A_reconstructed ≈ A  
  
true
```

19.3.3. Matrix Factorization and Factorization Machines

Matrix factorization is a popular technique in recommendation systems for modeling user-item interactions and making personalized recommendations. The core idea behind matrix factorization is to decompose the user-item interaction matrix into two lower-dimensional matrices, capturing latent factors that represent user preferences and item characteristics. By learning these latent factors, the recommendation system can make predictions for unseen user-item pairs.

Factorization Machines (FM) are a type of supervised machine learning model designed for tasks such as regression and classification, especially in the context of recommendation systems and predictive modeling with sparse data. FM models extend traditional linear models by incorporating interactions between features, allowing them to capture complex relationships within the data.

Example: In credit scoring or recommendation systems for financial products, these techniques reveal latent factors that influence customer behavior.

```
using Recommendation  
using SparseArrays  
using Random  
  
# 1. Synthetic events  
num_users = 100  
num_items = 50  
num_ratings = 500  
user_ids = rand(1:num_users, num_ratings)  
item_ids = rand(1:num_items, num_ratings)  
ratings = rand(1:5, num_ratings)  
  
events = [Event(user_ids[i], item_ids[i], Float64(ratings[i])) for i in  
        1:num_ratings]
```

```

# 2. Train/test split
Random.seed!(1234)
shuffle!(events)
train_size = round(Int, 0.8 * num_ratings)
train_events = events[1:train_size]
test_events = events[train_size+1:end]

# 3. DataAccessor
da_train = DataAccessor(train_events, num_users, num_items)

# 4. Instantiate MatrixFactorization recommender
recommender = MatrixFactorization(da_train)

# 5. Train with hyperparameters
build!(recommender;
      reg=0.05,
      learning_rate=0.007,
      eps=1e-4,
      max_iter=20,
      random_init=true)

# 6. Recommend for a user
user = 1
k = 5
candidates = collect(1:num_items)

# Recommendations for user
rec_list = recommend(recommender, user, k, candidates)

5-element view(::Vector{Pair{Int64, Float64}}, 1:5) with eltype Pair{Int64,
↪ Float64}:
46 => 4.944857708684035
35 => 4.908368961067186
14 => 4.819968020529924
15 => 4.488679841273234
18 => 4.421487980825274

```

Next we evaluate the RMSE:

```

# 7. Evaluate RMSE
predicted = Float64[]
actual = Float64[]
for ev in test_events
    pred_rating = Recommendation.predict(recommender, ev.user, ev.item)
    push!(predicted, pred_rating)
    push!(actual, ev.value)
end

rmse = measure(RMSE(), predicted, actual)

```

1.8805521370275635

Lower RMSE indicates better out-of-sample accuracy. In production you would also monitor business metrics — such as product uptake or credit conversion — but RMSE is a good sanity check that the latent factors are capturing the signal.

19.3.4. Principal component analysis

Principal Component Analysis (PCA) is a widely used technique in various fields for dimensionality reduction, data visualization, feature extraction, and noise reduction. PCA can also be applied to detect anomalies or outliers in the data by identifying data points that deviate significantly from the normal patterns captured by the principal components. Anomalies may appear as data points with large reconstruction errors or as outliers in the low-dimensional space spanned by the principal components.

Example: Compressing various economic indicators into a handful of principal components to illustrate predominant trends in market dynamics or risk factors.

```
using Random, MultivariateStats

# Generate some synthetic data
Random.seed!(1234)
data = randn(5, 100) # 100 samples, 5 features
# Perform PCA
pca_model = fit(PCA, data; maxoutdim=2) # Project to 2 principal components
# Transform the data
transformed_data = transform(pca_model, data)
# Access principal components and explained variance ratio
principal_components = projection(pca_model)
explained_variance_ratio = pca_model.prinvars ./ pca_model.tvar

# Print results
println("Principal Components:")
println(principal_components)
println("Explained Variance Ratio:")
println(explained_variance_ratio)
```

Principal Components:

```
[-0.18634403761212803 -0.1315444664047917; -0.31537354332026074;
 ↵ 0.08600768779892745; ... ; 0.3040245877657475 -0.44791816175497745;
 ↵ 0.6598321794879279 0.6908468907433147]
```

Explained Variance Ratio:

```
[0.29285000654971965, 0.22141889438681142]
```

MultivariateStats expects variables on the rows and observations in the columns. The reported variance ratios guide how many components to retain; in many market datasets the first few components align with broad risk factors such as market, curve, and credit spreads.

20. Learning from Data

CHAPTER AUTHORED BY: YUN-TIEN LEE

"In God we trust; all others bring data." — W. Edwards Deming

"The goal is to turn data into information, and information into insight." — Carly Fiorina

20.1. Chapter Overview

We will touch on how to use data to inform a model: fitting parameters, forecasting, and fundamental limitations on prediction.

20.2. How to learn from data

20.2.1. Understand the problem and define goals

Learning begins with a crisp articulation of the business question.

- Clarify objectives: What do we want to achieve with the data (e.g., prediction, classification, clustering, or insight extraction)?
- Identify key metrics: Determine how success will be measured (accuracy, RMSE, precision, etc.).
- Know the context: Understand the domain and business problem you are addressing to shape the data analysis process.

20.2.2. Collect data

Various data may be available in different formats.

- Ensure data relevance: The data should be relevant to the problem.
- Consider data quality: Collect data with high accuracy, completeness, and consistency.

20.2.3. Explore and preprocess the data

This involves data cleaning and preparation to ensure the dataset is suitable for analysis.

- Handle missing data: You could impute missing values (mean, median, or KNN imputation), or drop rows/columns with excessive missing data.
- Deal with outliers: Use statistical techniques (e.g., z-scores) to detect and remove or cap extreme values.

20. Learning from Data

- Feature scaling: Apply normalization or standardization to ensure features are on comparable scales (important for algorithms like SVM, K-means, etc.).
- Encode categorical data: Use techniques such as one-hot encoding for nominal data, or label encoding or ordinal encoding for ordered categories.
- Data visualization: Use tools like `Makie.jl` to visualize distributions, correlations, and missing values.

20.2.4. Exploratory data analysis (EDA)

EDA helps discover patterns, relationships, and insights within the data. You can perform the following analyses, though this is not an exhaustive list:

- Summary statistics: Check mean, variance, skewness, and correlations between variables.
- Visualize relationships: Use histograms, scatter plots, box plots, and heatmaps to identify trends and correlations.
- Detect multicollinearity: Check correlations between independent variables (e.g., Pearson's correlation matrix).

20.2.5. Select and engineer features

Feature engineering encodes domain knowledge (e.g., yield curve slopes, option moneyness, claim severity ratios) and reduces noise. Techniques range from manual transformations to automated selection (recursive feature elimination, embedded methods, mutual information), keeping parsimony in mind to avoid brittle models.

20.2.6. Choose the right algorithm or model

Depending on the problem type, choose appropriate algorithms for learning from the data:

- Supervised Learning (with labeled data):
 - Classification: Logistic regression, SVM, decision trees, random forests, or neural networks.
 - Regression: Linear regression, ridge regression, or gradient boosting.
- Unsupervised Learning (without labeled data):
 - Clustering: K-means, DBSCAN, hierarchical clustering.
 - Dimensionality Reduction: PCA, t-SNE, or UMAP.
- Reinforcement Learning: Learn from interactions with an environment (e.g., Q-learning, Deep Q-Networks).

20.2.7. Train and evaluate the model

- Split the data: Use either a train-test split (e.g., 80/20 or 70/30 split) or cross-validation (e.g., k-fold cross-validation).
- Fit the model: Train the model on the training set.
- Evaluate the model: Use evaluation metrics appropriate to the task.

20.2.8. Tune hyperparameters

Hyperparameters control how models learn. You can use techniques like the following to tune hyperparameters:

- Grid search: Test a range of hyperparameter values.
- Random search: Randomly explore combinations of hyperparameters.
- Bayesian optimization: Use probabilistic models to guide hyperparameter search.

20.2.9. Deploy and monitor the model

Once the model performs well, deploy it to make predictions on new data.

- Model deployment platforms: Use tools like Flask, FastAPI, or MLOps platforms.
- Monitor performance: Continuously monitor metrics to detect concept drift or performance degradation.

20.2.10. Draw insights and make decisions

Finally, interpret the results and use insights to make decisions or recommendations. Effective communication of findings is essential, especially for stakeholders.

- Visualization: Use dashboards or reports to communicate findings.
- Interpretability: Use explainable AI (e.g., SHAP values) to make model predictions transparent.

20.2.11. Limitations

However, there are certain fundamental limitations:

- There may often be inherent uncertainty and noise in the data itself.
- Every model has its own assumptions and simplifications.
- There may be non-stationarity in the data, especially in financial data. Non-stationary processes change over time, meaning that patterns learned from past data may no longer be valid in the future.
- Models may be overfitting or underfitting. Overfitting occurs when a model is too complex and captures noise instead of the underlying pattern, leading to poor generalization to new data. Underfitting occurs when the model is too simple to capture the relevant structure in the data.
- Sometimes in high-dimensional spaces, data becomes sparse, and meaningful patterns are harder to identify.
- Some predictions may be limited by ethical concerns (e.g., predicting criminal behavior) or legal restrictions (e.g., privacy laws that limit data collection).

20.3. Applications

Learning from data lies at the heart of modern analytics, enabling us to uncover patterns, estimate underlying relationships, and make informed predictions. Two common applications are discussed below.

20.3.1. Parameter fitting

Refer to Chapter 17 on Optimization for more details.

20.3.2. Forecasting

Forecasting is the process of making predictions about future events or outcomes based on historical data, patterns, and trends. It involves the use of statistical methods, machine learning models, or expert judgment to estimate future values in a time series or predict the likelihood of specific events. Forecasting is widely used in fields like economics, finance, meteorology, supply chain management, and business planning.

Here is an example of how to do time series forecasting in Julia, where a shaded band shows the 95% confidence interval derived from the forecast covariance:

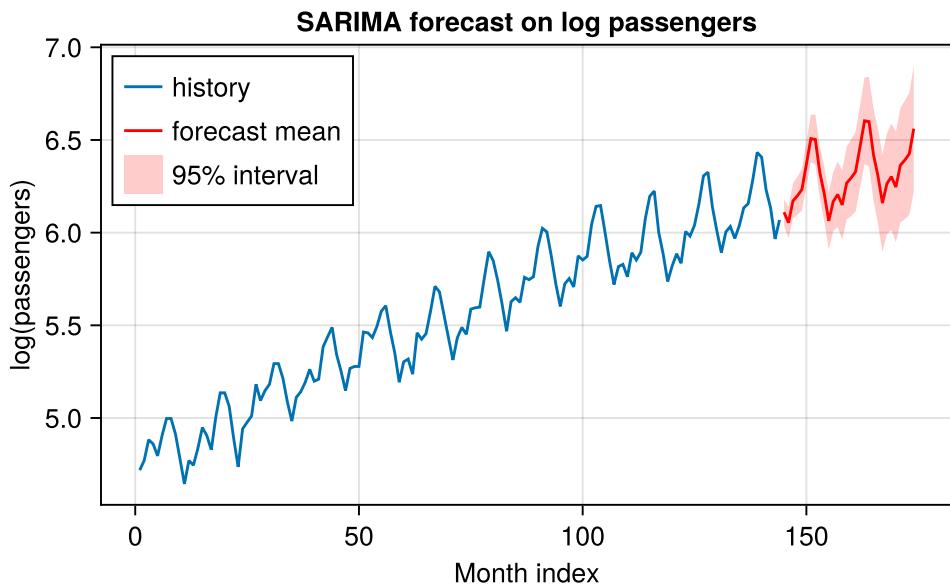
```
using CSV, DataFrames, CairoMakie, StateSpaceModels

airp = CSV.read(StateSpaceModels.AIR_PASSENGERS, DataFrame)
log_air_passengers = log.(airp.passengers)
steps_existing = length(log_air_passengers)
steps_ahead = 30

# SARIMA
model_sarima = SARIMA(log_air_passengers; order=(0, 1, 1),
    seasonal_order=(0, 1, 12))
using Optim: LBFGS, Options
fit!(model_sarima;
    optimizer=StateSpaceModels.Optimizer(LBFGS(), Options(f_reltol=1e-6)))
forec_sarima = forecast(model_sarima, steps_ahead)

forecast_mean = [μ[1] for μ in forec_sarima.expected_value]
forecast_std = [sqrt(Σ[1, 1]) for Σ in forec_sarima.covariance]
future_idx = collect(steps_existing+1:steps_existing+steps_ahead)

f = Figure()
ax = Axis(f[1, 1],
    title="SARIMA forecast on log passengers",
    xlabel="Month index",
    ylabel="log(passengers)")
)
lines!(ax, 1:steps_existing, log_air_passengers, label="history")
lines!(ax, future_idx, forecast_mean, color=:red, label="forecast mean")
band!(ax,
    future_idx,
    forecast_mean .- 1.96 .* forecast_std,
    forecast_mean .+ 1.96 .* forecast_std,
    color=(:red, 0.2),
    label="95% interval")
)
axislegend(ax, position=:lt)
f
```



The model is fit on log-transformed passengers to stabilize variance; exponentiating the forecast mean and interval endpoints returns predictions on the original passenger scale.

Part VI.

Developing in Julia

Aside from the essentials needed to convey the topics, the previous chapters have provided relatively limited insight into the workflow of developing in Julia. Extending our earlier analogy: up to this point we have discussed using hammers and saws to create widgets, and even how to make a widget run faster and more smoothly on different machines. This chapter turns to the process of building and designing larger systems—the practices that make your widget-making factory run smoothly. For a financial modeller this means managing large codebases, shared data feeds, versioned assumptions, continuous testing, and repeatable deployment of valuation and risk engines.

This chapter is heavy with Julia-specific tips and advice. We have deliberately delayed this content until well into the book, focusing on the concepts instead of getting bogged down on language-specific details. In this section, we dive into the messy business of building bigger, integrated things and tools to make that easier—package organisation, environments, Revise, testing, documentation, and distribution. Readers taking the concepts to other languages need not burden themselves with the details of Julia workflows and therefore can jump to the section beginning with Chapter 30.

 Note

The chapters in this section are adapted from *Modern Julia Workflows*, originally written by G. Dalle, J. Smit, and A. Hill. These chapters derive from that work and are also licensed under CC BY-SA 4.0.

The content has been modified to align with the rest of this book (e.g., adding cross-references and removing duplicated material) and to add or remove elements that the authors deemed more appropriate for financial modelers.

21. Writing Julia Code

CHAPTER AUTHORED BY: ALEC LOUDENBACK, MoJuWo CONTRIBUTORS

21.1. Chapter Overview

Installing and setting up your Julia environment. Text editor and REPL editing environments. Setting up your global environment for development. Creating packages. Logging and debugging code. The focus is on building reproducible, auditable modeling workflows that satisfy finance and actuarial governance demands.

21.2. Getting help

Before you write any line of code, it's good to know where to find help. The official help page is a good place to start. In particular, the Julia community is always happy to guide beginners.

As a rule of thumb, the Discourse forum is where you should ask your questions to make the answers discoverable for future users. If you just want to chat with someone, you have a choice between the open source Zulip and the closed source Slack.

For internal work, mirror this habit in your own organization: create an internal forum/wiki where modeling teams can document solutions, common stacktraces, and environment setup steps so regulatory reviews and onboarding go faster.

21.3. Installation

The most natural starting point to install Julia onto your system is the Julia downloads page, which will tell you to use juliaup.

1. Windows users can download Julia and juliaup together from the Windows Store.
2. MacOS or Linux users can execute the following terminal command:

```
curl -fsSL https://install.julialang.org | sh
```

In both cases, this will make the juliaup and julia commands accessible from the terminal (or Windows Powershell). On Windows this will also create an application launcher. All users can start Julia by running

```
julia
```

21. Writing Julia Code

Meanwhile, `juliaup` provides various utilities to download, update, organize and switch between different Julia versions. As a bonus, you no longer have to manually specify the path to your executable. This all works thanks to adaptive shortcuts called “channels”, which allow you to access specific Julia versions without giving their exact number.

For instance, the `release` channel will always point to the current stable version, and the `lts` channel will always point to the long-term support version. Upon installation of `juliaup`, the current stable version of Julia is downloaded and selected as the default.

💡 Tip

To use other channels, add them to `juliaup` and put a `+` in front of the channel name when you start Julia:

```
juliaup add lts  
julia +lts
```

You can get an overview of the channels installed on your computer with

```
juliaup status
```

When new versions are tagged, the version associated with a given channel can change, which means a new executable needs to be downloaded. If you want to catch up with the latest developments, just do

```
juliaup update
```

Keeping `juliaup` channels pinned (e.g., `+lts` for long-term support, or a specific version like `+1.10`) is a good idea for production workflows. With good model governance, you want to control the exact compiler/runtime version rather than silently jumping to the next release mid-quarter.

21.4. REPL

The Read-Eval-Print Loop (or REPL) is the most basic way to interact with Julia, check out its documentation for details. You can start a REPL by typing `julia` into a terminal, or by clicking on the Julia application in your computer. It will allow you to play around with arbitrary Julia code:

```
julia> a, b = 1, 2;  
julia> a + b  
3
```

This is the standard (Julia) mode of the REPL, but there are three other modes you need to know. Each mode is entered by typing a specific character after the `julia>` prompt. Once you’re in a non-Julia mode, you stay there for every command you run. To exit it, hit backspace after the prompt and you’ll get the `julia>` prompt back.

21.4.1. Help mode (?)

By pressing `?` you can obtain information and metadata about Julia objects (functions, types, etc.) or unicode symbols. The query fetches the docstring of the object, which explains how to use it.

```

help?> println
search: println print sprint pointer printstyled

println([io::IO], xs...)

Print (using print) xs to io followed by a newline. If io is not supplied,
↳ prints to the default output stream stdout.

See also printstyled to add colors etc.

Examples
=====

julia> println("Hello, world")
Hello, world

julia> io = IOBuffer();

julia> println(io, "Hello", ',', " world.")
julia> String(take!(io))
"Hello, world.\n"

```

If you don't know the exact name you are looking for, type a word surrounded by quotes to see in which docstrings it pops up.

21.4.2. Package mode ([])

By pressing] you access Pkg.jl, Julia's integrated package manager, whose documentation is an absolute must-read. Pkg.jl allows you to:

-]activate different local, shared or temporary environments;
-]instantiate them by downloading the necessary packages;
-]add,]update (or]up) and]remove (or]rm) packages;
- get the]status (or]st) of your current environment.

As an illustration, we download the package Example.jl inside a new environment we call `demo` (which will create an associated folder if it does not exist):

```

(demo) pkg> activate demo
Activating new project at `~/demo'

(demo) pkg> add Example
Resolving package versions...
Updating `~/demo/Project.toml'
[7876af07] + Example v0.5.5
Updating `~/demo/Manifest.toml'
[7876af07] + Example v0.5.5

(demo) pkg> status
Status `~/demo/Project.toml'
[7876af07] Example v0.5.5

```

Note that the same keywords are also available in Julia mode:

21. Writing Julia Code

```
julia> using Pkg  
  
julia> Pkg.rm("Example")  
Updating `~/demo/Project.toml`  
[7876af07] - Example v0.5.5  
Updating `~/demo/Manifest.toml`  
[7876af07] - Example v0.5.5
```

The package mode itself also has a help mode, accessed with ?, in case you're lost among all these new keywords.

21.4.3. Shell mode (;

By pressing ; you enter a terminal, where you can execute any command you want, such as changing the working directory to the folder we just created:

```
shell> cd demo  
/Users/myself/demo
```

21.5. Editor

In theory, any text editor suffices to write and modify Julia code. In practice, an Integrated Development Environment (or IDE) makes the experience much more pleasant, thanks to code-related utilities and language-specific plugins.

The best IDE for Julia is Visual Studio Code, or VSCode, developed by Microsoft. The Julia VSCode extension is the most feature-rich of all Julia IDE plugins. You can download it from the VSCode Marketplace and read its documentation.

💡 Tip

In what follows, we will sometimes mention commands and keyboard shortcuts provided by this extension. But the only shortcut you need to remember is **Ctrl + Shift + P** (or **Cmd + Shift + P** on Mac): this opens the VSCode command palette, in which you can search for any command. Type **julia** in the command palette to see what you can do.

21.6. Running code

You can execute a Julia script from your terminal, but in most cases that is not what you want to do.

```
julia myfile.jl # avoid this
```

Julia has a rather high startup and compilation latency. If you only use scripts, you will pay this cost every time you run a slightly modified version of your code. That is why many Julia developers fire up a REPL at the beginning of the day and run all of their code there, chunk by chunk, in an interactive way. Full files can be run interactively from the REPL with the `include` function. In production pipelines, wrap your script logic in a module so that `include` loads it once and you can iterate without recompiling on every run.

```
julia> include("myfile.jl")
```

Alternatively, `includet` from the `Revise.jl` package can be used to “include and track” a file. This will automatically update changes to function definitions in the file in the running REPL session.

💡 Tip

Running code is made much easier by the following commands:

- **Julia:** Restart REPL (shortcut Alt + J then Alt + R) - this will open or restart the integrated Julia REPL. It is different from opening a plain VSCode terminal and launching Julia manually from there.
- **Julia:** Execute Code in REPL and Move (shortcut Shift + Enter) - this will execute the selected code in the integrated Julia REPL, like a notebook.

When keeping the same REPL open for a long time, it’s common to end up with a “polluted” workspace where the definitions of certain variables or functions have been overwritten in unexpected ways. This, along with other events like `struct` redefinitions, might force you to restart your REPL now and again, and that’s okay.

21.7. Notebooks

Notebooks are a popular alternative to IDEs when it comes to short and self-contained code, typically in data science. They are also a good fit for literate programming, where lines of code are interspersed with comments and explanations.

The most well-known notebook ecosystem is Jupyter, which supports **Julia**, **Python** and **R** as its three core languages. To use it with Julia, you will need to install the `IJulia.jl` backend. Then, if you have also installed Jupyter with `pip install jupyterlab`, you can run this command to launch the server:

```
jupyter lab
```

If you only have `IJulia.jl` on your system, you can run this snippet instead:

```
julia> using IJulia
julia> IJulia.notebook() # <1>
```

- ① Launches classic Jupyter Notebook; for JupyterLab use `IJulia.jupyterlab()` if JupyterLab is installed

💡 Tip

Jupyter notebooks can be opened, edited, and run in VS Code via the Jupyter and Julia extensions, without installing Python’s Jupyter or `IJulia.jl` system-wide.

`Pluto.jl` is a newer, pure-Julia tool, adding reactivity and interactivity. It is also more amenable to version control than Jupyter notebooks because notebooks are saved as plain Julia scripts. Pluto is unique to Julia because of the language’s ability to introspect and analyze dependencies in its own code. Pluto also has built-in package/environment management, meaning that Pluto notebooks contain all the code needed to reproduce results (as long as Julia and Pluto are installed).

To try out Pluto, install the package and then run

21. Writing Julia Code

```
julia> using Pluto  
julia> Pluto.run()
```

21.8. Markdown

Markdown is a markup language used to add formatting elements to plain text content, such as Julia docstrings. Additionally, other tools such as Quarto (described below) are built using Markdown notation as the basis for their formatting, so it's useful to know about Markdown and the most essential elements.

21.8.1. Plain Text Markdown

Plain text markdown files, which have the `.md` extension, are not used for interactive programming, meaning that code written in the file cannot be run. As a result, plain text markdown files are usually rendered into a final product by other software.

This is an example of a plain text markdown file, including a code example contained within the ````` block:

```
# Title  
  
## Section Header  
  
This is example text.  
  
```julia  
println("hello world")
````
```

21.8.2. Quarto

Quarto “is an open-source scientific and technical publishing system.” Quarto’s primary authoring format is the `.qmd` (Quarto Markdown) file. Quarto can also render plain `.md` files, but `.qmd` enables executable code cells and richer metadata.

Like plain text markdown files, Quarto markdown files also integrate with editors, such as VSCode.



Install the Quarto extension for a streamlined experience.

Unlike plain text markdown files, Quarto markdown files have executable code chunks. These code chunks provide a functionality similar to notebooks, thus Quarto markdown files are an alternative to notebooks. Additionally, Quarto markdown files give users additional control over output and styling via the YAML header at the top of the `.qmd` file.

As of Quarto version 1.5, users can choose from two Julia engines to execute code - a native Julia engine and IJulia.jl. The primary difference between the native Julia engine and IJulia.jl is that the native Julia engine does not depend on Python and can utilize local environments. For this

reason it's recommended to start with the native Julia engine. Learn more about the native Julia engine in Quarto's documentation.

This book is built using Quarto documents to create the associated typeset book and website.

21.9. Environments and Dependencies

Julia comes bundled with Pkg.jl, an environment and package manager. It enables installation of packages from registries, pinning versions for compatibility, and analyzing your dependencies. Environment generally means the computer you use and the software installed on it. When we speak about **environments** in the Julia context, we mean the Julia version and packages available to the current Julia code. For example, is a given package installed and usable from the current code?

If you open a Julia REPL, by default you will be in the *global* environment. If you hit `]` to enter Pkg mode, you should see:

```
(@v1.10) pkg>
```

The `(@v1.10)` indicates that you are using the global environment for the current Julia version (there is no global environment which applies across all Julia versions installed). You can activate a new environment with `activate [environment name]`.

```
(@v1.10) pkg> activate MyNewEnv
Activating new project at `~/MyNewEnv`
```

This will not do anything, yet! When we add a package to this environment, *then* it will create a `Project.toml` and `Manifest.toml` file in that directory. Now that directory is a full fledged Julia project!

💡 Tip

Activate a temporary environment with `activate --temp`. This will give you a temporary environment with a random name, which is very useful for testing out things in a clean, simplified environment. Note that environment stacking still applies, so the global environment, like `@v1.10` will be available inside your temp environment. For auditability, always capture your Project/Manifest files and store them alongside model deliverables so results can be reproduced months later.

21.9.1. Project.toml

A `Project.toml` file defines attributes about the current project and its dependencies. Julia uses this to understand how to reference your current project and what dependencies it should look for from registries when instantiating the project.

ℹ️ Note

TOML (Tom's Obvious Markup Language) is a modern configuration file format used to store settings and data in a human-readable, plaintext format.

This is a bit abstract, so here is a quick, annotated tour of an example `Project.toml` file:

21. Writing Julia Code

```
name = "FinanceCore"          ①
uuid = "b9b1ffdd-6612-4b69-8227-7663be06e089" ②
authors = ["alecloudenback <alecloudenback@users.noreply.github.com> and
           ↵ contributors"]
version = "2.1.0"            ③

[deps]                         ④
Dates = "ade2ca70-3891-5945-98fb-dc099432e06a"
LoopVectorization = "bdcacae8-1622-11e9-2a5c-532679323890"
Roots = "f2b01f46-fcfa-551c-844a-d8ac1e96c665"

[sources]                      ⑤
SPIRVIntrinsics = {url = "https://github.com/JuliaGPU/OpenCL.jl", rev =
                        ↵ "main", subdir = "lib/intrinsics"}

[compat]                        ⑥
Dates = "1"
LoopVectorization = "^0.12"
Roots = "^1.0, 2"
julia = "1.6"
```

- ① The name is the name of your current project which only matters if you turn your project into a package.
- ② A **UUID** is a unique identifier and can be created with Julia's UUIDs standard library.
- ③ The version follows Semantic Versioning ("SemVer") to convey to Pkg (and users!) information that ties a specific version to a specific code commit¹.
- ④ The deps section records the name of direct dependencies and their UUIDs so that Julia can know which packages to grab in order to make your project run.
- ⑤ A sources section can be specified within `Project.toml` to indicate specific dependencies that are not available from a registry. You can specify a location (including locally on your computer), as well as sub-modules or specific branches. This is available with Julia 1.11 or later.
- ⑥ The compat section defines compatibility bounds for packages that can be enforced (via SemVer) to clarify which versions are allowed to be installed in case incompatibilities arise.

When you instantiate a project (see Section 21.9 for more), Julia will essentially add the packages listed under `deps`, and will **resolve** the compatible versions, generally picking the highest version number for the packages so long as the `compat` section rules are not broken.

When adding the dependencies, those packages themselves likely specify their own set of dependencies and Julia must resolve the entire **dependency graph** or **dependency tree** to allow your current project to work.

Semantic Versioning

Semantic Versioning ("SemVer") is a scheme which uses the three-component version code to convey meaning about different versions of a package to both users and computer systems. With the version scheme `vMAJOR.MINOR.PATCH`, the meaning is roughly as follows:

1. MAJOR increments denote changes to the code which make it incompatible with prior versions.
2. MINOR increments denote changes which add features that are compatible with the

¹When registering a package to a repository, the repository will record the version indicated in the `Project.toml` file along with the git commit id of the package when it is registered.

- prior versions.
3. PATCH increments denote changes which fix issues in prior versions and code written against the prior version is still compatible.

As an example, say we are currently using v2.10.4 of a package, and the following theoretical options are available for us to upgrade to:

- v2.10.5 - The 4 to 5 indicates that something may have been broken in the prior release and so we should upgrade without fear that we need to make changes to our code (unless we relied on the previously broken code!).
- v2.11.0 - The 10 to 11 bump suggests that the new release contains some features which should not require us to change any of our previously written code.
- v3.0.0 - The 2 to 3 indicates that we will potentially have to modify code that we have written that interfaces with this dependency.

SemVer cannot distill all possible compatibility and upgrade information about a set of packages (e.g. an author may release an update with a MINOR version which also includes fixes).

21.9.2. Manifest.toml

The `Manifest.toml` file includes a record of all external dependencies used by the project at hand. Unlike `Project.toml`, this file gets machine generated when Julia instantiates or updates the environment. The contents are basically a long list of your direct dependencies and the dependencies of those direct dependencies and looks something like this:

```
julia_version = "1.10.0"
manifest_format = "2.0"
project_hash = "5fea00df4808d89f9c977d15b8ee992bd408081b"

[[deps.AbstractFFTs]]
deps = ["LinearAlgebra"]
git-tree-sha1 = "d92ad398961a3ed262d8bf04a1a2b8340f915fef"
uuid = "621f4979-c628-5d54-868e-fcf4e3e8185c"
version = "1.5.0"
weakdeps = ["ChainRulesCore", "Test"]

[deps.AbstractFFTs.extensions]
AbstractFFTsChainRulesCoreExt = "ChainRulesCore"
AbstractFFTsTestExt = "Test"

... many more lines
```

i Note

Starting with Julia 1.11, Manifest files will include a version indication, making it nicer to work with multiple Julia versions at one time on a single system.

21.9.3. Reproducibility

Reproducibility fulfills both practical and principled goals. *Practical* in that we can record the complex chain of dependencies that is used in modern computing in order to potentially re-create a result or demonstrate an audit trail of the tools used. *Principled* in that there are circumstances (like science research) in which we want to be able to replicate results. The combination of `Project.toml` and `Manifest.toml` goes a long way towards accomplishing this, as you can share both and with the same hardware and Julia version should be able to get the exact same set of dependencies and therefore run the same code. In practice, this level of reproducibility isn't *usually* needed, as most of the time a set of code can be run accurately without requiring the exact same set of dependencies.

Since dependencies can vary between systems (Windows/Mac) and architectures (x86 vs x64), you may not be able to recreate the Manifest exactly. Nevertheless, it's a fairly low bar if you are trying to maintain the utmost level of rigor around the toolchain and Julia is one of the most robust languages regarding tools to support open replication of results.

Artifacts

Julia has a system called **artifacts** which allows specification of a location and hash (a cryptographic key) for data and binaries. The artifact system downloads files and verifies that their contents match the expected hash. This is designed for more permanent data and less end-user workflows, but we call it out here as another example where Julia takes steps to promote consistency and reproducibility.

For more on data workflows for the end-user, see Chapter 12.

Tip

You can configure the environment in which a VSCode Julia REPL opens. Just click the `Julia env: ...` button at the bottom. By default the Julia extension uses `juliaup`'s default channel, but you can override the executable path in VS Code settings.

21.9.4. A Recommended Environment Set-up

With the tools provided for environment management, there are many different patterns for your environment that *can* work. What *should* you do? We recommend the following pattern for your work to minimize conflicts and other issues that may arise from having an overly-wide set of installed packages.

Table 21.1.: A recommended Julia environment setup.

| Environment | What it is | How it is created | What to include here |
|---|---|--|----------------------|
| Global,
e.g. <code>@v1.12</code>
or <code>@v1.11</code> | A global environment for which all other environments will stack. | When each Julia version is installed, it creates an associated global environment. Development-type or convenience tools like <code>Revise.jl</code> or <code>BenchmarkTools.jl</code> . | |

| | | | |
|---|---|--|--|
| Named Project, e.g. @analysis, @notebooks | A named environment, not tied to a specific file location. | <code>activate @name</code> in Pkg mode. | Packages that assist in specific workflows like: |
| Individual Project Environments, e.g. reports/25Q4/Project.toml | An environment that is associated with a project folder. | <code>activate .</code> in Pkg mode when the working directory is the intended folder. | Dependencies supporting a particular analysis intended to share with others or record the dependency tree via a <code>Manifest.toml</code> . |
| Temporary environments | A throw-away temporary environment used for throwing together a quick example, or creating a minimal environment for testing something out. | <code>activate --temp</code> will create a temporary environment that is effectively gone when the Julia session ends. | One-off examples or ad-hoc analysis that you don't want to save to file. |

System Images (sysimages)

An environment containing a set of regularly used tools like @notebooks containing data analysis dependencies or @tooling containing development dependencies is a good candidate for creating `sysimages`.

System images (system images) are precompiled sets of packages, avoiding the precompiling step for loading packages in each session. In effect, this can make user packages more like system packages like Dates or LinearAlgebra which load very quickly since they are precompiled into the shipped Julia binary.

After making a sysimage, you tell Julia to start with that sysimage like this: `julia --project=@notebooks --sysimage=notebooks_sysimage.so`.

For guidance on making a sysimage, see PackageCompiler.jl or the Julia VS Code extension has features to assist in their creation.

21.10. Creating Local packages

Once your code base grows beyond a few scripts, you will want to create a package of your own. The first advantage is that you don't need to specify the path of every file: using `MyPackage` is enough to get access to the names you define and export (or using `MyPackage: myfunc1, myfunc2` to bring non-exported functions into your environment). Furthermore, by structuring

21. Writing Julia Code

your project as a package, you can specify versions for your package and its dependencies, making your code easier and safer to reuse.

To create a new package locally, the easy way is to use `]generate`.

```
using Pkg  
Pkg.generate("MyPackage");
```

This command initializes a simple folder with a `Project.toml` and a `src` subfolder. As we have seen, the `Project.toml` specifies the dependencies. Meanwhile, the `src` subfolder contains a file `MyPackage.jl` where a module called `MyPackage` is defined. It is the heart of your package, and will typically look like this when you're done:

```
module MyPackage  
  
# imported dependencies  
using OtherPackage1  
using OtherPackage2  
  
# files defining functions, types, etc.  
include("file1.jl")  
include("subfolder/file2.jl")  
  
# names you want to make public  
export myfunc # e.g. defined in 'file1.jl'  
export MyType  
  
end
```

21.10.1. PkgTemplates.jl

`PkgTemplates.jl` is like `]generate` from `Pkg.jl` but provides a number of options to pre-configure the repository for things such as continuous integration, testing, and compatibility. If you are not yet making use of that more advanced functionality, the `]generate` method will work just fine for you.

This will walk you through an interactive prompt to create a package in the desired folder. `~/.julia/dev` is a suggested location, but technically any folder will make do:

```
using PkgTemplates  
cd("~/julia/dev")  
Template(interactive=true)("MyPkg")
```

21.11. Development workflow

Once you have created a package, your development routine might look like this:

1. Open a REPL in which you import `MyPackage`
2. Run some functions interactively, either by writing them directly in the REPL or from a Julia file that you use as a notebook
3. Modify some files in `MyPackage`
4. Go back to step 2

For that to work well, you need code modifications to be taken into account automatically. That is why `Revise.jl` exists. If you start every REPL session by explicitly loading `Revise.jl` (using `Revise`), then all the other packages you import after that will have their code tracked. Whenever you edit a source file and hit save, the REPL will update its state accordingly. To automatically do this for every session, see Section 21.12.

Note

The Julia extension imports `Revise.jl` by default when it starts a REPL.

This is how you get started using your own package once it's set up:

```
using Revise, Pkg
Pkg.activate("./MyPackage")
using MyPackage
myfunc() # defined and exported in MyPackage
MyPackage.myfunc2() # defined and *not* exported in MyPackage
```

Note

If you are working on a set of interrelated packages, you may need to tell those packages to use the *development* version of the package you are modifying, instead of using the latest version available from a registry. For example, say you are working on revisions to `PkgA` in the following dependency tree:

PkgB -- depends on -- > PkgA

If you are modifying `PkgA`, then you might need to tell `PkgB` to use the development version. For this, then you would need to:

1. Create an outer environment where you want to run the packages for interactive use while developing (say `activate @mydevenv`).
2. `]dev PkgB` which will download the associated repository into `~/.julia/dev/PkgB`
3. Go into the environment `~/.julia/dev/PkgB` and tell that environment to use the development version of `PkgA` with `]dev PkgA` (assuming you are modifying `PkgA` also in the `~/.julia/dev/` folder)

Now, in the `@mydevenv` environment, when you load `PkgB` it will load the development version of `PkgA`.

21.12. Configuration

Julia accepts startup flags to handle settings such as the number of threads available or the environment in which it launches. In addition, most Julia developers also have a startup file which is run automatically every time the language is started. It is located at `~/.julia/config/startup.jl`.

The basic component that everyone puts in the startup file is `Revise.jl`. Users also commonly import packages that affect the REPL experience, as well as aesthetic, benchmarking, or profiling utilities. A typical example is `OhMyREPL.jl` which is widely used for syntax highlighting in the REPL. While other packages are often used, we suggest the following as a minimum:

21. Writing Julia Code

```
# save as a file in ~/.julia/config/startup.jl
try
    using Revise
    using OhMyREPL
catch e
    @warn "Error with startup packages"
end
```

More generally, the startup file allows you to define your own favorite helper functions and have them immediately available in every Julia session. StartupCustomizer.jl can help you set up your startup file.

Tip

Here are a few more startup packages that can make your life easier once you know the language better:

- AbbreviatedStackTraces.jl allows you to shorten error stacktraces, which can sometimes get pretty long (beware of its interactions with VSCode)
- Term.jl offers a completely new way to display things like types and errors (see the advanced configuration to enable it by default).

21.13. Interactivity

The Julia REPL comes bundled with InteractiveUtils.jl, a bunch of very useful functions for interacting with source code.

Here are a few examples:

```
using InteractiveUtils # not necessary in a REPL session
supertypes(Int64)

(Int64, Signed, Integer, Real, Number, Any)

subtypes(Integer)

3-element Vector{Any}:
Bool
Signed
Unsigned

# first five methods that take an integer argument
methodswith(Integer)[1:5]

[1] Array(s::LinearAlgebra.UniformScaling, m::Integer, n::Integer) @ LinearAlgebra
↳ ~/julia/juliaup/julia-1.12.5+0.aarch64.apple.darwin14/Julia-1.12.app/Content_
↳ s/Resources/julia/share/julia/stdlib/v1.12/LinearAlgebra/src/uniformscaling.j_
↳ l:436
[2] Float16(x::Integer) @ Base float.jl:240
[3] GenericMemoryRef(mem::GenericMemoryRef, i::Integer) @ Core boot.jl:598
[4] GenericMemoryRef(mem::GenericMemory, i::Integer) @ Core boot.jl:597
[5] Integer(x::Integer) @ Core boot.jl:990
```

```

@which exp(1) # where the currently used function is defined

exp(x::Real)
    @ Base.Math math.jl:1543

apropos("matrix exponential") # search docstrings

```

Base.exp
Base.^

When you ask for help on a Julia forum, you might want to include your local Julia information:

```

versioninfo()

Julia Version 1.12.5
Commit 5fe89b8ddc1 (2026-02-09 16:05 UTC)
Build Info:
  Official https://julialang.org release
Platform Info:
  OS: macOS (arm64-apple-darwin24.0.0)
  CPU: 14 x Apple M4 Max
  WORD_SIZE: 64
  LLVM: libLLVM-18.1.7 (ORCJIT, apple-m4)
  GC: Built with stock GC
Threads: 10 default, 1 interactive, 10 GC (on 10 virtual cores)
Environment:
  JULIA_NUM_THREADS = auto
  JULIA_PROJECT = @.
  JULIA_LOAD_PATH = @:@stdlib

```

Tip

The following packages can give you even more interactive power:

- InteractiveCodeSearch.jl to look for a precise implementation of a function.
- InteractiveErrors.jl to navigate through stacktraces.
- CodeTracking.jl to extend InteractiveUtils.jl

21.14. Testing

Testing in Julia primarily revolves around the built-in `Test` package, which provides a straightforward way to write and run tests using the `@test` macro. The basic syntax is simple - you write `@test expression` where the expression should evaluate to true for the test to pass.

To run tests in Julia, navigate to the package directory and run `Pkg.test()`, or use the `] test YourPackageName` command in the Julia REPL. This will run the file in the package directory contained in `test/runtests.jl`. The test infrastructure automatically handles setting up the correct environment and dependencies for testing. Test coverage reports can be generated to see which lines of code are exercised by your tests.

`runtests.jl` is just a normal Julia file and can include other files to help organize your tests. This structure integrates naturally with Julia's package manager and testing tools.

21. Writing Julia Code

Test organization is handled through `@testset` blocks, which group related tests together and provide summary statistics when tests are run. For example, extending the introduction to testing from Section 12.3:

```
using Test
function present_value(discount_rate, cashflows)
    v = 1.0
    pv = 0.0
    for cf in cashflows
        v = v / (1 + discount_rate)
        pv = pv + v * cf
    end
    return pv
end

@testset "Scalar Discount" begin
    @test present_value(0.05, 10) ≈ 10 / 1.05
    @test present_value(0.05, 20) ≈ 20 / 1.05
end
@testset "Vector Discount" begin
    @test present_value(0.05, [10]) ≈ 10 / 1.05
    @test present_value(0.05, [10, 20]) ≈ 10 / 1.05 + 20 / 1.05^2
end;
```

| Test Summary: | Pass | Total | Time |
|-----------------|------|-------|------|
| Scalar Discount | 2 | 2 | 0.0s |
| Test Summary: | Pass | Total | Time |
| Vector Discount | 2 | 2 | 0.0s |

There are many more related testing facilities described in the Julia Docs, such as combining for loops with test sets.

💡 Tip

For floating-point comparisons, you'll often want to use `isapprox` (as a shorter symbol: `≈`, typed as `\approx` then pressing Tab) instead of `==` to handle small numerical differences. Here are some examples:

```
@test 1/3 ≈ 0.3333333333333333  
@test 1/3 ≈ 0.333 atol = 1e-3  
@test 1/3 ≈ 0.333 rtol = 1e-3
```

(1)

(2)

(3)

- (1) Passes because the values are sufficiently close.
- (2) Passes because the absolute difference between the values is less than 1/1000.
- (3) Passes because the difference between values is less than 1/1000 times the larger of the two values.

Here's the default behavior for `isapprox`, excerpted from its docstring:

For real or complex floating-point values, if an `atol > 0` is not specified, `rtol` defaults to the square root of `eps` of the type of `x` or `y`, whichever is bigger (least precise). This corresponds to requiring equality of about half of the significant digits. Otherwise, e.g. for integer arguments or if an `atol > 0` is supplied, `rtol` defaults to zero.

The testing workflow in Julia supports both test-driven development and continuous integration seamlessly. Tests can be run locally during development, and services like GitHub Actions can automatically run your test suite on multiple Julia versions and operating systems when you push changes.

Good testing practices in Julia involve testing edge cases, using appropriate numerical tolerances, organizing tests logically, and ensuring adequate coverage of your code's functionality. It's also important to write tests that are clear and maintainable - each test should have a specific purpose and test one thing well.

21. Writing Julia Code

22. Troubleshooting Julia Code

CHAPTER AUTHORED BY: ALEC LOUDENBACK, MoJuWo CONTRIBUTORS

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." - Brian Kernighan

22.1. Chapter Overview

Debugging in Julia involves a mix of strategies, including using print statements, the Debugger package for step-by-step inspection, logging with the Logging module, and interactive debugging with Infiltrator. These tools and techniques help uncover why a pricing routine blew up at 3 a.m., which scenario caused a risk run to stall, or why yesterday's valuation disagreed with the trading desk. The sections below highlight patterns that show up frequently in financial-model code.

22.2. Error Messages and Stack Traces

Julia's error messages and stack traces can be quite informative. When an error occurs, Julia provides a traceback that shows the function call stack leading to the error, which helps in identifying where things went wrong.

```
function mysqrt(x)
    return sqrt(x)
end

mysqrt(-1) # This will raise a `DomainError`
```

```
DomainError: DomainError(-1.0, "sqrt was called with a negative real argument
    ↵ but will only return a complex result if called with a complex argument.
    ↵ Try sqrt(Complex(x)).")
DomainError with -1.0:
sqrt was called with a negative real argument but will only return a complex
    ↵ result if called with a complex argument. Try sqrt(Complex(x)).
Stacktrace:
[1] throw_complex_domainerror(f::Symbol, x::Float64)
    @ Base.Math ./math.jl:33
[2] sqrt
    @ ./math.jl:627 [inlined]
[3] sqrt
    @ ./math.jl:1546 [inlined]
[4] mysqrt(x::Int64)
    @ Main.Notebook ~/prog/julia-fin-book/julia-debugging.qmd:22
```

22. Troubleshooting Julia Code

```
[5] top-level scope
@ ~/prog/julia-fin-book/julia-debugging.qmd:25
```

The **stacktrace** will show us the sequence of function calls that led to the error. The printout will show the list of functions that were called (the **callstack**) that led to the code that errored. Additionally, help text is often printed, potentially offering some advice for resolving the issue. When you encounter errors in an interactive session, you can click on different parts of the stacktrace and be taken to the associated code in your editor.

22.2.1. Error Types

Notice that errors are given specific types and do not just result in a generic `Error`. This aids user understanding: if a `DomainError` occurs, you know that you passed the right type (e.g., a `Float64` to a function that takes a number), just that the value was not acceptable (as in the example above). Contrast that with a `MethodError` which will tell you that you've passed an invalid kind of thing to the function, not just that its value was off:

```
mysqrt("a string isn't OK")

MethodError: MethodError(sqrt, ("a string isn't OK",), 0x00000000000009745)
MethodError: no method matching sqrt(::String)
The function `sqrt` exists, but no method is defined for this combination of
 ↳ argument types.
Closest candidates are:
  sqrt(::ComplexF16)
    @ Base math.jl:1527
... (11 more lines omitted)
```

Make it a habit to read both the error type and the offending signature—they usually point directly to the place that needs attention.

22.3. Logging

Logging is useful when you encounter a problem in your code or want to track progress. A common reflex is to add `print` statements everywhere.

```
function printing_func(n)
    for i in 1:n
        println(i^2)
    end
end

printing_func (generic function with 1 method)

printing_func(3)

1
4
9
```

A slight improvement is given by the `@show` macro, which displays the variable name:

```

function showing_func(n)
    for i in 1:n
        @show i^2
    end
end

showing_func (generic function with 1 method)

showing_func(3)

```

```

i ^ 2 = 1
i ^ 2 = 4
i ^ 2 = 9

```

But you can go even further with the macros `@debug`, `@info`, `@warn` and `@error`. They have several advantages over printing:

- They display variable names and a custom message
- They show the line number they were called from
- They can be disabled and filtered according to source module and severity level
- They work well in multithreaded code
- They can write their output to a file

```

function warning_func(n)
    for i in 1:n
        @warn "This is bad" i^2
    end
end

warning_func (generic function with 1 method)

warning_func(3)

```

Warning: This is bad
 i ^ 2 = 1
 @ Main.Notebook ~/prog/julia-fin-book/julia-debugging.qmd:83
 Warning: This is bad
 i ^ 2 = 4
 @ Main.Notebook ~/prog/julia-fin-book/julia-debugging.qmd:83
 Warning: This is bad
 i ^ 2 = 9
 @ Main.Notebook ~/prog/julia-fin-book/julia-debugging.qmd:83

Refer to the logging documentation for more information.

Note

In particular, note that `@debug` messages are suppressed by default. You can enable them through the `JULIA_DEBUG` environment variable if you specify the source module name, typically `Main` or your package module.

Beyond the built-in logging utilities, `ProgressLogging.jl` has a macro `@progress`, which interfaces nicely with VSCode and Pluto to display progress bars. And `Suppressor.jl` can sometimes be handy when you need to suppress warnings or other bothersome messages (use at your own risk).

22.4. Commonly Encountered Macros

Aside from those mentioned in the context of Logging, there are a number of different useful macros, many of which are highlighted in the following table:

Table 22.1.: Useful macros for modeling work. There are others related to parallelism which will be covered in Chapter 11.

| Macro | Description |
|---|---|
| <code>BenchmarkTools.@benchmark</code> | Runs the given expression multiple times, collecting timing and memory allocation statistics. Useful for benchmarking and performance analysis. |
| <code>BenchmarkTools.@btime</code> | Similar to <code>@benchmark</code> , but focuses on the minimum execution time and provides a more concise output. |
| <code>@edit</code> | Opens the source code of a function or module in an editor for inspection or modification. |
| <code>@which</code> | Displays the method that would be called for a given function call, helping to understand method dispatch. |
| <code>@code_warntype</code> | Shows the type inference results for a given function call, highlighting any type instabilities or performance issues. |
| <code>@debug, @info, @warn, @error</code> | Used for logging messages at different severity levels (info, warning, error) during program execution. |
| <code>@assert</code> | Asserts that a given condition is true, throwing an error if the condition is false. Useful for runtime checks and debugging. |
| <code>@view, @views</code> | Access a subset of an array without copying the data in that slice. <code>@views</code> applies to all array slicing operations within the expressions that follow it. |
| <code>Test.@test, Test.@testset</code> | Used for defining unit tests. <code>@test</code> checks that a condition is true, while <code>@testset</code> groups related tests together. |
| <code>raw"..."</code> | A string literal prefix that disables string interpolation and escape sequences. Useful for writing raw string data. This is especially helpful when working with filepaths where the \ in Windows paths otherwise needs to be escaped with a leading slash (e.g. \\). |
| <code>@fastmath</code> | Enables aggressive floating-point optimizations within a block, potentially sacrificing strict IEEE compliance for performance. Avoid in numerically sensitive code unless you understand the implications for IEEE semantics. |
| <code>@inbounds</code> | Disables bounds checking for array accesses within a block, improving performance but removing safety checks. |
| <code>@inline</code> | Suggests to the compiler that a function should be inlined at its call sites, potentially improving performance by reducing function call overhead. |
| <code>@profview</code> | Used alongside VS Code to monitor performance and produce a visual indication of code sections that take a lot of time to run. |

Two favorites for performance tuning financial models are `@code_warntype` (to ensure valuation functions stay type-stable) and `BenchmarkTools.@btime` (to confirm a refactor actually sped things up before you check it in).

22.5. Debugging

The limitation of printing or logging is that you cannot interact with local variables or save them for further analysis. The following two packages solve this issue (consider adding to your default environment @v1.X, like Revise.jl).

22.5.1. Setting

Assume you want to debug a function checking whether the n -th Fermat number $F_n = 2^{2^n} + 1$ is prime:

```
function fermat_prime(n)
    k = 2^n
    F = 2^k + 1
    for d in 2:isqrt(F) # integer square root
        if F % d == 0
            return false
        end
    end
    return true
end

fermat_prime (generic function with 1 method)

fermat_prime(4), fermat_prime(6)

(true, true)
```

Unfortunately, $F_4 = 65537$ is the largest known Fermat prime, which means F_6 is incorrectly classified. Let's investigate why this happens!

22.5.2. Infiltrator.jl

Infiltrator.jl is a lightweight inspection package, which will not slow down your code at all. Its `@infiltrate` macro allows you to directly set breakpoints in your code. Calling a function which hits a breakpoint will activate the Infiltrator REPL-mode and change the prompt to `infil>`. Typing `?` in this mode will summarize available commands. For example, typing `@locals` in Infiltrator-mode will print local variables:

```
using Infiltrator

function fermat_prime_infil(n)
    k = 2^n
    F = 2^k + 1
    @infiltrate
    for d in 2:isqrt(F)
        if F % d == 0
            return false
        end
    end
    return true
end
```

22. Troubleshooting Julia Code

What makes Infiltrator.jl even more powerful is the `@exfiltrate` macro, which allows you to move local variables into a global storage called the `safehouse`.

```
julia> fermat_prime_infil(6)
Infiltrating fermat_prime_infil(n::Int64)
  at REPL[2]:4

infil> @exfiltrate k F
Exfiltrating 2 local variables into the safehouse.

infil> @continue

true

julia> safehouse.k
64

julia> safehouse.F
1
```

The diagnosis is a classic one: integer overflow. Indeed, 2^{64} is larger than the maximum integer value in Julia:

```
typemax(Int)

9223372036854775807
```

And the solution is to call our function on “big” integers with an arbitrary number of bits:

```
fermat_prime(big(6))

false
```

Stashing intermediate values in the `safehouse` is particularly handy when a failing scenario only shows up deep in a parallel risk run—you can capture local state from the worker and inspect it later without rerunning the full job.

22.5.3. Debugger.jl

Debugger.jl allows us to interrupt code execution anywhere we want, even in functions we did not write. Using its `@enter` macro, we can enter a function call and walk through the call stack, at the cost of reduced performance.

The REPL prompt changes to `1|debug>`, allowing you to use custom navigation commands to step into and out of function calls, show local variables and set breakpoints. Typing a backtick ` will change the prompt to `1| julia>`, indicating evaluation mode. Any expression typed in this mode will be evaluated in the local context. This is useful to show local variables, as demonstrated in the following example:

```
julia> using Debugger

julia> @enter fermat_prime(6)
In fermat_prime(n) at REPL[7]:1
1  function fermat_prime(n)
>2      k = 2^n
```

```

3      F = 2^k + 1
4      for d in 2:isqrt(F) # integer square root
5          if F % d == 0
6              return false

About to run: (^)(2, 6)
1|debug> n
In fermat_prime(n) at REPL[7]:1
1  function fermat_prime(n)
2      k = 2^n
>3      F = 2^k + 1
4      for d in 2:isqrt(F) # integer square root
5          if F % d == 0
6              return false
7      end

About to run: (^)(2, 64)
1|julia> k
64

```

 Tip

VSCode offers a nice graphical interface for debugging. Click to the left of a line number in an editor pane to add a *breakpoint*, which is represented by a red circle. In the debugging pane of the Julia extension, click Run and Debug to start the debugger. The program will automatically halt when it hits a breakpoint. Using the toolbar at the top of the editor, you can then *continue*, *step over*, *step into* and *step out* of your code. The debugger will open a pane showing information about the code such as local variables inside of the current function, their current values and the full call stack.

The debugger can be sped up by selectively compiling modules that you will not need to step into via the + symbol at the bottom of the debugging pane. It is often easiest to start by adding ALL_MODULES_EXCEPT_MAIN to the compiled list, and then selectively remove the modules you need to have interpreted.

23. Distributing and Sharing Julia Code

CHAPTER AUTHORED BY: ALEC LOUDENBACK, MoJuWo CONTRIBUTORS

23.1. Chapter Overview

Applying software engineering best practices (Chapter 12) in Julia, including testing, documentation, coverage metrics, and release management tailored to financial modeling teams. We focus on collaborating across teams, embedding checks required by model governance, and publishing packages that can power pricing and risk engines across the firm.

23.2. Setup

A vast majority of Julia packages are hosted on GitHub (although less common, other options like GitLab are also possible). GitHub is a platform for collaborative software development based on the version control system Git (see Chapter 12 for an introduction).

The first step is therefore creating an empty repository on GitHub (don't add a README, License, etc. at this step).

Tip

You should try to follow package naming guidelines and add a ".jl" extension at the end, like so: "MyAwesomePackage.jl".

Locally, use PkgTemplates.jl (see Section 21.10.1) to then create the package's folder locally on your computer, which will create a package with several subfolders (these will be described as the chapter progresses).

To sync this up with the newly created GitHub repository, git push this new folder to the remote repository <https://github.com/myuser/MyAwesomePackage.jl>. GitHub should show you how to do this on the associated repository page (something like this):

```
# terminal commands from inside your new package directory:  
git init  
git add .  
git commit -m "Initial commit"  
git branch -M main  
git remote add origin https://github.com/myuser/MyAwesomePackage.jl.git  
git push -u origin main
```

Note

Before publishing anything externally, check with your legal/compliance teams about licensing and data rights—financial code often embeds vendor data contracts or internal IP

that must stay private.

23.3. GitHub Actions

The most useful aspect of PkgTemplates.jl is that it automatically generates workflows for GitHub Actions. These are stored as YAML files in `.github/workflows`, with a slightly convoluted syntax that you don't need to fully understand. For instance, the file `CI.yml` contains instructions that execute the tests of your package (see below) for each pull request, tag or push to the `main` branch. This is done on a GitHub server and should theoretically cost you money, but if your GitHub repository is public, you get an unlimited workflow budget for free.

A variety of workflows and functionalities are available through optional plugins. The interactive setting `Template(..., interactive=true)` allows you to select the ones you want for a given package. Otherwise, you will get the default selection, which you are encouraged to look at. Common add-ons for modeling teams are coverage reports (to satisfy internal control requirements), documentation builds (so users and validators can read how to use the library), and deployment hooks to internal registries.

23.4. Testing

The purpose of the `test` subfolder in your package is unit testing: automatically checking that your code behaves the way you want it to. For instance, if you write your own square root function, you may want to test that it gives the correct results for positive numbers, and errors for negative numbers.

```
using Test

@test sqrt(4) ≈ 2

@testset "Invalid inputs" begin
    @test_throws DomainError sqrt(-1)
    @test_throws MethodError sqrt("abc")
end;
```

Such tests belong in `test/runtests.jl`, and they are executed with the `]test` command (in the REPL's Pkg mode). Unit testing may seem rather naive, or even superfluous, but as your code grows more complex, it becomes easier to break something without noticing. Testing each part separately will increase the reliability of the software you write.

Tip

To test the arguments provided to the functions within your code (for instance their sign or value), avoid `@assert` (which can be deactivated) and use `ArgCheck.jl` instead.

That is, avoid this:

```
function mysqrt(x)
    @assert x >= 0
    ...

```

And do this instead:

```
using ArgCheck
function mysqrt(x)
    @argcheck x >= 0 DomainError
    ...
end
```

At some point, your package may require test-specific dependencies. In essence, you give the test subfolder its own environment and `Project.toml` file. This often happens when you need to test compatibility with another package, on which you do not depend for the source code itself. Or it may simply be due to testing-specific packages like the ones we will encounter below. For interactive testing work, use `TestEnv.jl` to activate the full test environment (faster than running `]test` repeatedly).

💡 Tip

The Julia extension also offers its own more advanced testing framework, which relies on defining “test items” in the code. The benefit of this is that the tests will integrate more directly with the VS Code interface and specific subgroups of tests can be run independently, on-demand. See `TestItemRunner.jl` for more.

💡 Tip

If you want to have more control over your tests, you can try

- `ReferenceTests.jl` to compare function outputs with reference files.
- `ReTest.jl` to define tests next to the source code and control their execution.
- `TestSetExtensions.jl` to make test set outputs more readable.
- `TestReadme.jl` to test whatever code samples are in your `README`.
- `ReTestItems.jl` for an alternative take on VSCode’s test item framework.

23.4.1. Code Coverage

Code coverage refers to the fraction of lines in your source code that are covered by tests (described in more detail in Section 12.3.2). `Codecov` is a website that provides easy visualization of this coverage, and many Julia packages use it. It is available as a `PkgTemplates.jl` plugin. For public GitHub repositories using GitHub Actions, uploads are typically token-less. For private repositories or other CI providers, you’ll need to add the `CODECOV_TOKEN` secret as documented by `Codecov`.

23.5. Code Style

To make your code easy to read, it is recommended to follow a consistent set of guidelines. The official style guide is very short, so most people use third party style guides like `BlueStyle` or `SciMLStyle`.

23. Distributing and Sharing Julia Code

23.5.1. Formatters

23.5.1.1. JuliaFormatter.jl

JuliaFormatter.jl is an automated formatter for Julia files which can help you enforce the style guide of your choice. Just add a file `.JuliaFormatter.toml` at the root of your repository, containing a single line like

```
style = "blue"
```

Then, the package directory will be formatted in the BlueStyle whenever you call

```
import JuliaFormatter
# run from the package root, and
# formats according to .JuliaFormatter.toml
JuliaFormatter.format(".")
```

Note

The default formatter uses JuliaFormatter.jl.

Tip

You can format code automatically in GitHub pull requests with the `julia-format` action, or add the formatting check directly to your test suite.

23.5.1.2. Runic.jl

Runic.jl is a popular choice as well. Like Python's popular Black formatter, there are no configuration options for formatting when using Runic. The benefit is increased consistency and no time wasted debating formatting decisions.

Runic requires you to set it up using Pkg. The instructions are straightforward and available on the Runic.jl repository.

23.6. Code quality

Of course, there is more to code quality than just formatting. Aqua.jl (Auto QUality Assurance) provides a set of routines that examine other aspects of your package, from ensuring that there are no unused dependencies to catching ambiguous methods statically.

Include the following in your tests to have Aqua.jl run various checks each time your tests run:

```
using Aqua, MyAwesomePackage
Aqua.test_all(MyAwesomePackage)
```

JET.jl is a tool that is similar to a static linter in other languages. This means that it can inspect your code and 'understand' it well enough to catch many types of errors before runtime. It does this by running type inference and figuring out how a given type will flow through the call stack of methods.

You can either use it in report mode (with a nice VSCode display) or in test mode as follows:

```
using JET, MyAwesomePackage
JET.report_package(MyAwesomePackage)
JET.test_package(MyAwesomePackage)
```

Note that both Aqua.jl and JET.jl might pick up false positives: refer to their respective documentations for ways to make them less sensitive.

Finally, ExplicitImports.jl can help you get rid of generic imports to specify where each of the variables in your package comes from. As a project gets more complex, using SomePackage can bring many, sometimes conflicting symbols into your current namespace. ExplicitImports forces you to either qualify the usage (e.g. SomePackage.somefunction(...)) or explicitly opt into importing certain variables.

23.7. Documentation

Refer to Section 12.4.2 for more detail on documentation and its importance. Here are some additional workflow tips for setting up documentation for your package.

DocStringExtensions.jl provides a few shortcuts that can speed up docstring creation by taking care of the obvious parts.

In addition to docstrings, Documenter.jl allows you to design a website based on Markdown files contained in the docs subfolder of your package. Unsurprisingly, its own documentation is excellent and will teach you a lot. To build the documentation locally, just run

```
julia> using Pkg
julia> Pkg.activate("docs")
julia> include("docs/make.jl")
```

Then, use LiveServer.jl from your package folder to visualize and automatically update the website as the code changes (similar to Revise.jl, but for your docpages instead of your code):

```
julia> using LiveServer
julia> servedocs()
```

To host the documentation online easily, just select the Documenter plugin from PkgTemplates.jl during creation. Not only will this fill the docs subfolder with the appropriate starting files: it will also initialize a GitHub Actions workflow to build and deploy your website on GitHub pages. Lastly, in your repository's Pages settings, configure deployment from the gh-pages branch (root).

23.8. Literate programming

Literate programming is so-called for combining written documents with the output of programs (literature + code = literate programming). These tools allow you to interleave code with texts, formulas, images and so on.

In addition to Pluto.jl and Jupyter notebooks, take a look at Literate.jl to enrich your code with comments and translate it to various formats. Books.jl is relevant to draft long documents in a pure Julia way.

23. Distributing and Sharing Julia Code

Quarto is an open-source scientific and technical publishing system that supports Python, R and Julia. Quarto can render markdown files (.md), Quarto markdown files (.qmd), and Jupyter Notebooks (.ipynb) into documents (Word, PDF, presentations), web pages, blog posts, books, and more. Additionally, Quarto makes it easy to share or publish rendered content to various online hosts.

PPTX.jl will create Microsoft PowerPoint files.

23.9. Versions and registration

23.9.1. Versions and Compatibility

The Julia community has adopted semantic versioning, which means every package must have a version, and the version numbering follows strict rules (the concept of versioning was covered in Section 12.6.2).

To comply with the versioning requirements in Pkg's resolver, you need to specify compatibility bounds for your dependencies: this happens in the [compat] section of your `Project.toml`. To initialize these bounds with current dependency versions, use the `]compat` command in the Pkg mode of the REPL, or the package `PackageCompatUI.jl`.

Over time, new versions of your dependencies will be released. The `CompatHelper.jl` GitHub Action will help you monitor upstream Julia dependencies and suggest changes to your `Project.toml`'s [compat] section accordingly. In addition, Dependabot can monitor the dependencies... of your GitHub actions themselves. Both of these are included in the default `PkgTemplates` setup.

Tip

It may also happen that you incorrectly promise compatibility with an old version of a package and not realize it (since Pkg prefers newer versions within the compatibility bounds, not all combinations get tested). To prevent that, the `julia-downgrade-compat` GitHub action tests your package with the oldest possible version of every dependency, and verifies that everything still works.

23.9.2. Registration

If your package is useful to others in the community, it may be a good idea to register it, that is, make it part of the pool of packages that can be installed with

```
pkg> add MyAwesomePackage # made possible by registration
```

Note that unregistered packages can also be installed by anyone from the GitHub URL, but this is a less reproducible solution:

```
pkg> add https://github.com/myuser/MyAwesomePackage # not ideal
```

To register your package, check out the general registry guidelines. The `Registrator.jl` bot can help you automate the process. Another handy bot, provided by default with `PkgTemplates.jl`, is TagBot: it automatically tags new versions of your package following each registry release. If you have performed the necessary SSH configuration, TagBot will also trigger documentation website builds following each release.

23.9.2.1. Local Registry

For distributing privately (or publicly if you make the repository public), LocalRegistry.jl provides convenience functions for creating a new registry, adding new packages, and updating package versions. If you want to share packages internally, create and register packages in a repository that's hosted somewhere you and your team can access. If you wanted to make the repository public, you can publish the registry repository somewhere publicly accessible (such as a public GitHub repository).

Once established, other users can add a repository as easily as entering package mode and running `registry add`. Say that we have already put a registry we called `FinancePackages` in a repository on the company intranet:

```
| pkg> registry add http://company-intranet.com/git/FinancePackages.git
```

23.9.2.2. Hosted Registries

Alternatively to a self-hosted local registry, third party services such as JuliaHub provide managed registries well suited for corporate environments.

23.10. Reproducibility

Obtaining consistent and reproducible results is an essential part of model auditing and compliance. One tool to consider is DrWatson.jl. It is a general toolbox for running and re-running models in an orderly fashion.

Some specific issues come up in attempting to ensure reproducibility:

A first hurdle is random number generation, which is not guaranteed to remain stable across Julia versions. To ensure that the random streams remain exactly the same, you need to use StableRNGs.jl. The downside to this is that the random number generation will be considerably slower than the usual generator.

Another aspect is dataset download and management. The packages DataDeps.jl, DataToolkit.jl and ArtifactUtils.jl can help you bundle non-code elements with your package (some of these rely on artifacts - discussed in Section 12.6.3).

💡 Financial Modeling Pro Tip

Always version-control both Project.toml and Manifest.toml for regulated (auditable) workflows. Instantiating the environment on another machine ensures identical dependency versions, which is crucial for reproducible risk and valuation reports.

```
| julia> using Pkg
| julia> Pkg.activate(".")
| julia> Pkg.instantiate()    # resolves to exact versions recorded in
|   ↳ Manifest.toml
```

Note that for this to fully work, the replicating machine needs to have the same architecture (e.g., x64), OS (e.g., Windows), and Julia version (e.g., v1.10). If the versions differ, Julia may need to use a different set of dependencies for compatibility reasons. However, it's still a good practice to store the Manifest.toml for important workflows.

And remember that with *package* repositories, you generally *do not* want to check in the

23. Distributing and Sharing Julia Code

Manifest.toml. Instead, create scripts for the production workflows that *do* check in the Manifest.toml.

23.11. Interoperability

To ensure compatibility with earlier Julia versions, Compat.jl is your best ally.

Making packages play nice with one another is a key goal of the Julia ecosystem. Since Julia 1.9, this can be done with package extensions, which override specific behaviors based on the presence of a given package in the environment. For example, if you want to provide pre-configured plotting, but don't in general need to include a plotting library as part of your package for all users and use cases. PackageExtensionTools.jl eases setting up extensions for your package.

Furthermore, the Julia ecosystem plays nicely with other programming languages too. C and Fortran are natively supported. Python can be easily interfaced with the combination of CondaPkg.jl and PythonCall.jl. Other language compatibility packages can be found in the JuliaInterop organization, like RCall.jl.

23.12. Customization

Part of interoperability is also flexibility and customization: the Preferences.jl package provides a convenient way to specify various options in TOML files. These customizable preferences persist across sessions and provide the preferences at both compile and runtime. For example, say different parts of a company have different preferred data sources but otherwise use the same code. This could be configured via Preferences.jl so that each team can share the logic while seamlessly defaulting to different data sources.

23.13. Collaboration

Once your package grows big enough, you might need to bring in some help. Working together on a software project has its own set of challenges, which are partially addressed by a good set of ground rules like SciML ColPrac. Of course, collaboration goes both ways: if you find a Julia package you really like, you are more than welcome to contribute as well, for example by opening issues or submitting pull requests.

24. Optimizing Julia Code

CHAPTER AUTHORED BY: ALEC LOUDENBACK, MoJuWo CONTRIBUTORS

24.1. Chapter Overview

The two fundamental principles for writing fast Julia code:

1. Ensure that **the compiler can infer the type** of every variable.
2. Avoid **unnecessary (heap) allocations**.

24.2. Type Inference

The compiler's job is to optimize and translate Julia code into runnable machine code. If a variable's type cannot be deduced before the code runs, then the compiler won't generate efficient code to handle that variable. This phenomenon is called "type instability". Enabling type inference means making sure that every variable's type in every function can be deduced from the types of the function inputs alone. Type inference in Julia operates at the function level — outside of functions (e.g., at the REPL or in global scope), Julia does not attempt to infer types in the same way.

In other words, the compiler will be able to create more optimized code if it can analyze the function you've written and determine what type will be returned. In the following function, the compiler can analyze the expressions and determine with certainty that if `mysum` is given two `Ints` as arguments, the return value will also be an `Int`.

```
function mysum(a,b)
    a + b
end
```

24.3. Avoiding Heap Allocations

A "heap allocation" (or simply "allocation") occurs when we create a new variable without knowing how much space it will require (such as a `Vector` with flexible length). This has two implications:

1. Allocating memory on the heap takes substantially more time than creating stack-allocated memory.
2. Periodically, a **garbage collector** (GC) needs to run to de-allocate (free up) memory on the heap that is no longer used by the program.

24. Optimizing Julia Code

Execution of code is stopped while the garbage collector runs, so minimizing its usage is important.

The vast majority of performance tips come down to these two fundamental ideas.

Typically, the most common beginner pitfall is the use of non-constant global variables without passing them as arguments. Why is it bad? Because the type of a global variable can change outside of the body of a function, so it causes type instabilities wherever it is used. Even a type-annotated global variable does not enable the same optimizations as a local variable or a `const`. Those type instabilities in turn lead to more heap allocations.



Tip

If you must use globals (e.g., model parameters loaded once), declare them `const` so their type can't change and the compiler can specialize code that uses them.

Much more detail on performance considerations is covered in Chapter 10.

24.4. Measuring performance

The simplest way to measure how fast a piece of code runs is to use the `@time` macro, which returns the result of the code and prints the measured runtime and allocations. Because code needs to be compiled before it can run, you should first run a function without timing it so it can be compiled, and then time it:

```
sum_abs(vec) = sum(abs(x) for x in vec);
v = rand(100);

using BenchmarkTools
@time sum_abs(v); # Inaccurate, note the >99% compilation time
@time sum_abs(v); # Accurate

0.010093 seconds (92.12 k allocations: 4.466 MiB, 99.90% compilation time)
0.000002 seconds (1 allocation: 16 bytes)
```

Using `@time` is quick but has flaws because your function is only measured once. That measurement might have been influenced by other things going on in your computer at the same time. In general, running the same block of code multiple times is a safer measurement method, because it diminishes the probability of only observing an outlier.

24.4.1. BenchmarkTools

`BenchmarkTools.jl` is the most popular package for repeated measurements on function executions. Similar to `@time`, `BenchmarkTools` offers `@btime`, which can be used in exactly the same way but will run the code multiple times and provide an average. Additionally, by using `$` to interpolate external values, you remove the overhead caused by global variables.

```
using BenchmarkTools
@btime sum_abs(v); # includes global access overhead
@btime sum_abs($v); # interpolates v, avoiding global overhead

29.774 ns (1 allocation: 16 bytes)
17.285 ns (0 allocations: 0 bytes)
```

In more complex settings, you might need to construct variables in a setup phase that is run before each sample. This can be useful to generate a new random input every time, instead of always using the same input.

```
my_matmul(A, b) = A * b;
@btime my_matmul(A, b) setup =
    A = rand(1000, 1000); # use semi-colons between setup lines
    b = rand(1000)
);
```

147.958 μs (3 allocations: 8.06 KiB)

For better visualization, the `@benchmark` macro shows performance histograms:

i Note

Certain computations may be optimized away by the compiler before the benchmark takes place. If you observe suspiciously fast performance, especially below the nanosecond scale, this is very likely to have happened.

24.4.2. Other tools

`BenchmarkTools.jl` works fine for relatively short and simple blocks of code (microbenchmarking). To find bottlenecks in a larger program, you should use a profiler (see next section) or the package `TimerOutputs.jl`. It allows you to label different sections of your code, then time them and display a table grouped by label.

Finally, if you know a loop is slow and you'll need to wait for it to be done, you can use `ProgressMeter.jl` or `ProgressLogging.jl` to track its progress. This will display a progress bar in VS Code or in a notebook, indicating how far along a loop has progressed.

24.5. Profiling

Profiling can identify performance bottlenecks at the function level, and graphical tools such as `ProfileView.jl` are the best way to use it.

24.5.1. Sampling

Whereas a benchmark measures the overall performance of some code, a profiler breaks it down function by function to identify bottlenecks. Sampling-based profilers periodically ask the program which line it is currently executing and aggregate results by line or by function. Julia offers two kinds: one for runtime (in the module `Profile`) and one for memory (in the submodule `Profile.Allocs`).

These built-in profilers print textual outputs, but the result of profiling is best visualized as a flame graph. In a flame graph, each horizontal layer corresponds to a specific level in the call stack and the width of a tile shows how much time was spent in the corresponding function. Here's an example:

24. Optimizing Julia Code

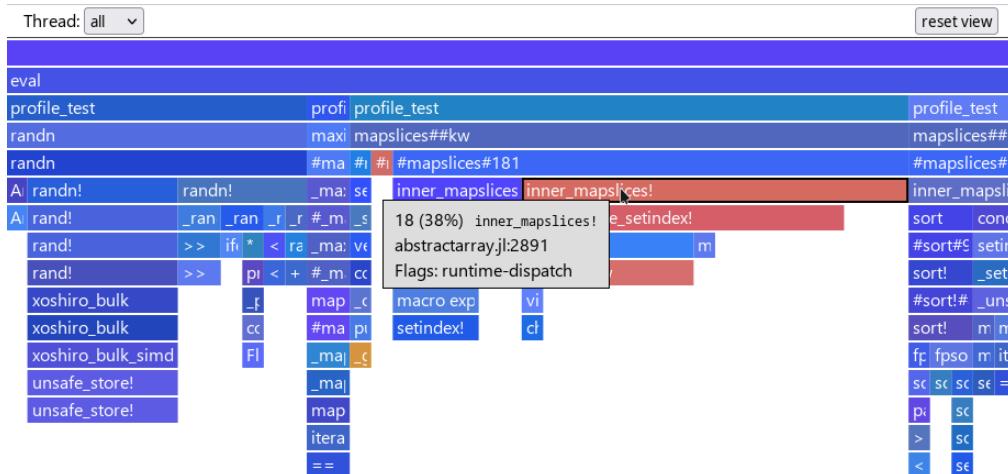


Figure 24.1.: flamegraph

24.5.2. Visualizing Profile Results

The packages `ProfileView.jl` and `PProf.jl` both allow users to record and interact with flame graphs. `ProfileView.jl` is simpler to use, but `PProf` is more featureful and is based on `pprof`, an external tool maintained by Google which applies to more than just Julia code. Here we only demonstrate the former:

```
using ProfileView  
@profview do_work(some_input)
```

💡 Tip

Calling `@profview do_work(some_input)` in the integrated Julia REPL will open an interactive flame graph, similar to `ProfileView.jl` but without requiring a separate package.

In VS Code, you can also call `@profview do_work(some_input)` without adding `ProfileView.jl`; the Julia extension provides this macro.

To integrate profile visualizations into environments like Jupyter and Pluto, use `ProfileSVG.jl` or `ProfileCanvas.jl`, whose outputs can be embedded into a notebook.

No matter which tool you use, if your code is too fast to collect samples, you may need to run it multiple times in a loop.

💡 Tip

To visualize memory allocation profiles, use `PProf.jl` or VSCode's `@profview_allocs`. A known issue with the allocation profiler is that it is not able to determine the type of every object allocated, instead `Profile.Allocs.UnknownType` is shown instead. Inspecting the call graph can help identify which types are responsible for the allocations.

24.5.3. External profilers

Apart from the built-in `Profile` standard library, there are a few external profilers that you can use including Intel VTune (in combination with `IntelITT.jl`), NVIDIA Nsight Systems (in combination with `NVTX.jl`), and Tracy.

24.6. Type stability

A function is **type stable** when the compiler can infer a concrete return type from the types of the inputs alone. “Concrete” means that the size of memory that needs to be allocated to store its value is known at compile time. Types declared abstract with `abstract` type are not concrete and neither are parametric types whose parameters are not specified:

```
@show isconcretetype(Any)
@show isconcretetype(AbstractVector)
@show isconcretetype(Vector) # Shorthand for `Vector{T}` where T
@show isconcretetype(Vector{Real})
@show isconcretetype(eltype(Vector{Real}))
@show isconcretetype(Vector{Int64})

isconcretetype(Any) = false
isconcretetype(AbstractVector) = false
isconcretetype(Vector) = false
isconcretetype(Vector{Real}) = true
isconcretetype(eltype(Vector{Real})) = false
isconcretetype(Vector{Int64}) = true

true
```

 Note

`Vector{Real}` is concrete despite `Real` being abstract for subtle typing reasons but it will still be slow in practice because the type of its elements is abstract.

Type-stable code enables static dispatch and aggressive optimizations (e.g., inlining and specialization). Type-unstable code triggers dynamic dispatch at runtime which prevents many optimizations and often increases allocations.

Type stability is contagious: if a variable’s type cannot be inferred, then the types of variables that depend on it may not be inferrable either. (In advanced cases, “function barriers” can be used to contain instability; see Julia’s performance tips documentation.) Most code should be type-stable unless it has a good reason not to be.

24.6.1. Detecting Instabilities

The simplest way to detect an instability is with the built-in macro `@code_warntype`. The output of `@code_warntype` is difficult to parse, but the key takeaway is the return type of the function’s Body: if it is an abstract type like `Any`, something is wrong. In a normal Julia REPL, such cases would show up colored in red as a warning.

```
using InteractiveUtils # loaded automatically if using a notebook or REPL
function put_in_vec_and_sum(x)
```

24. Optimizing Julia Code

```
v = []           # Vector{Any} → type-unstable
push!(v, x)
return sum(v)
end;

@code_warntype put_in_vec_and_sum(1)

MethodInstance for put_in_vec_and_sum(::Int64)
  from put_in_vec_and_sum(x) @ Main In[7]:2
Arguments
  #self#::Core.Const(Main.put_in_vec_and_sum)
  x::Int64
Locals
  v::Vector{Any}
Body :: Any
1 -      (v = Base.vect())
  %2 = Main.push! ::Core.Const(push!)
  %3 = v::Vector{Any}
  (%2)(%3, x)
  %5 = Main.sum ::Core.Const(sum)
  %6 = v::Vector{Any}
  %7 = (%5)(%6)::Any
  return %7
```

Unfortunately, `@code_warntype` is limited to one function body: calls to other functions are not expanded, which makes deeper type instabilities easy to miss. That is where JET.jl can help: it provides optimization analysis aimed primarily at finding type instabilities. While test integrations are also provided, the interactive entry point of JET is the `@report_opt` macro.

```
using JET
@report_opt put_in_vec_and_sum(1)

[ Info: tracking Base

===== 6 possible errors found =====
put_in_vec_and_sum(x::Int64) @ Main ./In[7]:5
  sum(a::Vector{Any}) @ Base ./reducedim.jl:979
    sum(a::Vector{Any}; dims::Colon, kw::Kwargs{}) @ Base ./reducedim.jl:979
      _sum(a::Vector{Any}, ::Colon) @ Base ./reducedim.jl:983
      _sum(a::Vector{Any}, ::Colon; kw::Kwargs{}) @ Base ./reducedim.jl:983
      _sum(f::typeof(identity), a::Vector{Any}, ::Colon) @ Base
        ./reducedim.jl:984
        _sum(f::typeof(identity), a::Vector{Any}, ::Colon; kw::Kwargs{}) @ Base
          ./reducedim.jl:984
          mapreduce(f::typeof(identity), op::typeof(Base.add_sum), A::Vector{Any})
            @ Base ./reducedim.jl:326
            mapreduce(f::typeof(identity), op::typeof(Base.add_sum), A::Vector{Any});
              dims::Colon, init::Base._InitialValue) @ Base ./reducedim.jl:326
              _mapreduce_dim(f::typeof(identity), op::typeof(Base.add_sum),
                ::Base._InitialValue, A::Vector{Any}, ::Colon) @ Base ./reducedim.jl:334
              _mapreduce(f::typeof(identity), op::typeof(Base.add_sum),
                ::IndexLinear, A::Vector{Any}) @ Base ./reduce.jl:436
```

```

||||||| ↳ mapreduce_impl(f::typeof(identity), op::typeof(Base.add_sum),
↳   A::Vector{Any}, ifirst::Int64, ilast::Int64) @ Base ./reduce.jl:269
||||||| ↳ mapreduce_impl(f::typeof(identity), op::typeof(Base.add_sum),
↳   A::Vector{...}, ifirst::Int64, ilast::Int64, blksize::Int64) @ Base
↳   ./reduce.jl:249
|||||||   runtime dispatch detected: Base.mapreduce_first(f::typeof(identity),
↳   op::typeof(Base.add_sum), %19::Any)::Any
|||||||   ↳
|||||||   ↳ mapreduce_impl(f::typeof(identity), op::typeof(Base.add_sum),
↳   A::Vector{...}, ifirst::Int64, ilast::Int64, blksize::Int64) @ Base
↳   ./reduce.jl:254
|||||||   runtime dispatch detected: op::typeof(Base.add_sum)(%42::Any,
↳   %61::Any)::Any
|||||||   ↳
|||||||   ↳ mapreduce_impl(f::typeof(identity), op::typeof(Base.add_sum),
↳   A::Vector{...}, ifirst::Int64, ilast::Int64, blksize::Int64) @ Base
↳   ./reduce.jl:265
|||||||   runtime dispatch detected: op::typeof(Base.add_sum)(%138::Any,
↳   %140::Any)::Any
|||||||   ↳
|||||||   ↳ ↳ _mapreduce(f::typeof(identity), op::typeof(Base.add_sum),
↳     ::IndexLinear, A::Vector{Any}) @ Base ./reduce.jl:424
|||||||       runtime dispatch detected: Base.mapreduce_first(f::typeof(identity),
↳     op::typeof(Base.add_sum), %26::Any)::Any
|||||||       ↳
|||||||       ↳ _mapreduce(f::typeof(identity), op::typeof(Base.add_sum),
↳         ::IndexLinear, A::Vector{Any}) @ Base ./reduce.jl:429
|||||||           runtime dispatch detected: op::typeof(Base.add_sum)(%48::Any,
↳           %66::Any)::Any
|||||||           ↳
|||||||           ↳ _mapreduce(f::typeof(identity), op::typeof(Base.add_sum),
↳             ::IndexLinear, A::Vector{Any}) @ Base ./reduce.jl:432
|||||||               runtime dispatch detected: op::typeof(Base.add_sum)(%69::Any,
↳               %90::Any)::Any
|||||||               ↳

```

 Tip

The Julia extension features a static linter, and runtime diagnostics with JET can be automated to run periodically on your codebase and show any problems detected.

`Cthulhu.jl`¹ exposes the `@descend` macro which can be used to interactively “step through” lines of the corresponding typed code, and “descend” into a particular line if needed. This is akin to repeatedly calling `@code_warntype` deeper and deeper into your functions (“slowly succumbing to the madness...” of type instability).

¹So-named for the “slow descent into madness” when descending into functions to follow the Julia compiler’s type inference across many layers of function calls.

24.6.2. Fixing Instabilities

The Julia manual has a collection of tips to improve type inference.

 Tip

To be more forceful about ensuring type stability in your code, one approach is to error whenever a type instability occurs: the macro `@stable` from `DispatchDoctor.jl` allows exactly that.

When working with parametric types, avoid using generic type parameters (e.g., `Array{Any}`) whenever possible. For custom types, make use of parametric types to create type-stable abstractions.

In the next example, `Bond1` stores fields as `Any`, which forces dynamic dispatch when accessing fields and often leads to allocations. `Bond2` fixes the field types to `Float64`, which is concrete but inflexible. `Bond3` is parametric: it's concrete for each `T` (e.g., `Bond3{Float64}`), allowing specialization while remaining flexible.

```
struct Bond1
    par
    coupon
end

struct Bond2
    par :: Float64
    coupon :: Float64
end

struct Bond3{T}
    par :: T
    coupon :: T
end

b1 = Bond1(100.0, 0.05)           # fields are Any → slower in numeric code
b2 = Bond2(100.0, 0.05)           # concrete and fast, but fixed to Float64
b3 = Bond3(100.0, 0.05)           # specializes to Bond3{Float64}
```

24.7. Memory management

After ensuring type stability, one should try to reduce the number of heap allocations a program makes. Again, the Julia manual has a series of tricks related to arrays and allocations that you should review. In particular, try to modify existing arrays instead of allocating new objects, and try to access arrays in the right order for Julia, i.e., accessing data down columns instead of across rows.

Alternatively, to ensure that non-allocating functions never regress in future versions of your code, you can write a test set to check allocations by providing the function and a concrete type signature.

```
using AllocCheck, Test
@testset "non-allocating" begin
    @test isempty(AllocCheck.check_allocs(my_func, (Float64, Float64)))
```

```
    end
```

24.8. Compilation

A number of tools allow you to reduce Julia's latency, also referred to as TTFX (time to first X, where X was historically plotting a graph).

24.8.1. Precompilation

PrecompileTools.jl reduces the amount of time taken to run functions loaded from a package or local module that you wrote. It allows module authors to specify methods to precompile when a module is loaded for the first time. The methods chosen should represent those which would be used during typical user workflows. These methods then have the same latency as if they had already been run by the user. This adds upfront pre-compilation time when installing a package version for the first time, but then subsequent uses will be much quicker.

Here's an example of precompilation, adapted from the package's documentation:

```
module MyPackage

using PrecompileTools: @compile_workload

struct MyType
    x::Int
end

myfunction(a::Vector) = a[1].x

@compile_workload begin
    a = [MyType(1)]
    myfunction(a)
end
end
```

Note that every method that is called will be compiled, no matter how far down the call stack or which module it comes from. To see if the intended calls were compiled correctly or diagnose other problems related to precompilation, use SnoopCompile.jl. This is especially important for writers of registered Julia packages, as it allows you to diagnose recompilation that happens due to invalidation.

24.8.2. Package compilation

To reduce the time that packages take to load, you can use PackageCompiler.jl to generate a custom version of Julia, called a **sysimage** (system image), with its own custom standard library. As packages in the standard library are already compiled, any `using` or `import` statement involving them is almost instant.

Once `PackageCompiler.jl` is added to your global environment, activate a local environment for which you want to generate a sysimage, ensure all of the packages you want to compile are in its `Project.toml`, and run `create_sysimage` as in the example below. The filetype of `sysimage_path` differs by operating system: Linux has `.so`, MacOS has `.dylib`, and Windows has `.dll`.

24. Optimizing Julia Code

```
using PackageCompiler # installed in global environment
packages_to_compile = ["Makie", "DifferentialEquations"]
create_sysimage(packages_to_compile; sysimage_path="MySysimage.so")
```

Once a sysimage is generated, it can be used with the command-line flag: `julia --sysimage=path/to/sysimage`.

💡 Tip

The generation and loading of sysimages can be streamlined with VSCode. By default, the command sequence Task: Run Build Task followed by Julia: Build custom sysimage for current environment will compile a sysimage containing all packages in the current environment, but additional details can be specified in a `./vscode/JuliaSysimage.toml` file. To automatically detect and use a custom sysimage, set `useCustomSysimage` to true in the application settings.

24.8.3. Static compilation

`PackageCompiler.jl` also facilitates the creation of apps and libraries that can be shared to and run on machines that don't have Julia installed.

At a basic level, all that's required to turn a Julia module `MyModule` into an app is a function `julia_main()::Cint` that returns `0` upon successful completion. Then, with `PackageCompiler.jl` loaded, run `create_app("MyModule", "MyAppCompiled")`. Command-line arguments to the resulting app are assigned to the global variable `ARGS::Vector{String}`, the handling of which can be made easier by `ArgParse.jl`.

In Julia, a library is just a sysimage with some extras that enable external programs to interact with it. Any functions in a module marked with `Base.@ccallable` and whose type signature involves C-conforming types (e.g., `Cint`, `Cstring`, and `Cvoid`) can be compiled into an externally callable library with `create_library`, similar to `create_app`. Unfortunately, the process of compiling and sharing a standalone executable or callable library must take relocatability into account, which is beyond the scope of this book.

ℹ Note

An alternative way to compile a shareable app or library that doesn't need to compile a sysimage and therefore results in smaller binaries is to use `StaticCompiler.jl` and its sister package `StaticTools.jl`. The biggest tradeoff of not compiling a sysimage is that Julia's garbage collector is no longer included, so all heap allocations must be managed manually and all compiled code *must* be type-stable. To work around this limitation, you can use static equivalents of dynamic types, such as a `StaticArray` (`StaticArrays.jl`) instead of an `Array` or a `StaticString` (`StaticTools.jl`), use `malloc` and `free` from `StaticTools.jl` directly, or use arena allocators with `Bumper.jl`. The README of `StaticCompiler.jl` contains a more detailed guide on how to prepare code to be compiled.

ℹ Note

Starting with Julia 1.12, there is a new way to compile Julia to small, static binaries with a tool called `juliac`.

24.9. Parallelism

Code can be made to run faster through parallel execution with multithreading (shared-memory parallelism) or multiprocessing/distributed computing. Parallelism was covered in a whole chapter (Chapter 11), but this section provides insight into recommended packages and patterns specific to Julia.

Many common operations such as maps and reductions can be trivially parallelized through either method by using their respective Julia packages (e.g., `pmap` from `Distributed.jl` and `tmap` from `OhMyThreads.jl`). Multithreading is available on almost all modern hardware, whereas distributed computing is most useful to users of high-performance computing clusters.

24.9.1. Multithreading

To use multi-threading, Julia needs to be started with more than one thread. This can be done by setting the environment variable `JULIA_NUM_THREADS` to either `auto` or a specific number like `4`. You can also specify how many threads to start Julia with using the `-t` command-line argument (such as running `julia -t 4` to start Julia with four threads from the command line). Once Julia is running, you can check if this was successful by calling `Threads.threads()`.

 Tip

In VS Code, the default number of threads can be edited by adding "julia.NumThreads": `auto`, to your settings. This will be applied to the integrated terminal. Setting the threads at the machine level through the environment variables is preferred, since it will apply that setting to more than just your VS Code sessions.

Why doesn't Julia automatically start with more than one thread? Between "hyper-threading" (synthetic additional thread capacity), multi-core architectures, and the different types of threads, it's actually difficult to predict how many threads will be optimal for a given system. Julia's current default is to take the more conservative approach and start single-threaded unless otherwise specified. The "auto" option is a best guess but can, on certain systems and configurations, be very bad for performance. The authors recommend for most common systems to just use "auto".

 Tip

Linear algebra code calls the low-level libraries BLAS and LAPACK. These libraries manage their own pool of threads, so single-threaded Julia processes can still make use of multiple threads. If you also enable Julia threads, you may oversubscribe cores (Julia threads \times BLAS threads) which can hurt performance. Consider limiting BLAS to one thread when using Julia threads.

In this case, once `LinearAlgebra` is loaded, BLAS can be set to use only one thread by calling `BLAS.set_num_threads(1)`. For more information see the docs on multithreading and linear algebra.

Regardless of the number of threads, you can parallelize a `for` loop with the macro `Threads.@threads`. The macros `@spawn` and `@async` function similarly, but require more manual management of tasks and their results. For this reason `@threads` is recommended for those who do not wish to use third-party packages.

When designing multithreaded code, you should generally try to write to shared memory as rarely as possible. Where it cannot be avoided, you need to be careful to avoid "race conditions",

24. Optimizing Julia Code

i.e. situations when competing threads try to write different things to the same memory location. It is usually a good idea to separate memory accesses with loop indices, as in the example below:

```
results = zeros(Int, 4)
Threads.@threads for i in 1:4
    results[i] = i^2
end
```

Almost always, it is **not** a good idea to use `threadid()`.

Even if you manage to avoid any race conditions in your multithreaded code, it is very easy to run into subtle performance issues (like false sharing). For these reasons, you might want to consider using a high-level package like `OhMyThreads.jl`, which provides a user-friendly alternative to `Threads` and makes managing threads and their memory use much easier. The helpful translation guide demonstrates how Base multi-threading loops can be translated into the `OhMyThreads` API.

If the latency of spinning up new threads becomes a bottleneck, check out `Polyester.jl` for very lightweight threads that are quicker to start.

If you're on Linux, you should consider using `ThreadPinning.jl` to pin your Julia threads to CPU cores to obtain stable and optimal performance. The package can also be used to visualize where the Julia threads are running on your system (see `threadinfo()`).

24.9.2. Distributed computing

Julia's multiprocessing and distributed relies on the standard library `Distributed`. The main difference from multi-threading is that data isn't shared between worker processes. Once Julia is started, processes can be added with `addprocs` and their number can be queried with `nworkers`.

The macro `Distributed.@distributed` is a *syntactic* equivalent for `Threads.@threads`. Hence, we can use `@distributed` to parallelize a for loop as before, but we have to additionally deal with sharing and recombining the results array. We can delegate this responsibility to the standard library `SharedArrays`. However, in order for all workers to know about a function or module, we have to load it `@everywhere`:

```
using Distributed

# Add additional workers then load code on the workers
addprocs(3)
@everywhere using SharedArrays
@everywhere f(x) = 3x^2

results = SharedArray{Int}(4)
@sync @distributed for i in 1:4
    results[i] = f(i)
end
results

4-element SharedVector{Int64}:
 3
12
27
48
```

Note that `@distributed` does not force the main process to wait for other workers; we must use `@sync` to block execution until all computations are done.

One feature `@distributed` has over `@threads` is the possibility to specify a reduction function (an associative binary operator) that combines the results of each worker. In this case, `@sync` is implied, as the reduction cannot happen unless all of the workers have finished.

```
using Distributed
addprocs(2)

2-element Vector{Int64}:
5
6

@distributed (+) for i in 1:4
    i^2
end

30
```

Alternatively, the convenience function `pmap` can be used to easily parallelize a `map`, both in a distributed and multi-threaded way.

```
results = pmap(x -> 3 * x^2, 1:100; distributed=true, batch_size=25,
                on_error=ex -> 0)

100-element Vector{Int64}:
3
12
27
48
75
108
147
192
243
300
363
432
507
⋮
23763
24300
24843
25392
25947
26508
27075
27648
28227
28812
29403
30000
```

For more functionalities related to higher-order functions, `Transducers.jl` and `Folds.jl` are the way to go.

 Tip

MPI.jl implements the Message Passing Interface standard, which is heavily used in high-performance computing beyond Julia. The C library that MPI.jl wraps is *highly* optimized, so Julia code that needs to be scaled up to a large number of cores, such as an HPC cluster, will typically run faster with MPI than with plain Distributed.

Elemental.jl is a package for distributed dense and sparse linear algebra which wraps the Elemental library written in C++, itself using MPI under the hood.

24.9.3. GPU programming

GPU programming involves GPUs that specialize in executing instructions in parallel over a large number of threads. While they were originally designed for accelerating graphics rendering, more recently they have been used to train and evaluate machine learning models.

Julia's GPU ecosystem is managed by the JuliaGPU organization, which provides individual packages for directly working with each GPU vendor's instruction set. The most popular one is CUDA.jl, which also simplifies installation of CUDA drivers for NVIDIA GPUs. Through KernelAbstractions.jl, you can easily write code that is agnostic to the type of GPU where it will run.

24.9.4. SIMD instructions

In the Single Instruction, Multiple Data (SIMD) paradigm, several processing units perform the same instruction at the same time, differing only in their inputs. The range of operations that can be parallelized (or "vectorized") in this way is more limited than in the previous sections and slightly harder to control. Julia can automatically vectorize repeated numerical operations (such as those found in loops) provided a few conditions are met:

1. Reordering operations must not change the result of the computation.
2. There must be no control flow or branches in the core computation.
3. All array accesses must follow some linear pattern.

While this may seem straightforward, there are a number of important caveats which prevent code from being vectorized. Performance annotations like `@simd` or `@inbounds` help enable vectorization in some cases, as does replacing control flow with `ifelse`.

If this isn't enough, SIMD.jl allows users to force the use of SIMD instructions and bypass the check for whether this is possible. One particular use-case for this is for vectorizing non-contiguous memory reads and writes through `SIMD.vgather` and `SIMD.vscatter` respectively.

 Tip

You can detect whether the optimizations have occurred by inspecting the output of `@code_llvm` or `@code_native` and looking for vector registers (e.g., `ymm/xmm/zmm`), packed operations, and vectorized loops. Note that the exact things you're looking for will vary between code and CPU instruction set; an example of what to look for can be seen in this blog post by Kristoffer Carlsson.

24.9.5. Additional Packages

Some additional packages to be aware of include:

- LoopVectorization.jl which can enhance vectorized loops even further, such as handling the “tail” of vectorized loops more efficiently than the base compiler. The “tail” refers to situations like where you have a vector width of 8, but don’t have a collection that’s a nice multiple of 8 (say 1001 elements).
- Octavian.jl implements a linear algebra-like library, utilizing parallelism via vectorization to generate efficient code for the system it’s running on.
- Tullio.jl is an einsum library, a domain-specific language for tensor operations, common in machine learning and linear algebra.

24.10. Efficient types

Using an efficient data structure is a tried-and-true way of improving performance. While users can write their own efficient implementations through officially documented interfaces, a number of packages containing common use cases are more tightly integrated into the Julia ecosystem.

24.10.1. Static arrays

Using StaticArrays.jl, you can construct arrays that contain size information in their type. Through multiple dispatch, statically sized arrays give rise to specialised, efficient methods for certain algorithms like linear algebra. In addition, the SArray, SMatrix, and SVector types are immutable, so the array does not need to be garbage collected as it can be stack-allocated. Creating new SArrays comes at almost no extra cost, compared to directly editing the data of a mutable object. With MArray, MMatrix, and MVector, data remains mutable as in normal arrays.

To handle mutable and immutable data structures with the same syntax, you can use Accessors.jl:

```
using StaticArrays, Accessors

sx = SA[1, 2, 3] # SA constructs an SArray
@set sx[1] = 3 # Returns a copy, does not update the variable
@reset sx[1] = 4 # Replaces the original

3-element SVector{3, Int64} with indices SOneTo(3):
4
2
3
```

24.10.2. Classic data structures

All but the most obscure data structures can be found in the packages from the JuliaCollections organization, especially DataStructures.jl which has all the standards from the computer science courses (stacks, queues, heaps, trees and the like). Iteration and memoization utilities are also provided by packages like IterTools.jl and Memoize.jl.

24.10.3. Bits types

When you create custom `structs`, keeping the fields as simple, concrete types makes it more likely that the compiler will be able to allocate these objects on the stack instead of the heap. An example of this was shown in Section 13.5.3.

24.10.4. Putting it into practice

In day-to-day modeling work you typically pull these threads together:

1. **Measure** with `@btime` or `@benchmark` on representative data (e.g., a block of policies or a slice of scenarios).
2. **Profile** the slow sections (runtime and allocations) to see where time is spent.
3. **Inspect types** with `@code_warntype`, JET, or Cthulhu to ensure the hot paths remain type-stable.
4. **Refactor** data structures (make concrete, preallocate, use `StaticArrays` where small shapes are common) and loop order to improve cache locality.
5. **Repeat** until the code meets SLA requirements—whether that’s a daily risk report before 6 a.m. or a pricing request that must return within seconds.

Keeping these steps in mind turns optimization from “mysterious black art” into a deliberate process aligned with business deadlines.

Part VII.

Applied Financial Modeling Techniques

This section focuses on practical implementation: concrete examples and strategies for building effective models across various applications. We cover model design, optimization, and validation, bridging theory and practice for actual financial challenges.

25. Scenario Generation

CHAPTER AUTHORED BY: ALEC LOUDENBACK, YUN-TIEN LEE

"Scenarios are the rehearsal of possible futures. They are not predictions, but pathways that help us prepare for the unknown." — Peter Schwartz

25.1. Chapter Overview

Scenario generation is the discipline of simulating coherent paths for the economic and market variables that feed valuation, risk, and planning models. You will often combine several stochastic sub-models (interest rates, equity returns, credit spreads, inflation, demographics) to create realistic joint scenarios. This chapter reviews commonly used building blocks and illustrates their Julia implementations so you can stitch together scenario engines for portfolio analytics, risk transfer pricing, or asset-liability management (ALM).

25.2. Common Use Cases of Scenario Generators

Scenario generators are widely used in risk management, investment analysis, and regulatory compliance to model potential future outcomes. For forecasting under the real-world (physical) measure P , use real-world (RW) scenarios. For valuation and hedging, use risk-neutral (RN) scenarios under the martingale measure Q (e.g., with drift adjusted to the short rate minus dividends for equities and to fit the initial curve for interest rates).

25.2.1. Risk management, especially Value at Risk (VaR) & Expected Shortfall (ES).

RW scenario generators are used to simulate market movements to estimate potential portfolio losses. Basel III regulatory capital requirements have adopted these approaches.

25.2.2. Stress Testing & Regulatory Compliance

RW scenario generators can also be used to generate extreme but plausible market conditions to assess resilience, which is required by central banks and financial regulators (e.g., Federal Reserve and ECB).

25.2.3. Portfolio Optimization & Asset Allocation

RW scenario generators are used to simulate thousands of market conditions to determine optimal portfolio allocations, which is commonly used in modern portfolio theory (MPT) and Black-Litterman models.

25. Scenario Generation

25.2.4. Pension & Insurance Risk Modeling

RW scenario generators can be used to simulate longevity risk, policyholder behavior, and interest rate movements. They are also used for economic capital estimation under uncertain economic scenarios.

25.2.5. Economic & Macro-Financial Forecasting

Central banks and institutions (e.g., IMF, World Bank) use RW scenario generators to predict macroeconomic trends.

25.2.6. Asset Pricing & Hedging

RN scenario generators help value options using stochastic models (e.g., Black-Scholes model). They can help simulate future stock price movements under different volatility conditions. They can also be used for hedging purposes to test how a portfolio performs under different inflation, interest rate, or commodity price scenarios.

25.2.7. Fixed Income & Interest Rate Modeling

Yield curve modeling uses RN scenarios to value bonds and interest rate derivatives. Swaps, swaptions, and credit default swaps (CDS) also rely on RN pricing. RN scenario generators can also simulate yield curves for bond and fixed-income pricing. Models like Cox-Ingersoll-Ross (CIR) or Hull-White generate future interest rate paths.

25.2.8. Regulatory & Accounting Valuations

IFRS 13 & fair value accounting use RN models to determine the market-consistent value of liabilities. Solvency II for insurers requires valuation of policyholder guarantees using RN scenarios.

25.3. Common Economic Scenario Generation Approaches

Economic scenario generation involves the development of plausible future economic scenarios to assess the potential impact on financial portfolios, investments, or decision-making processes. Various approaches are used to generate economic scenarios, such as adapting underlying stochastic differential equations (SDEs) for Monte Carlo scenario generation techniques.

25.3.1. Interest Rate Models

Short-rate models produce mean-reverting simulated rates that can be bootstrapped into full yield curves. They remain popular in ALM where projecting discount curves, reinvestment rates, and collateral flows is critical.

25.3.1.1. Vasicek and Cox-Ingersoll-Ross (CIR)

The Vasicek model is a one-factor model commonly used for simulating interest rate scenarios. It describes the dynamics of short-term interest rates using a stochastic differential equation (SDE). In a Monte Carlo simulation, we can use the Vasicek model to generate multiple interest rate paths. The CIR model is an extension of the Vasicek model with non-constant volatility. It is a square-root diffusion with state-dependent volatility that helps keep rates positive (strictly so when the Feller condition $2\kappa\theta \geq \sigma^2$ holds).

Vasicek is defined as

$$dr(t) = \kappa(\theta - r(t)) dt + \sigma dW(t)$$

where

- $r(t)$ is the short-term interest rate at time t .
- κ is the speed of mean reversion, representing how quickly the interest rate reverts to its long-term mean.
- θ is the long-term mean or equilibrium level of the interest rate.
- σ is the volatility of the interest rate.
- $dW(t)$ is a Wiener process or Brownian motion, representing a random shock.

And CIR is defined as

$$dr(t) = \kappa(\theta - r(t)) dt + \sigma\sqrt{r(t)} dW(t)$$

where

- $r(t)$ is the short-term interest rate at time t .
- κ is the speed of mean reversion, representing how quickly the interest rate reverts to its long-term mean.
- θ is the long-term mean or equilibrium level of the interest rate.
- σ is the volatility of the interest rate.
- $dW(t)$ is a Wiener process or Brownian motion, representing a random shock.

The following code shows a simplified implementation of a CIR model. The specification of dr can be changed to make it a Vasicek model.

```
using Random, Statistics, CairoMakie

# Function to simulate CIR process
function cir_simulation(
    rng::AbstractRNG, κ, θ, σ, r₀, Δt, num_steps, num_sims)
    interest_rate_paths = zeros(num_steps, num_sims)
    st = sqrt(Δt)
    for j in 1:num_sims
        shocks = randn(rng, num_steps - 1)
        interest_rate_paths[1, j] = r₀
        for i in 2:num_steps
            # dr = κ*(θ - r)*Δt + σ*dW for Vasicek
            r_prev = interest_rate_paths[i-1, j]
            drift = κ * (θ - r_prev) * Δt
            diffusion = σ * sqrt(max(r_prev, 0.0)) * st * shocks[i-1]
            dr = drift + diffusion
            # Ensure non-negativity
            if dr < 0
                interest_rate_paths[i, j] = 0
            else
                interest_rate_paths[i, j] = dr
            end
        end
    end
end
```

25. Scenario Generation

```
        interest_rate_paths[i, j] = max(r_prev + dr, 0.0)
    end
end
return interest_rate_paths
end

let
# Set seed for reproducibility
rng = MersenneTwister(1234)

# CIR model parameters
κ = 0.2 # Speed of mean reversion
θ = 0.05 # Long-term mean
σ = 0.1 # Volatility

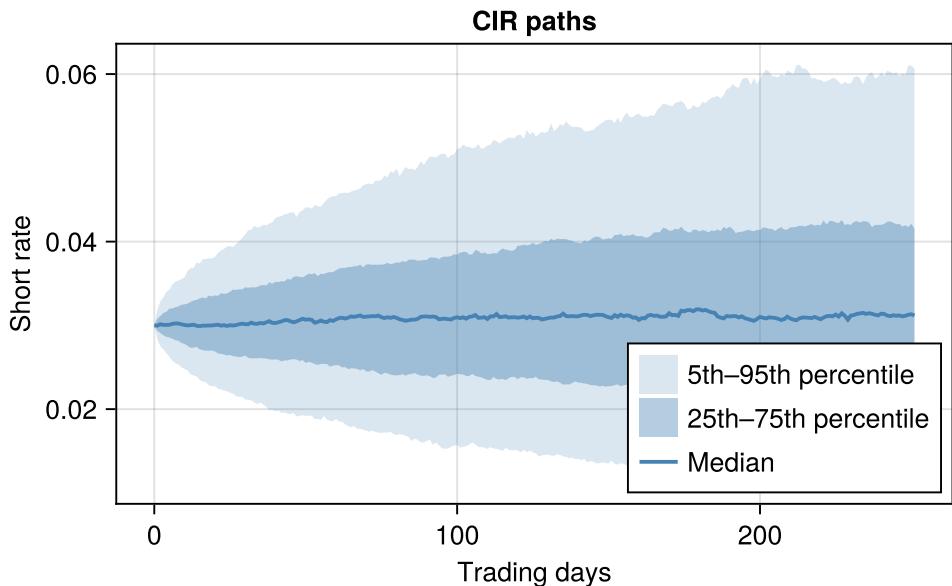
# Initial short-term interest rate
r₀ = 0.03

# Number of time steps and simulations
num_steps = 252
num_simulations = 1_000

# Time increment
Δt = 1 / 252
# Run CIR simulation
cir_paths = cir_simulation(rng, κ, θ, σ, r₀, Δt, num_steps,
← num_simulations)

# Compute percentiles across simulations at each time step
ti = 0:(num_steps-1)
p05 = [quantile(cir_paths[i, :], 0.05) for i in 1:num_steps]
p25 = [quantile(cir_paths[i, :], 0.25) for i in 1:num_steps]
p50 = [quantile(cir_paths[i, :], 0.50) for i in 1:num_steps]
p75 = [quantile(cir_paths[i, :], 0.75) for i in 1:num_steps]
p95 = [quantile(cir_paths[i, :], 0.95) for i in 1:num_steps]

# Plot shaded bands
f = Figure()
ax = Axis(f[1, 1], xlabel="Trading days",
          ylabel="Short rate", title="CIR paths")
band!(ax, ti, p05, p95, color=(:steelblue, 0.2), label="5th-95th
← percentile")
band!(ax, ti, p25, p75, color=(:steelblue, 0.4), label="25th-75th
← percentile")
lines!(ax, ti, p50, color=:steelblue, linewidth=2, label="Median")
axislegend(ax, position=:rb)
f
end
```



The exact CIR process has a non-central chi-squared distribution for $r(t)$ in closed form.

$$\begin{aligned} d &= \frac{4\kappa\theta}{\sigma^2} \\ c &= \frac{\sigma^2(1 - e^{-\kappa\Delta t})}{4\kappa} \\ \lambda &= \frac{4\kappa e^{-\kappa\Delta t} r_t}{\sigma^2(1 - e^{-\kappa\Delta t})} \\ r_{t+\Delta t} &\sim c \cdot \chi_d'^2(\lambda) \end{aligned}$$

where $\chi_d'^2(\lambda)$ means the non-central chi-square distribution with d degrees of freedom and a non-centrality parameter λ . The following code shows an exact implementation of a CIR model.

```
using Random, Statistics, CairoMakie, Distributions

function cir_exact_paths(
    rng::AbstractRNG, κ, θ, σ, r₀, T, num_steps, num_sims)
    Δt = T / num_steps
    d = 4 * κ * θ / (σ * σ)
    c = (σ * σ * (1 - exp(-κ * Δt))) / (4 * κ)
    rates = zeros(Float64, num_steps + 1, num_sims)
    time = range(0, T; length=num_steps + 1)
    rates[1, :] .= r₀
    for j in 1:num_sims
        r = r₀
        for i in 2:length(time)
            λ = (4 * κ * exp(-κ * Δt) * r) / (σ * σ * (1 - exp(-κ * Δt)))
            r = c * rand(rng, NoncentralChisq(d, λ))
            rates[i, j] = r
        end
    end
end
```

25. Scenario Generation

```
        end
    end
    return time, rates
end

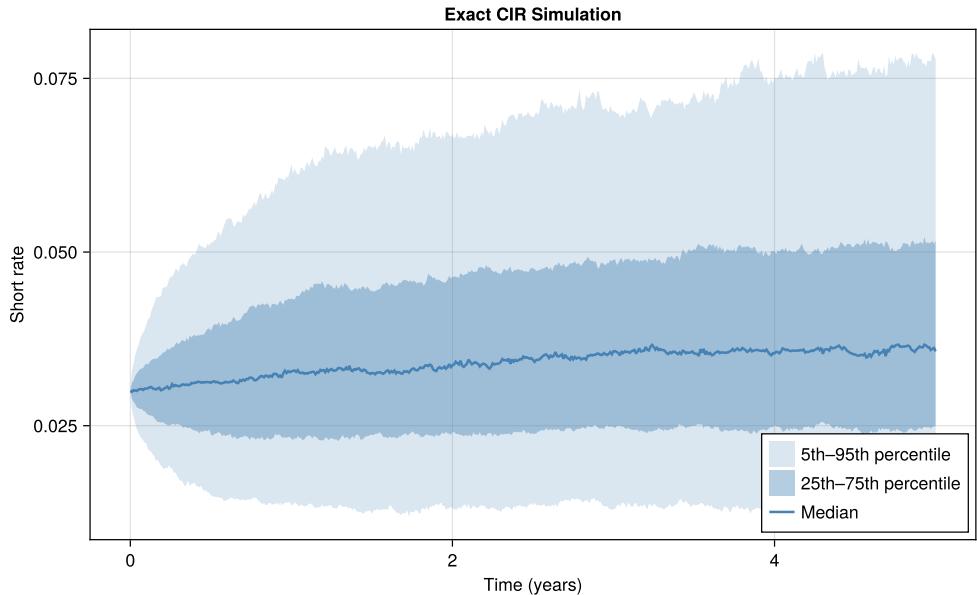
let
    rng = MersenneTwister(1234)

    # CIR parameters
    κ = 0.5 # mean reversion speed
    θ = 0.04 # long-term mean
    σ = 0.1 # volatility coefficient
    r₀ = 0.03 # initial rate
    T = 5.0 # years
    nsteps = 500 # time steps
    nsims = 1000 # number of simulated paths

    tgrid, rpaths = cir_exact_paths(rng, κ, θ, σ, r₀, T, nsteps, nsims)

    # Compute percentiles across simulations at each time step
    nsteps_total = length(tgrid)
    p05 = [quantile(rpaths[i, :], 0.05) for i in 1:nsteps_total]
    p25 = [quantile(rpaths[i, :], 0.25) for i in 1:nsteps_total]
    p50 = [quantile(rpaths[i, :], 0.50) for i in 1:nsteps_total]
    p75 = [quantile(rpaths[i, :], 0.75) for i in 1:nsteps_total]
    p95 = [quantile(rpaths[i, :], 0.95) for i in 1:nsteps_total]

    # Plot shaded bands
    fig = Figure(size=(800, 500))
    ax = Axis(fig[1, 1], xlabel="Time (years)",
              ylabel="Short rate", title="Exact CIR Simulation")
    band!(ax, collect(tgrid), p05, p95, color=:steelblue, 0.2,
          label="5th-95th percentile")
    band!(ax, collect(tgrid), p25, p75, color=:steelblue, 0.4,
          label="25th-75th percentile")
    lines!(ax, collect(tgrid), p50, color=:steelblue, linewidth=2,
          label="Median")
    axislegend(ax, position=:rb)
    fig
end
```



25.3.1.2. Hull-White

The Hull-White model is a one-factor short-rate model that extends the Vasicek model by allowing the mean reversion level and volatility to be time-dependent. It is commonly used for pricing interest rate derivatives. A separate class of models, the Libor Market Models (LMMs) such as the Brace-Gatarek-Musiela (BGM) model, instead model the evolution of forward rates directly rather than the short rate. The Hull-White model is defined as

$$dr(t) = (\theta(t) - ar(t)) dt + \sigma(t) dW(t)$$

where

- $r(t)$ is the short-term interest rate at time t .
- $\theta(t)$ is the long-term mean or calibrated market-implied equilibrium level of the interest rate.
- a is the speed of mean reversion.
- $\sigma(t)$ is the time-dependent volatility of the interest rate.
- $dW(t)$ is a Wiener process or Brownian motion, representing a random shock.

```
using DataInterpolations, Random, Statistics, CairoMakie

function hull_white_theta_timevar_sigma(years, prices, σt; a::Float64)
    n = length(years)
    @assert n == length(prices) == length(σt) "mismatched lengths"

    # Interpolate logP for smooth derivatives (cubic spline)
    logP = log.(prices)
    interp_logP = CubicSpline(logP, years)

    # instantaneous forward f(0,t) = -∂_t log P(0,t)
```

25. Scenario Generation

```

f0(t) = -DataInterpolations.derivative(interp_logP, t)
df0(t) = -DataInterpolations.derivative(interp_logP, t, 2)

# Precompute Δs_j (left Riemann widths).
# assumes years[1] == 0 or handles first width as years[1].
Δs = Vector{Float64}(undef, length(years))
Δs[1] = years[1] # if years[1]==0 this is zero; else left interval from 0
for j in 2:length(years)
    Δs[j] = years[j] - years[j-1]
end

# Compute integral term I(t_i) = 1/2 * ∫_0^{t_i} e^{-2a(t_i - s)} σ(s)^2
# ← ds
I = zeros(length(years))
for i in 1:length(years)
    ti = years[i]
    s = 0.0
    acc = 0.0
    # left Riemann: use sigma at s = years[j] for
    # interval [years[j], years[j+1])
    for j in 1:i
        sj = years[j]
        # width Δs[j] approximates length of
        # [sj, sj+1) (or [0, years[1]] for j=1)
        w = Δs[j]
        if w <= 0
            # if grid is pure points (years[1]==0 and Δs[1]==0),
            # skip zero-width
            continue
        end
        # integrand evaluated at left point sj
        integrand = exp(-2a * (ti - sj)) * (σt[j]^2)
        acc += integrand * w
    end
    I[i] = 0.5 * acc
end

# theta vector: df0 + a*f0 + I
θt = [df0(years[i]) + a * f0(years[i]) + I[i] for i in 1:length(years)]

return θt, f0, df0
end

let
    # Seed for reproducibility
    rng = MersenneTwister(1234)

    # Market curve setup (synthetic)
    years = collect(0.0:0.1:10.0)
    yields = 0.02 .+ 0.002 .* years
    prices = exp.(-yields .* years)

    # Parameters
    a = 0.1 # mean reversion speed

```

```

# sinusoidal time-varying sigma
σt = 0.008 .+ 0.004 .* sin.(0.5 .* years)
θt, f0, df0 = hull_white_theta_timevar_sigma(years, prices, σt; a=a)
r0 = 0.03 # initial short rate
nsim = 1000 # number of simulated paths

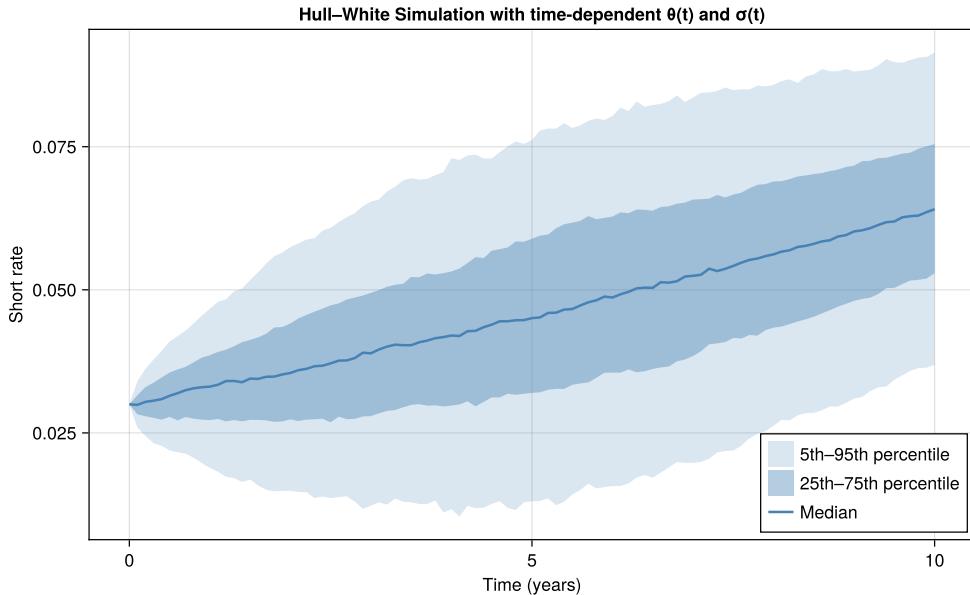
# Preallocate
rpaths = zeros(length(years), nsim)

# Euler-Maruyama simulation
for path in 1:nsim
    r = r0
    rpaths[1, path] = r
    for i in 2:length(years)
        dt = years[i] - years[i-1]
        dW = sqrt(dt) * randn(rng)
        drift = (θt[i-1] - a * r) * dt
        diffusion = σt[i-1] * dW
        r += drift + diffusion
        rpaths[i, path] = r
    end
end

# Compute percentiles across simulations at each time step
nsteps_total = length(years)
p05 = [quantile(rpaths[i, :], 0.05) for i in 1:nsteps_total]
p25 = [quantile(rpaths[i, :], 0.25) for i in 1:nsteps_total]
p50 = [quantile(rpaths[i, :], 0.50) for i in 1:nsteps_total]
p75 = [quantile(rpaths[i, :], 0.75) for i in 1:nsteps_total]
p95 = [quantile(rpaths[i, :], 0.95) for i in 1:nsteps_total]

# Plot shaded bands
fig = Figure(size=(800, 500))
ax = Axis(
    fig[1, 1],
    xlabel="Time (years)",
    ylabel="Short rate",
    title="Hull-White Simulation with time-dependent θ(t) and σ(t)")
band!(ax, years, p05, p95, color=:steelblue, 0.2, label="5th-95th
    ↵ percentile")
band!(ax, years, p25, p75, color=:steelblue, 0.4, label="25th-75th
    ↵ percentile")
lines!(ax, years, p50, color=:steelblue, linewidth=2, label="Median")
axislegend(ax, position=:rb)
fig
end

```



25.3.2. Equity Models

Equity scenario blocks usually mix a diffusion for price levels with a separate volatility process (to match option smiles) or with fat-tailed jump structures. We start with GBM for its closed-form intuition and then add a simple GARCH(1,1) volatility process.

25.3.2.1. Geometric Brownian Motion (GBM)

Geometric Brownian Motion (GBM) is a stochastic process commonly used to model the price movement of financial instruments, including stocks. It assumes constant volatility and is characterized by a log-normal distribution. It is defined as

$$dS(t) = \mu S(t) dt + \sigma S(t) dW(t)$$

where

- $S(t)$ is the stock price at time t .
- μ is the drift coefficient (expected return).
- σ is the volatility coefficient.
- $dW(t)$ is a Wiener process or Brownian motion, representing a random shock.

```
using Random, Statistics, CairoMakie

# Function to simulate GBM
function gbm_simulation(
    rng::AbstractRNG, μ, σ, S₀, Δt, num_steps, num_simulations)
    stock_price_paths = zeros(num_steps, num_simulations)
    drift = (μ - 0.5 * σ * σ) * Δt
    vol = σ * sqrt(Δt)
```

```

for j in 1:num_simulations
    stock_price_paths[1, j] = S₀
    shocks = randn(rng, num_steps - 1)
    for i in 2:num_steps
        S_prev = stock_price_paths[i-1, j]
        stock_price_paths[i, j] = S_prev * exp(drift + vol * shocks[i-1])
    end
end
return stock_price_paths
end

let
    # Set seed for reproducibility
    rng = MersenneTwister(1234)

    # GBM parameters
    μ = 0.05 # Drift (expected return)
    σ = 0.2 # Volatility

    # Initial stock price
    S₀ = 100.0

    # Number of time steps and simulations
    num_steps = 252
    num_simulations = 1_000

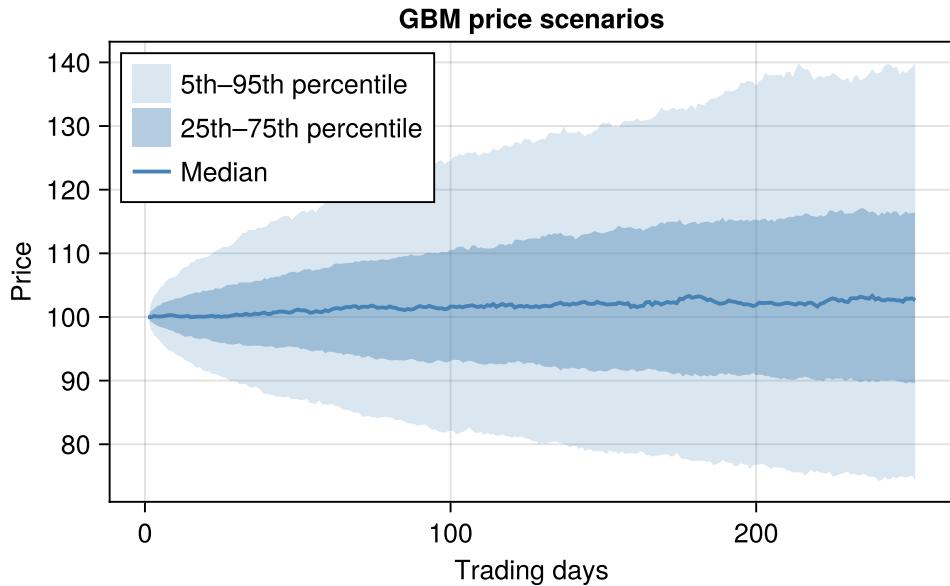
    # Time increment
    Δt = 1 / 252

    # Run GBM simulation
    gbm_paths = gbm_simulation(rng, μ, σ, S₀, Δt, num_steps, num_simulations)

    # Compute percentiles across simulations at each time step
    ti = 1:num_steps
    p05 = [quantile(gbm_paths[i, :], 0.05) for i in 1:num_steps]
    p25 = [quantile(gbm_paths[i, :], 0.25) for i in 1:num_steps]
    p50 = [quantile(gbm_paths[i, :], 0.50) for i in 1:num_steps]
    p75 = [quantile(gbm_paths[i, :], 0.75) for i in 1:num_steps]
    p95 = [quantile(gbm_paths[i, :], 0.95) for i in 1:num_steps]

    # Plot shaded bands
    f = Figure()
    ax = Axis(f[1, 1], xlabel="Trading days", ylabel="Price", title="GBM
    price scenarios")
    band!(ax, collect(ti), p05, p95, color=(:steelblue, 0.2), label="5th-95th
    percentile")
    band!(ax, collect(ti), p25, p75, color=(:steelblue, 0.4),
    label="25th-75th percentile")
    lines!(ax, collect(ti), p50, color=:steelblue, linewidth=2,
    label="Median")
    axislegend(ax, position=:lt)
    f
end

```



25.3.2.2. Generalized Autoregressive Conditional Heteroskedasticity (GARCH)

GARCH models capture time-varying volatility. They are often used in conjunction with other models to forecast volatility. The GARCH(1,1) model is defined as

$$\sigma_t^2 = \omega + \alpha_1 r_{t-1}^2 + \beta_1 \sigma_{t-1}^2$$

$$r_t = \varepsilon_t \sqrt{\sigma_t^2}$$

- σ_t^2 is the conditional variance at time t
- r_t is the return at time t
- ε_t is a white noise or innovation process
- $\omega, \alpha_1, \beta_1$ are model parameters

```
using Random, Statistics, CairoMakie

# Function to simulate GARCH(1,1) volatility
function garch_simulation(rng::AbstractRNG, ω, α₁, β₁, num_steps, num_sims)
    volatility_paths = zeros(num_steps, num_sims)
    σ²_ss = ω / (1 - α₁ - β₁)
    returns = zeros(Float64, num_steps, num_sims)
    for j in 1:num_sims
        returns[1, j] = volatility_paths[1, j] * randn(rng)
        for i in 2:num_steps
            σ² = ω + α₁ * returns[i-1, j] * returns[i-1, j] +
                β₁ * volatility_paths[i-1, j] * volatility_paths[i-1, j]
            volatility_paths[i, j] = sqrt(σ²)
            returns[i, j] = volatility_paths[i, j] * randn(rng)
        end
    end
end
```

```

        end
    end
    return volatility_paths
end

let
    # Set seed for reproducibility
    rng = MersenneTwister(1234)

    # GARCH(1,1) parameters
    ω = 0.01 # Constant term
    α₁ = 0.1 # Coefficient for lagged squared returns
    β₁ = 0.8 # Coefficient for lagged conditional volatility
    @assert α₁ + β₁ < 1 "Stationarity requires α₁ + β₁ < 1"

    # Number of time steps and simulations
    num_steps = 252
    num_simulations = 1_000

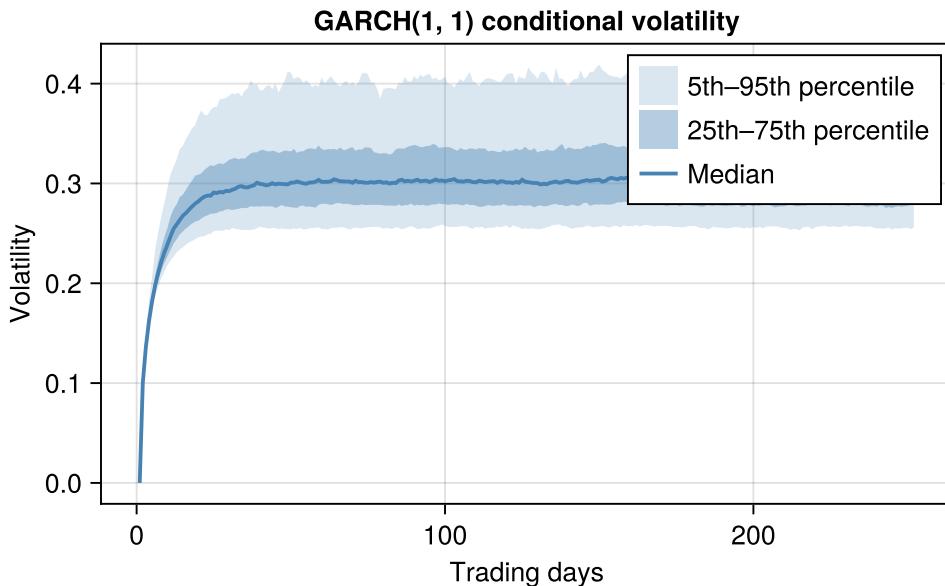
    # Time increment
    Δt = 1 / 252

    # Run GARCH simulation
    garch_paths = garch_simulation(rng, ω, α₁, β₁, num_steps,
    ↵ num_simulations)

    # Compute percentiles across simulations at each time step
    ti = 1:num_steps
    p05 = [quantile(garch_paths[i, :], 0.05) for i in 1:num_steps]
    p25 = [quantile(garch_paths[i, :], 0.25) for i in 1:num_steps]
    p50 = [quantile(garch_paths[i, :], 0.50) for i in 1:num_steps]
    p75 = [quantile(garch_paths[i, :], 0.75) for i in 1:num_steps]
    p95 = [quantile(garch_paths[i, :], 0.95) for i in 1:num_steps]

    # Plot shaded bands
    f = Figure()
    ax = Axis(
        f[1, 1],
        xlabel="Trading days", ylabel="Volatility",
        title="GARCH(1, 1) conditional volatility"
    )
    band!(ax, collect(ti), p05, p95, color=(:steelblue, 0.2), label="5th-95th
    ↵ percentile")
    band!(ax, collect(ti), p25, p75, color=(:steelblue, 0.4),
    ↵ label="25th-75th percentile")
    lines!(ax, collect(ti), p50, color=:steelblue, linewidth=2,
    ↵ label="Median")
    axislegend(ax, position=:rt)
    f
end

```



25.3.3. Copulas

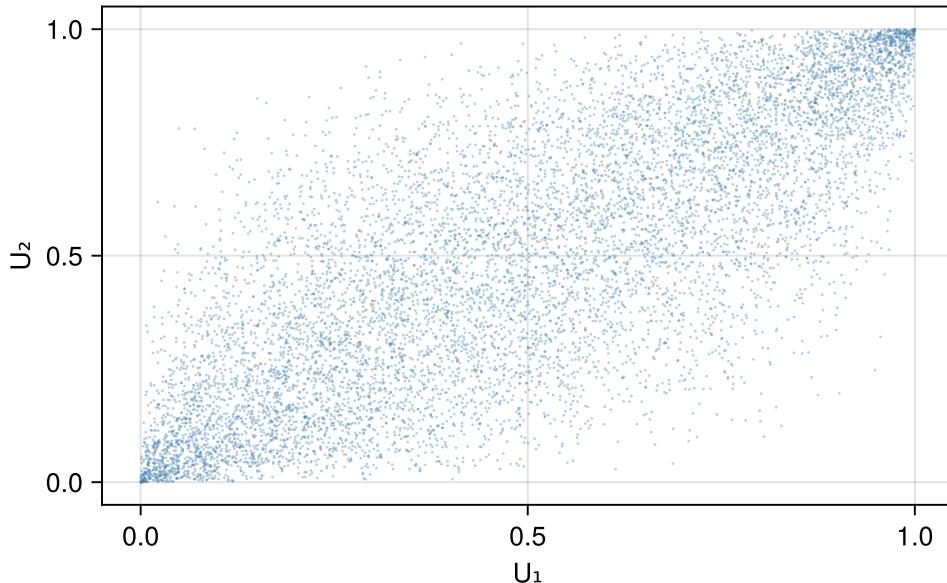
Copulas give us a flexible way to model dependence structures between random variables, independently from their marginal distributions. With copulas, we can model each variable's distribution (marginal) separately, and then "glue" them together with a copula to describe the dependence structure. This is useful when variables have very different distributions (e.g. stock returns vs. interest rates). Moreover, copulas can capture nonlinear dependence and tail dependence (how extreme events co-occur), whereas traditional correlation (e.g., Pearson) only captures linear dependence.

Simulating data using copulas involves generating multivariate samples with specified marginal distributions and a copula structure.

```
using Random, Copulas, CairoMakie

let
    Random.seed!(1234)

    # 2D Gaussian copula with correlation ρ = 0.8
    ρ = 0.8
    Σ = [1.0 ρ;
          ρ 1.0]
    Cg = GaussianCopula(Σ)
    U = rand(Cg, 10_000) # n × d matrix with U(0,1) margins
    f = Figure()
    ax = Axis(f[1, 1], xlabel="U1", ylabel="U2")
    scatter!(ax, U[1, :], U[2, :], markersize=2, color=:steelblue, alpha=0.4)
    f
end
```



Copulas can also be used to infer combined distributions from data samples.

```
using Random, Copulas, Distributions, CairoMakie

let
    X1 = Gamma(2.0, 3.0)
    X2 = Pareto(3.0, 1.0)
    X3 = LogNormal(0.0, 1.0)
    C = ClaytonCopula(3, 0.7) # A 3-variate Clayton Copula with θ = 0.7
    D = SklarDist(C, (X1, X2, X3)) # The final distribution

    # Generate a dataset (returns dimensions × observations)
    simu = rand(D, 1000)
    # Estimate marginals by MLE
    ð₁ = fit(Gamma, simu[1, :])
    ð₂ = fit(Pareto, simu[2, :])
    ð₃ = fit(LogNormal, simu[3, :])
    # Probability integral transform to uniforms
    Û = permutedims(
        hcat(cdf.(ð₁, simu[1, :]), cdf.(ð₂, simu[2, :]), cdf.(ð₃, simu[3,
            ↴ :])))
    )
    # Fit copula on uniforms
    Ĉ = fit(ClaytonCopula, Û)
    # Compose the estimated Sklar distribution
    Ð = SklarDist(Ĉ, (ð₁, ð₂, ð₃))

    # Generate new samples from the fitted distribution
    simu_fitted = rand(Ð, 1000)

    # Create visualizations
```

25. Scenario Generation

```
fig = Figure(size=(1000, 800))

# Plot pairwise scatter plots
ax1 = Axis(fig[1, 1], xlabel="X1 (Gamma)", ylabel="X2 (Pareto)",
           title="Original Data")
scatter!(ax1, simu[1, :], simu[2, :], markersize=3, color=:steelblue,
         alpha=0.5)

ax2 = Axis(fig[1, 2], xlabel="X1 (Gamma)", ylabel="X2 (Pareto)",
           title="Fitted Distribution")
scatter!(ax2, simu_fitted[1, :], simu_fitted[2, :], markersize=3,
         color=:coral, alpha=0.5)

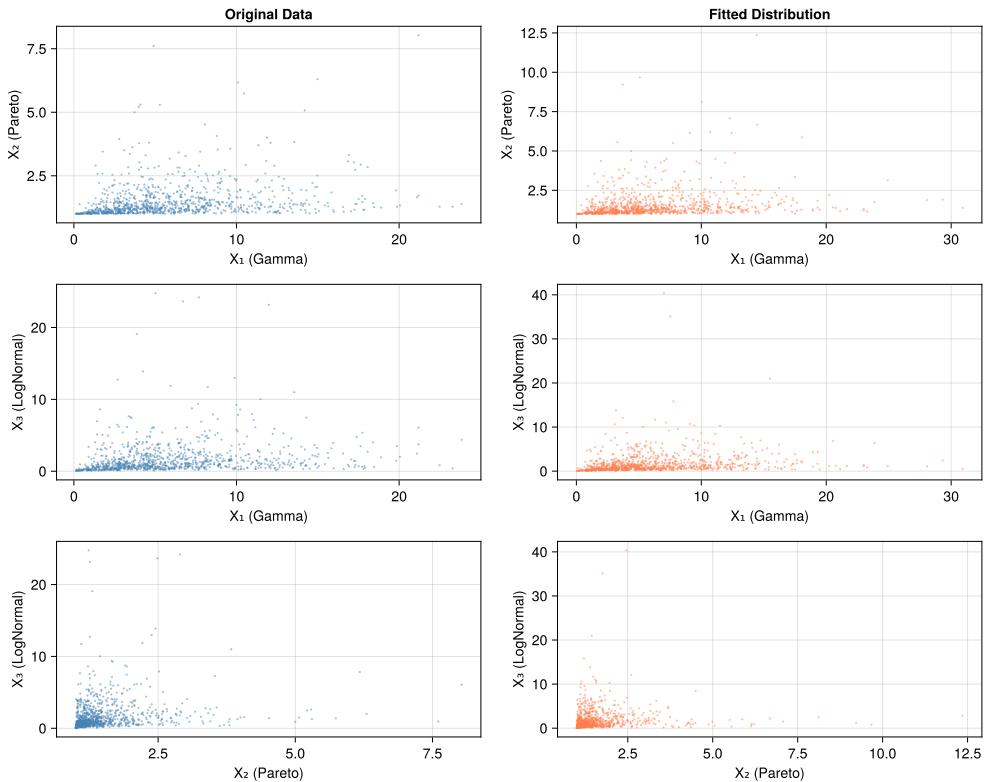
ax3 = Axis(fig[2, 1], xlabel="X1 (Gamma)", ylabel="X3 (LogNormal)")
scatter!(ax3, simu[1, :], simu[3, :], markersize=3, color=:steelblue,
         alpha=0.5)

ax4 = Axis(fig[2, 2], xlabel="X1 (Gamma)", ylabel="X3 (LogNormal)")
scatter!(ax4, simu_fitted[1, :], simu_fitted[3, :], markersize=3,
         color=:coral, alpha=0.5)

ax5 = Axis(fig[3, 1], xlabel="X2 (Pareto)", ylabel="X3 (LogNormal)")
scatter!(ax5, simu[2, :], simu[3, :], markersize=3, color=:steelblue,
         alpha=0.5)

ax6 = Axis(fig[3, 2], xlabel="X2 (Pareto)", ylabel="X3 (LogNormal)")
scatter!(ax6, simu_fitted[2, :], simu_fitted[3, :], markersize=3,
         color=:coral, alpha=0.5)

fig
end
```



25.4. Where to Go Next

Realistic enterprise scenario engines incorporate macro regime switches, stochastic volatility surfaces, credit migration, and balance-sheet feedback loops. Julia's multiple dispatch and composability make it straightforward to encapsulate each component (e.g., `AbstractInterestRateProcess`, `EquityBlock`, `CopulaLink`) and then assemble them into reproducible scenario trees or Monte Carlo streams. In subsequent chapters we will look at calibrating these models and feeding the resulting scenarios into portfolio analytics and ALM dashboards.

25. *Scenario Generation*

26. Similarity Analysis

CHAPTER AUTHORED BY: YUN-TIEN LEE

“By understanding the similarities, we unlock the potential for new insights, as patterns across different contexts often reveal hidden truths.” — Unknown

26.1. Chapter Overview

Similarity analysis helps us find “look-alike” portfolios, policies, or scenarios so we can transfer insights across business lines. In practice this powers:

- underwriting triage (match new submissions to historical claims experience),
- customer segmentation (group households by benefit needs),
- scenario analogues (locate past macro paths closest to today’s conditions),
- model monitoring (compare new feature vectors to the training set).

This chapter walks through common similarity metrics for structured and unstructured data, shows their Julia implementations, and closes with a k-nearest-neighbor (kNN) search that you can plug into actuarial workflows.

26.2. The Data

Actuarial datasets tend to mix **structured** (tabular) and **unstructured** (text, image, voice) inputs. Regardless of origin, similarity measures require numeric vectors, so we usually:

- scale numeric fields (premium, age, balances) to comparable ranges,
- encode categorical features (education, occupation, territory) with one-hot encoding or embedding schemes,
- convert unstructured signals to vectors via domain encoders (TF-IDF, Word2Vec, CNNs, spectrograms).

Stored data can generally be categorized into two formats: tabular (structured) and non-tabular (unstructured). Structured data format is a structured way of organizing and presenting data in rows and columns, resembling a table. This format is widely used for storing and representing structured datasets, making it easy to read, analyze, and manipulate data. The most common example of structured data is a spreadsheet, where data is organized into rows and columns. Structured data can also be stored in relational databases for easier lookups and matching. On the other hand, unstructured data refers to data that lacks a predefined data model or structure. Unlike structured data, which fits neatly into tables or databases, unstructured data does not have a predefined schema. It can include text documents, images, audio files, video files, social media posts, and more.

Structured data can be further categorized into numerical and categorical data based on the types of values they represent. The following data tables will be referenced throughout the chapter.

26. Similarity Analysis

Real numerical data can easily be converted or normalized to a series of floating points, and real categorical data to a series of binary literals through one-hot encoding procedures.

For unstructured data, due to the nature of their variety, the choice of representation depends on the type of data and the specific task at hand. For text data, a Word2Vec embedding is commonly used, while Convolutional Neural Networks (CNNs) are for image data and wave transforms are for audio data. No matter which transformation is applied, unstructured data can generally be converted to a series of floating points, just like numerical structured data.

```
sample_csv_data =
    IOBuffer(
        raw"id,sex,benefit_base,education,occupation,issue_age
1,M,100000.0,college,1,30.0
2,F,200000.0,master,3,20.0
3,M,150000.0,high_school,4,40.0
4,F,50000.0,college,2,60.0
5,M,250000.0,college,1,40.0
6,F,200000.0,high_school,2,30.0";
)
```

```
using CSV, DataFrames, TableTransforms

df = CSV.read(sample_csv_data, DataFrame)
df_num = apply(MinMax()), df[:, [:benefit_base, :issue_age]][1]
```

```
Precompiling packages...
  1013.0 ms  ✓ QuartoNotebookWorkerJSONExt (serial)
1 dependency successfully precompiled in 1 seconds
```

| | benefit_base | issue_age |
|---|--------------|-----------|
| | Float64 | Float64 |
| 1 | 0.25 | 0.25 |
| 2 | 0.75 | 0.0 |
| 3 | 0.5 | 0.5 |
| 4 | 0.0 | 1.0 |
| 5 | 1.0 | 0.5 |
| 6 | 0.75 | 0.25 |

```
using StatsBase

arr_cat = hcat(
    indicatormat(df.sex),
    indicatormat(df.education),
    indicatormat(df.occupation)
)
```

```
6x9 Matrix{Bool}:
0 1 1 0 0 1 0 0 0
1 0 0 0 1 0 0 1 0
0 1 0 1 0 0 0 0 1
1 0 1 0 0 0 1 0 0
0 1 1 0 0 1 0 0 0
1 0 0 1 0 0 1 0 0
```

`df_num` contains scaled benefit bases and issue ages, while `arr_cat` stores the one-hot encoding for sex, education, and occupation. Drop or regularize any column with zero range before scaling to avoid division-by-zero warnings. For unstructured inputs, substitute your favorite embedding model (Word2Vec for text, CNN features for images, spectrograms for audio); the downstream similarity routines stay the same once you have numeric vectors.

26.3. Common Similarity Measures

The following measures are commonly used to calculate similarities.

26.3.1. Euclidean Distance (L2 norm)

Euclidean distance, also known as the L2 norm, is defined as

$$d = \sqrt{\sum_{i=1}^n (w_i - v_i)^2}$$

The distance is usually meaningful when applied to numerical data. The following Julia code shows the Euclidean distance for the first two columns in `df_num`.

```
using LinearAlgebra

#d12 = sqrt(sum((df_num[:, 1] - df_num[:, 2]) .^ 2))
#d12 = norm(df_num[:, 1] - df_num[:, 2])
d12 = LinearAlgebra.norm(Array(df_num[:, 1]) - Array(df_num[:, 2]))
```

1.4361406616345072

26.3.2. Manhattan Distance (L1 Norm)

Manhattan distance, also known as the L1 norm or taxicab distance, is defined as

$$d = \sum_{i=1}^n |w_i - v_i|$$

The name comes from the grid-like street layout of Manhattan: instead of measuring the straight-line path between two points, you sum the absolute differences along each axis, as if navigating city blocks.

Manhattan distance is well suited for numerical data, particularly when features are measured on different scales or when you want to reduce the influence of outliers. Unlike Euclidean distance, which squares differences (amplifying large deviations), Manhattan distance treats all deviations linearly. This makes it more robust when a single feature has an unusually large value. In actuarial contexts, Manhattan distance can be useful for comparing policies where one dimension (e.g., benefit base) might have extreme values that would otherwise dominate a Euclidean measure.

```
using LinearAlgebra

d12 = norm1(Array(df_num[:, 1]) - Array(df_num[:, 2]))
```

26. Similarity Analysis

26.3.3. Cosine Similarity

Cosine similarity is defined as

$$d = \frac{\sum_{i=1}^n w_i \cdot v_i}{\sqrt{\sum_{i=1}^n w_i^2} \cdot \sqrt{\sum_{i=1}^n v_i^2}}$$

The similarity is meaningful when applied to both numerical and categorical data.

The following Julia code shows the cosine similarity for the first two columns in df_num.

```
using LinearAlgebra

d1_2 = (Array(df_num[:, 1]) .* Array(df_num[:, 2])) /
       norm(df_num[:, 1]) / norm(df_num[:, 2])

0.5024594344170622
```

The following Julia code shows the cosine similarity for the first and the third rows in arr_cat.

```
using LinearAlgebra

d1_3 = (arr_cat[1, :] .* arr_cat[3, :]) / norm(arr_cat[1, :]) / norm(arr_cat[3,
                           :, :])

0.3333333333333337
```

Note how similar the syntax of processing for numerical or categorical data is. Multiple dispatch allows Julia to identify the most efficient underlying procedure for different types of data. For categorical data, the dot operation on binary vectors essentially counts the number of shared 1's (or the number of categories on which two observations match), while for numerical data it is the dot operation for most numerical processing libraries.

26.3.4. Jaccard Similarity

Jaccard similarity is defined as

$$d = \frac{|W \cap V|}{|W \cup V|}$$

The similarity is usually meaningful when applied to categorical data. The following Julia code shows the Jaccard similarity for the first and the third rows in arr_cat.

```
d1_3 = (arr_cat[1, :] .* arr_cat[3, :]) / sum(arr_cat[1, :] .| arr_cat[3, :])

0.2
```

26.3.5. Hamming Distance

Hamming distance is defined as $d = \text{Number of positions at which } w \text{ and } v \text{ differ.}$

$$d = \sum_{i=1}^n \mathbf{1}_{w_i \neq v_i}$$

The similarity is usually meaningful when applied to categorical data. The following Julia code shows the Hamming distance for the first and the third rows in arr_cat.

```
d1_3 = sum(arr_cat[1, :] .v arr_cat[3, :])  
4
```

26.3.6. Choosing a Metric

| Data type | Typical metric | Notes |
|--|------------------------|---|
| Continuous (scaled) | Euclidean, Mahalanobis | Euclidean assumes uncorrelated features; Mahalanobis accounts for covariance. |
| Sparse non-negative (counts, embeddings) | Cosine, Jaccard | Cosine stays stable when magnitude varies; Jaccard focuses on overlap. |
| Binary/categorical | Hamming, Jaccard | Hamming counts mismatches; Jaccard ignores joint zeros. |

Pick the measure that best reflects how “similarity” should behave for your business question (e.g., two borrowers sharing underwriting flags may be “close” even if balances differ).

26.4. **k**-Nearest Neighbor (*k*NN) Clustering

k-Nearest Neighbor (*k*NN) is primarily known as a classification algorithm, but it can also be used for clustering, particularly in the context of density-based clustering. Density-based clustering identifies regions in the data space where the density of data points is higher, and it groups points in these high-density regions. The core idea of *k*NN clustering is to assign each data point to a cluster based on the density of its neighbors. A data point becomes a core point if it has at least a specified number of neighbors within a certain distance.

```
using Random, NearestNeighbors, CairoMakie  
  
# Generate synthetic data  
Random.seed!(1234)  
data = rand(2, 10) # 10 points with 2 dimensions  
println("Dataset:\n", data)  
  
# Create a KD-tree for efficient nearest neighbor search  
kdtree = KDTree(data)  
  
# Define a query point (for which we want to find nearest neighbors)  
query_point = [0.5, 0.5]  
  
# Specify how many neighbors to find  
k = 3  
indices, distances = knn(kdtree, query_point, k)  
  
# Display the results  
println("\nQuery Point: ", query_point)  
println("Indices of Nearest Neighbors: ", indices)  
println("Distances to Nearest Neighbors: ", distances)
```

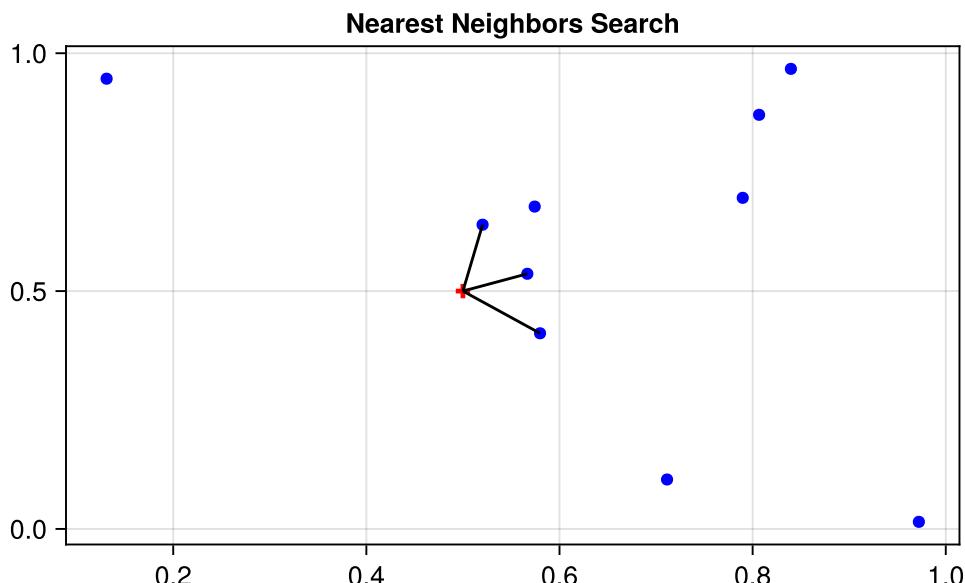
26. Similarity Analysis

```
# Visualize the points and the query
f = Figure()
axis = Axis(f[1, 1], title="Nearest Neighbors Search")
scatter!(data[1, :], data[2, :], label="Data Points", color=:blue)
scatter!([query_point[1]], [query_point[2]], label="Query Point",
        color=:red, marker=:cross, markersize=10)

# Highlight nearest neighbors
for idx in indices
    lines!([query_point[1], data[1, idx]], [query_point[2], data[2, idx]],
           ↵ color=:black)
end
f

Precompiling packages...
  3749.4 ms  ✓ QuartoNotebookWorkerMakieExt (serial)
1 dependency successfully precompiled in 4 seconds
Precompiling packages...
  3589.7 ms  ✓ QuartoNotebookWorkerCairoMakieExt (serial)
1 dependency successfully precompiled in 4 seconds
Dataset:
[0.5798621201341324 0.9721360824554687 ... 0.13102565622085904 0.5743234852783174;
 ↵ 0.4112941179498505 0.014908849285099945 ... 0.9464532262313834
 ↵ 0.6776499075995779]

Query Point: [0.5, 0.5]
Indices of Nearest Neighbors: [3, 6, 1]
Distances to Nearest Neighbors: [0.14103817169408245, 0.07597457975710152,
 ↵ 0.11935950629343954]
```



indices point to the nearest policies and distances report the Euclidean distance in the scaled space. Replace the numeric embedding with any other feature matrix (cosine-normalized TF-IDF

26.4. *k*-Nearest Neighbor (*kNN*) Clustering

vectors, CNN image embeddings) to reuse the same workflow, or feed the neighbor set into downstream analytics (experience weighting, peer benchmarking). If the query point falls outside the observed min–max range, clip or rescale to keep the distances interpretable.

26. Similarity Analysis

27. Portfolio Optimization

CHAPTER AUTHORED BY: YUN-TIEN LEE

“Portfolio optimization is not about finding the perfect mix, but about managing risk and return in a way that aligns with your goals and circumstances.” — Unknown

27.1. Chapter Overview

This chapter introduces optimization in a portfolio context, illustrating how mathematical and computational techniques can be applied to the classic problem of allocating capital among different assets. We begin with the foundations of mean-variance optimization, describing how investors balance expected return and risk, and how these trade-offs can be expressed as a constrained optimization problem. The chapter then expands to cover a variety of real-world extensions and constraints, including risk-based capital, prior-posterior views and Sharpe ratios.

27.2. The Data

We will use a toy three-asset portfolio throughout the chapter. The vector μ contains annualized expected returns and Σ stores the covariance matrix implied by historical or modeled volatilities. Keeping the dataset intentionally small lets us focus on the optimization mechanics before scaling up to tens or hundreds of assets in practice.

```
μ = [0.1, 0.15, 0.12] # expected returns
Σ = [0.1 0.05 0.03;
      0.05 0.12 0.04;
      0.03 0.04 0.08] # covariances
n_assets = length(μ) # number of assets
```

3

27.3. Theory

Harry Markowitz introduced modern portfolio theory in 1952. The main idea is that investors seek to maximize their expected return of a portfolio given a certain amount of risk. Under standard assumptions, achieving higher expected returns typically requires accepting higher risk; portfolios that simultaneously deliver higher return and lower risk dominate others and define the upper boundary of the feasible set. The efficient frontier traces the best achievable tradeoff.

27.4. Mathematical tools

27.4.1. Mean-variance optimization model

Mean-variance optimization is a mathematical framework that seeks to maximize expected returns while minimizing portfolio variance (or standard deviation). It involves calculating the expected return and risk of individual assets and finding the optimal combination of assets to achieve the desired risk-return tradeoff.

$$\begin{aligned} & \text{minimize} && w^T \Sigma w \\ & \text{subject to} && r^T w \geq \mu_{\text{target}} \\ & && 1^T w = 1 \\ & && w \geq 0 \end{aligned}$$

```
using JuMP, Ipopt, LinearAlgebra

# Create an optimization model
model = Model(Ipopt.Optimizer)
set_silent(model)
# Set up weights as variables to optimize
@variable(model, w[1:n_assets] >= 0.0)
# Objective: minimize portfolio variance
@objective(model, Min, dot(w, Σ * w))
# Constraints: Sum of portfolio weights should equal to 1,
# and all weights should be zero or positive
@constraint(model, sum(w) == 1.0)
# May also add additional constraints
# target_return = 0.1
# @constraint(model, dot(μ, w) >= target_return)
# Solve the optimization problem
optimize!(model)
w_meanvar = value.(w)
μ_meanvar = dot(μ, w_meanvar)
σ_meanvar = sqrt(dot(w_meanvar, Σ * w_meanvar))

# Print results
println("Optimal minimum mean-variance portfolio weights:")
for (i, wi) in enumerate(w_meanvar)
    println("Asset ", i, ":", round(wi, digits=4))
end
println("Portfolio mean optimized under mean-variance = ", round(μ_meanvar,
    digits=4))
println("Portfolio standard deviation optimized under mean-variance = ",
    round(σ_meanvar, digits=4))
```

This program contains Ipopt, a library for large-scale nonlinear optimization.
 Ipopt is released as open source code under the Eclipse Public License (EPL).
 For more information visit <https://github.com/coin-or/Ipopt>

```

Optimal minimum mean-variance portfolio weights:
Asset 1: 0.3333
Asset 2: 0.1667
Asset 3: 0.5
Portfolio mean optimized under mean-variance = 0.1183
Portfolio standard deviation optimized under mean-variance = 0.238

```

In real-world settings you may impose lower/upper bounds, forbid short positions, or layer liquidity constraints, but the structure is identical. Once the baseline model is producing sensible weights, iterate on additional business rules.

In mean-variance portfolio optimization, incorporating a cost of risk-based capital on assets is a practical consideration that reflects the additional capital required to support riskier assets in a portfolio. This approach ensures that the optimization process not only maximizes returns relative to risk but also considers the regulatory or internal cost implications associated with holding riskier assets.

$$\begin{aligned}
& \text{maximize} && w^T R_{adj} \\
& \text{subject to} && w^T \Sigma w \leq \sigma_{max}^2 \\
& && 1^T w = 1 \\
& && w \geq 0
\end{aligned}$$

where $R_{adj} = [(\mu_1 - \lambda_1), (\mu_2 - \lambda_2), \dots, (\mu_N - \lambda_N)]$ are the risk-adjusted expected returns.

```

using JuMP, Ipopt, LinearAlgebra

# Create an optimization model
model = Model(Ipopt.Optimizer)
set_silent(model)
λ = [0.01, 0.02, 0.05] # RBC cost per asset
r = μ .- λ # risk adjusted returns
σ²_max = 0.1 # maximum portfolio variance
# Set up weights as variables to optimize
@variable(model, w[1:n_assets] >= 0.0)
# Objective: maximize risk-adjusted return
@objective(model, Max, dot(r, w))
# Constraints: Sum of portfolio weights should equal to 1,
# and all weights should be zero or positive
@constraint(model, sum(w) == 1.0)
# Constraints: Sum of allowable portfolio variance is limited
@constraint(model, dot(w, Σ * w) <= σ²_max)
# May also add additional constraints
# target_return = 0.1
# @constraint(model, dot(μ, w) >= target_return)
# Solve the optimization problem
optimize!(model)
w_rbc = value.(w)
μ_rbc = dot(μ, w_rbc)
σ_rbc = sqrt(dot(w_rbc, Σ * w_rbc))

# Print results
println("Portfolio weights with RBC-adjusted returns:")
for (i, wi) in enumerate(w_rbc)
    println("Asset ", i, ":", round(wi, digits=4))

```

27. Portfolio Optimization

```
end
println("Portfolio mean optimized under RBC = ", round(μ_rbc, digits=4))
println("Portfolio standard deviation optimized under RBC = ", round(σ_rbc,
    ↴ digits=4))

Portfolio weights with RBC-adjusted returns:
Asset 1: 0.1667
Asset 2: 0.8333
Asset 3: 0.0
Portfolio mean optimized under RBC = 0.1417
Portfolio standard deviation optimized under RBC = 0.3162
```

Capital charges effectively haircut the expected return of risky assets. In this simple illustration the optimizer shifts weight toward the lower-cost asset once the variance constraint is binding. In a production setting you would source the adjustments from your solvency framework (for example, regulatory RBC factors or internal economic capital) and consider separate limits for issuer, sector, or region exposures.

27.4.2. Efficient frontier analysis

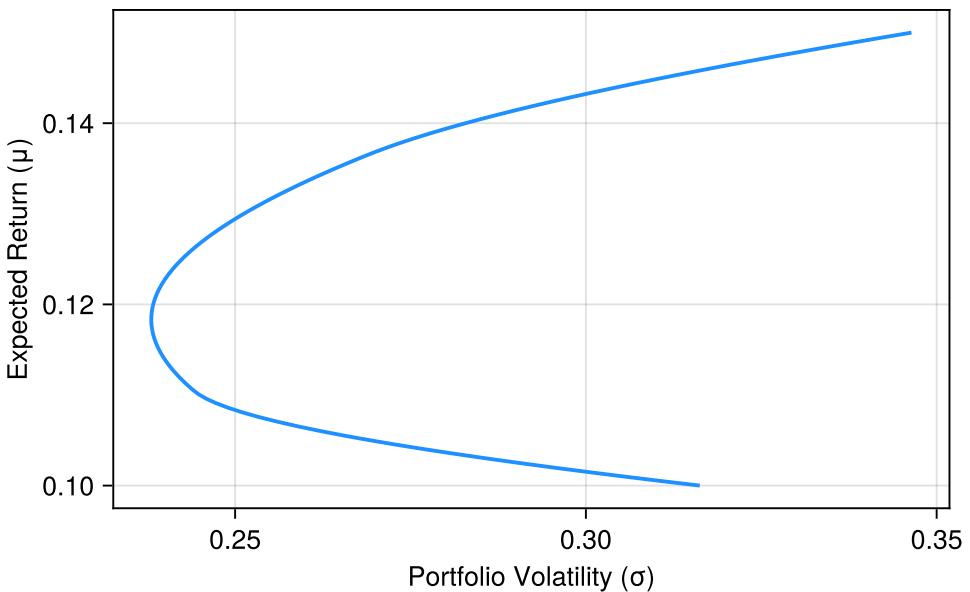
The efficient frontier represents the set of portfolios that offer the highest expected return for a given level of risk or the lowest risk for a given level of return. Efficient frontier analysis involves plotting risk-return combinations for different portfolios and identifying the optimal portfolio on the frontier.

```
using JuMP, Ipopt, CairoMakie, LinearAlgebra
import MathOptInterface as MOI

function efficient_frontier(μ, Σ; points::Int=100)
    targets = range(minimum(μ), maximum(μ), length=points)
    frontier = Tuple{Float64,Float64}[]
    for target in targets
        model = Model(Ipopt.Optimizer)
        set_silent(model)
        n = length(μ)
        @variable(model, w[1:n] >= 0.0)
        @objective(model, Min, dot(w, Σ * w))
        @constraint(model, sum(w) == 1.0)
        @constraint(model, dot(μ, w) == target)
        optimize!(model)
        status = termination_status(model)
        if status in (MOI.LOCALLY_SOLVED, MOI.OPTIMAL)
            push!(frontier, (sqrt(objective_value(model)), target))
        end
    end
    frontier
end

ef = efficient_frontier(μ, Σ)
# Plot Efficient Frontier
fig = Figure()
Axis(fig[1, 1], xlabel="Portfolio Volatility (σ)", ylabel="Expected Return
    ↴ (μ)")
lines!(first.(ef), last.(ef), color=:dodgerblue, linewidth=2)
```

fig



Tracing the efficient frontier is useful for communicating trade-offs to stakeholders. Each point corresponds to a distinct target return; the `efficient_frontier` helper simply sweeps a grid of targets and resolves the mean-variance program. In client reporting this visual is usually overlaid with realized portfolios, policy benchmarks, and stress scenarios.

27.4.3. Black-Litterman

The Black-Litterman model combines the views of investors with market equilibrium assumptions to generate optimal portfolios. It starts with a market equilibrium portfolio and adjusts it based on investor views and confidence levels. The model incorporates subjective opinions while maintaining diversification and risk management principles.

$$\begin{aligned}
 & \text{maximize} \quad \mu_{BL}^T w - \frac{\lambda}{2} w^T \Sigma w \\
 & \text{subject to} \quad \sum_{i=1}^N w_i = 1 \\
 & \quad w_i \geq 0, \quad \forall i \\
 & \text{where } \mu_{BL} = \\
 & \quad \left((\tau \Sigma)^{-1} + P^T \Omega^{-1} P \right)^{-1} \left((\tau \Sigma)^{-1} \mu + P^T \Omega^{-1} Q \right)
 \end{aligned}$$

μ = prior (implied) returns from the market equilibrium. P = matrix encoding which assets your views are on. Q = your views (expected returns for specific portfolios). Ω = uncertainty (covariance) of your views. τ = scaling factor for the prior covariance (usually small, like 0.025).

27. Portfolio Optimization

```

using JuMP, Ipopt, LinearAlgebra

# Market equilibrium parameters (prior)
λ = 2.5 # risk aversion
μ_market = fill(0.08, n_assets) # Market equilibrium return
Σ_prior = Σ # Market equilibrium covariance matrix
# Investor views
Q = μ # Expected returns on assets according to investor views
P = Matrix(I, n_assets, n_assets) # Pick matrix specifying which assets views
# are on
Ω = Diagonal([1e-6, 4e-6, 9e-6]) # Views uncertainty (covariance matrix)
τ = 0.05 # Scaling factor

# Black-Litterman expected return adjustment
Σ_inv = inv(Σ_prior)
# Calculate the posterior expected returns
A = Σ_inv / τ + P' * (Ω \ P)
B = Σ_inv * μ_market / τ + P' * (Ω \ Q)
μ_BL = A \ B

# Create an optimization model
model = Model(Ipopt.Optimizer)
set_silent(model)
# Set up weights as variables to optimize
@variable(model, w[1:n_assets] >= 0.0)
# Objective: maximize sharpe ratio
@objective(model, Max, dot(μ_BL, w) - (λ / 2) * dot(w, Σ_prior * w))
# Constraints: Sum of portfolio weights should equal to 1,
# and all weights should be zero or positive
@constraint(model, sum(w) == 1.0)
# Solve the optimization problem
optimize!(model)
w_BL = value.(w)
μ_BLpost = dot(μ_BL, w_BL)
σ_BLpost = sqrt(dot(w_BL, Σ_prior * w_BL))
# Print results
println("Portfolio Weights optimized under Black-Litterman:")
for (i, wi) in enumerate(w_BL)
    println("Asset ", i, ": ", round(wi, digits=4))
end
println("Portfolio mean optimized under Black-Litterman = ",
    round(μ_BLpost, digits=4))
println("Portfolio standard deviation optimized under Black-Litterman = ",
    round(σ_BLpost, digits=4))

```

Portfolio Weights optimized under Black-Litterman:

Asset 1: 0.178

Asset 2: 0.3444

Asset 3: 0.4776

Portfolio mean optimized under Black-Litterman = 0.1267

Portfolio standard deviation optimized under Black-Litterman = 0.245

Black-Litterman provides a disciplined way to blend equilibrium returns (for instance, those implied by a global cap-weighted index) with discretionary views. The prior acts as an anchor; the pick matrix P , the view vector Q , and the confidence matrix Ω tilt the portfolio toward convictions.

In practice you would calibrate τ and the Ω diagonal entries so that strong views meaningfully shift allocations while weak views barely move the dial.

27.4.4. Risk Parity

Risk parity is an asset allocation strategy that allocates capital based on risk rather than traditional measures such as market capitalization or asset prices. It aims to balance risk contributions across different assets or asset classes to achieve a more stable portfolio. Risk parity portfolios often include assets with different risk profiles, such as stocks, bonds, and commodities.

$$\begin{aligned} \min_{w \in \mathbb{R}^N} \quad & \sum_{i=1}^N (RC_i - t)^2 \\ \text{subject to} \quad & \sum_{i=1}^N w_i = 1, \\ & w_i \geq 0, \quad i = 1, \dots, N \\ \text{where} \quad & RC_i = \frac{w_i(\Sigma w)_i}{w^T \Sigma w + \epsilon} \end{aligned}$$

```
using JuMP, Ipopt, LinearAlgebra

# Create an optimization model
model = Model(Ipopt.Optimizer)
set_silent(model)
# Set up weights as variables to optimize
@variable(model, w[1:n_assets] >= 0.0)
# Objective: equalize risk contributions across assets
@expression(model, Σw[i=1:n_assets], sum(Σ[i, j] * w[j] for j in 1:n_assets))
@expression(model, var_p, sum(w[i] * Σw[i] for i in 1:n_assets))
@expression(model, RC[i=1:n_assets], (w[i] * Σw[i]) / (var_p + eps()))
@expression(model, target_contribution, var_p / n_assets)
@objective(model, Min, sum((RC[i] - target_contribution)^2 for i in
    ↳ 1:n_assets))
# Constraints: Sum of portfolio weights should equal to 1,
# and all weights should be zero or positive
@constraint(model, sum(w) == 1.0)
# Solve the optimization problem
optimize!(model)
w_rp = value.(w)
Σw_val = Σ * w_rp
rc = w_rp .* Σw_val
rc_share = rc ./ sum(rc)
# Print results
println("Optimal portfolio weights under risk parity:")
for (i, wi) in enumerate(w_rp)
    println("Asset ", i, ": ", round(wi, digits=4))
    println("RC share ", i, ": ", round(rc_share[i], digits=4))
end

Optimal portfolio weights under risk parity:
Asset 1: 0.3309
RC share 1: 0.3334
```

27. Portfolio Optimization

```
Asset 2: 0.2951
RC share 2: 0.3369
Asset 3: 0.3741
RC share 3: 0.3297
```

Each asset contributes one third of total portfolio variance at the optimum, a hallmark of risk parity construction. Compared with the minimum-variance allocation, the solution spreads weight more evenly because it penalizes large disparities in marginal risk contributions.

Tip

The helper `eps()` in the formulation avoids dividing by zero when the portfolio variance is extremely small. In production code you would replace this with an explicit numerical tolerance.

27.4.5. Sharpe Ratio Maximization

The Sharpe ratio measures the risk-adjusted return of a portfolio and is calculated as the ratio of excess return to volatility. Maximizing the Sharpe ratio involves finding the portfolio allocation that offers the highest risk-adjusted return. This approach focuses on achieving the best tradeoff between risk and return.

$$\begin{aligned} & \text{maximize} \quad \frac{E[R_p] - rfr}{\sigma_p} \\ & \text{subject to} \quad \sum_{i=1}^N w_i = 1 \\ & \quad w_i \geq 0, \quad \forall i \end{aligned}$$

```
using JuMP, Ipopt, LinearAlgebra

# Create an optimization model
model = Model(Ipopt.Optimizer)
set_silent(model)
# Set up weights as variables to optimize
@variable(model, w[1:n_assets] >= 0.0)
# Objective: maximize Sharpe ratio
rfr = 0.05 # risk free rate
@objective(
    model,
    Max,
    (dot(μ, w) - rfr) /
    sqrt(sum(w[i] * Σ[i, j] * w[j] for i in 1:n_assets, j in 1:n_assets) +
    ← eps())
)
# Constraints: Sum of portfolio weights should equal to 1,
# and all weights should be zero or positive
@constraint(model, sum(w) == 1.0)
# Solve the optimization problem
optimize!(model)
w_sharpe = value.(w)
μ_sharpe = dot(μ, w_sharpe)
```

```

σ_sharpe = sqrt(dot(w_sharpe, Σ * w_sharpe))
sharpe_ratio = (μ_sharpe - rfr) / σ_sharpe

# Print results
println("Optimal portfolio weights using Sharpe Ratio:")
for (i, wi) in enumerate(w_sharpe)
    println("Asset ", i, ":", round(wi, digits=4))
end
println("Sharpe ratio = ", round(sharpe_ratio, digits=4))
println("Portfolio mean optimized using Sharpe Ratio = ",
       round(μ_sharpe, digits=4))
println("Portfolio standard deviation optimized using Sharpe Ratio = ",
       round(σ_sharpe, digits=4))

Optimal portfolio weights using Sharpe Ratio:
Asset 1: 0.018
Asset 2: 0.5309
Asset 3: 0.451
Sharpe ratio = 0.3217
Portfolio mean optimized using Sharpe Ratio = 0.1356
Portfolio standard deviation optimized using Sharpe Ratio = 0.266

```

Maximizing the Sharpe ratio is equivalent to choosing the tangency portfolio when a risk-free asset is available. The non-linear objective makes the problem slightly harder than the quadratic programs we solved earlier, but Ipopt handles it comfortably. Many practitioners linearize the problem by fixing variance and maximizing excess return, then scanning along the frontier as needed.

27.4.6. Robust Optimization

Robust optimization techniques aim to create portfolios that are resilient to uncertainties and fluctuations in market conditions. These techniques consider a range of possible scenarios and optimize portfolios to perform well across different market environments. A robust parameter in robust portfolio optimization is typically chosen to ensure the portfolio's performance remains stable and satisfactory under different market conditions or variations in input data. Robust optimization may involve incorporating stress tests, scenario analysis, or robust risk measures into the portfolio construction process.

$$\begin{aligned}
& \text{minimize} && w^T \Sigma w + \gamma \|w - w_0\|_2^2 \\
& \text{subject to} && \sum_{i=1}^N w_i = 1 \\
& && w_i \geq 0, \quad \forall i \\
& && \|(\Sigma^{1/2}(w - w_0))\|_2 \leq \epsilon
\end{aligned}$$

```

using JuMP, Ipopt, LinearAlgebra

# Create an optimization model
model = Model(Ipopt.Optimizer)
set_silent(model)
# Set up weights as variables to optimize
@variable(model, w[1:n_assets] >= 0.0)

```

27. Portfolio Optimization

```

# Objective: minimize variance with robustness penalty
ε = 0.05 # Uncertainty level
γ = 0.1 # Robustness parameter
w₀ = [0.3, 0.4, 0.3] # expected weights
@objective(model, Min, dot(w, Σ * w) + γ * sum((w[i] - w₀[i])^2 for i in
    ↵ 1:n_assets))
# Constraints: Sum of portfolio weights should equal to 1,
# and all weights should be zero or positive
@constraint(model, sum(w) == 1)
@constraint(model,
    sum(((w[i] - w₀[i]) * Σ[i, j] * (w[j] - w₀[j]))) for i in 1:n_assets, j in
    ↵ 1:n_assets) <= ε)
# Solve the optimization problem
optimize!(model)
w_robust = value.(w)
μ_robust = dot(μ, w_robust)
σ_robust = sqrt(dot(w_robust, Σ * w_robust))
# Print results
println("Portfolio Weights under robust optimization:")
for (i, wi) in enumerate(w_robust)
    println("Asset ", i, ": ", round(wi, digits=4))
end
println("Portfolio mean under robust optimization = ",
    round(μ_robust, digits=4))
println("Portfolio standard deviation under robust optimization = ",
    round(σ_robust, digits=4))

```

Portfolio Weights under robust optimization:

Asset 1: 0.3125

Asset 2: 0.3125

Asset 3: 0.375

Portfolio mean under robust optimization = 0.1231

Portfolio standard deviation under robust optimization = 0.2427

Robust optimization frameworks vary from simple shrinkage toward a reference portfolio (as shown here) to full-fledged worst-case analysis across an uncertainty set. They are especially helpful when market regimes shift quickly or when return estimates are notoriously noisy, as is common with alternative assets and private markets.

27.4.7. Asset weights from different methodologies

Table 27.1.: Optimized asset weights from different methodologies

| Methodology | Asset weights |
|--|--|
| Standard mean variance
(with RBC costs) | [0.33, 0.17, 0.50]
[0.17, 0.83, 0.00] |
| Black-Litterman | [0.18, 0.34, 0.48] |
| Risk parity | [0.33, 0.30, 0.37] |
| Sharpe ratio | [0.02, 0.53, 0.45] |
| Robust | [0.31, 0.31, 0.38] |

Comparing the allocations highlights how each methodology encodes a different business question. The RBC-adjusted run rotates capital toward the asset with the lowest capital charge even though it is not the highest-return instrument. The Sharpe maximizer hugs the asset with the best risk-adjusted profile, while risk parity and robust optimization deliberately keep weights closer to equal to avoid concentration risk.

27.5. Key Takeaways

- Always start with clean inputs: align expected returns, covariances, and capital charges to the same horizon and currency before optimizing.
- Quadratic programs (mean-variance, risk parity, Black-Litterman) are convex and deterministic; use them to benchmark more exotic heuristics.
- Adjust expected returns for regulatory or internal capital costs when comparing portfolios with very different risk profiles.
- Non-linear objectives (Sharpe, robust formulations) often require interior-point solvers; provide good initial guesses and check convergence diagnostics.
- Communicate results via efficient frontier plots and side-by-side weight tables so stakeholders understand why recommendations differ.

27.6. Practical considerations

27.6.1. Fractional purchases of assets

In traditional portfolio optimization, fractional purchases of assets refer to the ability to allocate fractions or percentages of capital to individual assets. However, in certain contexts or practical implementations, fractional purchases may not be allowed or considered.

- Practical constraints. Some investment vehicles or platforms may restrict investors from purchasing fractions of shares or assets. For instance, certain mutual funds, exchange-traded funds (ETFs), or other investment products may require whole units of shares to be purchased.
- Simplicity and cost-effectiveness. Handling fractional shares can add complexity and operational costs to portfolio management, especially in terms of transaction fees, administrative overhead, and reconciliation processes.
- Market liquidity. Some assets may have limited liquidity or trading volumes, making it impractical or difficult to execute fractional purchases without significantly impacting market prices or transaction costs.
- Regulatory considerations. Regulations in certain jurisdictions may impose restrictions on fractional share trading or ownership, potentially limiting the ability to include fractional purchases in portfolio optimization strategies.

27.6.2. Large number of assets

In portfolio optimization, a penalty factor for a large volume of assets typically refers to a mechanism or adjustment applied to the optimization process to mitigate the potential biases or challenges that arise when dealing with a large number of assets. This concept is particularly relevant in the context of mean-variance optimization and other optimization frameworks where computational efficiency and practical portfolio management considerations come into play. Too many assets may have the following issues.

27. Portfolio Optimization

- Dimensionality. As the number of assets (or dimensions) increases in a portfolio, traditional optimization methods may become computationally intensive or prone to overfitting. This is because the complexity of the optimization problem grows exponentially with the number of assets.
- Sparsity and concentration. In practice, not all assets may contribute equally to portfolio performance. Some assets may have negligible impact on the overall portfolio characteristics (such as risk or return) due to low weights or correlations with other assets.
- Penalizing excessive complexity. A penalty factor can be introduced to penalize portfolios that overly diversify or allocate small weights to a large number of assets. This encourages the optimization process to focus on more significant assets or reduce the complexity of the portfolio structure.

There are various ways to implement a penalty factor for a large volume of assets:

- Regularization techniques. Techniques like Lasso (L1 regularization) or Ridge (L2 regularization) regression can penalize small weights or excessive diversification by adding a penalty term to the objective function.
- Subset selection. Methods that explicitly select a subset of assets based on their contribution to portfolio performance, rather than including all assets indiscriminately.
- Heuristic adjustments. Introducing heuristic rules or adjustments based on practical portfolio management principles or empirical observations.

28. Sensitivity Analysis

CHAPTER AUTHORED BY: YUN-TIEN LEE

"Sensitivity analysis is the art of understanding how small changes in assumptions can lead to big changes in outcomes, helping us navigate uncertainty with clarity." — Unknown

28.1. Chapter Overview

Sensitivity analysis tells us how fragile our valuation or risk metrics are once key drivers shift. We will move from **local techniques** (derivatives, finite differences, automatic differentiation) that explain the effect of tiny shocks to **global techniques** (regression-based screening, Sobol, Morris, FAST) that summarize how entire distributions of inputs ripple through a model. The chapter closes with practical scenario-analysis guidance so you can turn the math into board-ready stress narratives.

28.2. Setup

To keep the examples concrete we will work with a simplified block of participating whole life policies.

```
using Dates

@enum Sex Female = 1 Male = 2
@enum Risk Standard = 1 Preferred = 2

mutable struct Policy
    id::Int
    sex::Sex
    benefit_base::Float64
    pir::Float64 # the policy interest rate
    mode::Int
    prem::Float64
    pp::Int
    issue_date::Date
    issue_age::Int
    risk::Risk
end
```

The fields give us everything we need to project reserves: demographic attributes, benefit base, the policy interest rate (`pir`), premium pattern, and risk class. Real models would store many more underwriting features, but this toy structure is enough to showcase the techniques.

28.3. The Data

```

using MortalityTables
sample_csv_data =
    IOBuffer(
        raw"id,sex,benefit_base,pir,mode,prem,pp,issue_date,issue_age,risk
        1,M,100000.0,0.03,1,1000.0,10,1999-12-05,30,Std"
    )

mort = Dict(
    # assuming no issue ages < 2 or > 120
    Male => MortalityTables.table(262).ultimate,
    Female => MortalityTables.table(261).ultimate,
)

```

Dict{Sex, OffsetArrays.OffsetVector{Float64, Vector{Float64}}} with 2 entries:

```

Male   => [0.00051, 0.00038, 0.00035, 0.00032, 0.0003, 0.00027, 0.00025, 0.00...
Female => [0.00045, 0.00031, 0.00025, 0.00022, 0.0002, 0.00019, 0.00019, 0.00...

```

```

using CSV, DataFrames

policies = let

    # read CSV directly into a dataframe
    # df = CSV.read("sample_inforce.csv",DataFrame) # use local string for
    #       ↵ notebook
    df = CSV.read(sample_csv_data, DataFrame)

    # map over each row and construct an array of Policy objects
    map(eachrow(df)) do row
        Policy(
            row.id,
            row.sex == "M" ? Male : Female,
            row.benefit_base,
            row.pir,
            row.mode,
            row.prem,
            row.pp,
            row.issue_date,
            row.issue_age,
            row.risk == "Std" ? Standard : Preferred,
        )
    end
end

1-element Vector{Policy}:
Policy(1, Male, 100000.0, 0.03, 1, 1000.0, 10, Date("1999-12-05"), 30, Standard)

```

For example, `res(2, policies[1])` returns the reserve at the end of policy year 2 for our sample contract. We will refer to `policies[1]` when we need a concrete contract for the sensitivity examples.

Given a basic insurance product, a pure whole of life (WOL) policy with level benefits and level premiums payable within the first 10 years, the reserve in-force at the end of the y^{th} ($y \geq 0$) policy year is defined by

$$res(y) = \sum_{t=age+y}^{120} (sur_{t-age-y} * mort_t * B_y / \sqrt{1 + pir}) - (P_y * sur_{t-age-y})$$

where

- $mort_t$ is the mortality at age t
- sur_y is the survival probability adjusted with the policy interest rate, with values of
 - $sur_0 = 1$,
 - $sur_x = sur_{x-1} * (1 - mort_{age+x-1}) / (1 + pir)$ for $x \geq y$, and
 - 0 for $x < y - 1$ or $age + x \geq 120$, or ultimate age of the current mortality table
- B_y is the level benefit throughout the policy
- P_y is the level premium within the first 10 policy years which is 0 for policy years after 10
- pir is the level policy interest rate throughout the policy

```
# Can be made more efficient by storing values in
# a pre-calculated table.
# The recursive calculation here is for illustrative purposes.
function sur(y:Int, pol:Policy; Q=120)
  if y == 0
    1.0
  elseif y < 0 || Q - y <= pol.issue_age
    0.0
  else
    sur(y - 1, pol) * (1 - mort[pol.sex][pol.issue_age+y-1]) / (1 +
      → pol.pir)
  end
end

function res(y:Int, pol:Policy; Q=120)
  s = 0.0
  if y >= 1 && y <= Q - pol.issue_age
    for t in (pol.issue_age+y):Q
      prem = 0.0
      if y <= pol.pp
        prem = pol.prem
      end
      s += sur(t - pol.issue_age, pol) * mort[pol.sex][t] *
        → pol.benefit_base /
          sqrt(1 + pol.pir) - prem * sur(t - pol.issue_age, pol)
    end
  end
  s
end

res (generic function with 1 method)
```

28.4. Common Sensitivity Analysis Methodologies

28.4.1. Finite Differences

Central finite differences approximate derivatives by shocking one input up and down while holding everything else fixed. The result can be interpreted as a “delta” (absolute change per unit

28. Sensitivity Analysis

of input) or as an elasticity if we scale by the base reserve.

```
function res_wrt_r_fd(y::Int, pol::Policy, r::Float64, h=1e-3)
    p+, p- = deepcopy(pol), deepcopy(pol)
    p+.pir, p-.pir = r + h, r - h
    (res(y, p-) - res(y, p+)) / (2 * h * res(y, pol))
end

# percentage changes in reserve at year 2 when
# the interest rate at 3% with a perturbation of 0.1%
res_wrt_r_fd(2, policies[1], 0.03)
```

186.02131593930355

Use a smaller h when pir is quoted in decimals (e.g., 3%) and a larger h if you work in basis points. Always sanity check that the perturbation stays within practical limits (no negative credited rates).

28.4.2. Regression Analyses

Regression-based sensitivity (also called standardized regression coefficients) fits a metamodel between sampled inputs and the resulting reserves, then reports correlations and partial correlations as quick importance scores.

```
using GlobalSensitivity

function r1_wrt_r(r)
    p = deepcopy(policies[1])
    p.pir = r[2]
    p.prem = r[3]
    res(Int(round(r[1])), p)
end

# reserve @ year 1 or 2, interest rate ∈ [0.01, 0.05], prem ∈ [500, 1500]
reg_anal = gsa(
    r1_wrt_r, RegressionGSA(),
    [[1, 2], [0.01, 0.05], [500, 1500]],
    samples=1000)
println(reg_anal.pearson)

[0.03523290349325451 -0.7431891564149411 -0.6186193238786605]
```

The Pearson coefficients show the correlation coefficient matrix between inputs and outputs.

28.4.3. Sobol Indices

Sobol indices are a variance-based method that decomposes the variance of the output of the model or system into fractions which can be attributed to inputs or sets of inputs. This helps to get not just the individual parameter's sensitivities, but also gives a way to quantify the effect and sensitivity from the interaction between the parameters.

$$Y = f_0 + \sum_{i=1}^d f_i(X_i) + \sum_{i < j}^d f_{ij}(X_i, X_j) + \dots + f_{1,2,\dots,d}(X_1, X_2, \dots, X_d)$$

$$Var(Y) = \sum_{i=1}^d V_i + \sum_{i < j}^d V_{ij} + \dots + V_{1,2,\dots,d}$$

The Sobol Indices are “ordered”, the first order indices given by $S_i = \frac{V_i}{Var(Y)}$, the contribution to the output variance of the main effect of X_i . Therefore, it measures the effect of varying X_i alone, but averaged over variations in other input parameters. It is standardized by the total variance to provide a fractional contribution. Higher-order interaction indices S_{ij}, S_{ijk} and so on can be formed by dividing other terms in the variance decomposition by $Var(Y)$.

i Randomized quasi-Monte Carlo for design matrices

Sobol sensitivity analysis requires multiple **independent** design matrices (A, B, and cross-matrices AB_i) so that the variance decomposition is unbiased. A plain deterministic Sobol sequence split across matrices shares the same underlying points, violating that independence assumption.

Passing R = Shift() to SobolSample applies a Cranley-Patterson rotation — a uniform random shift modulo 1 — to each matrix independently. This preserves the low-discrepancy spacing of the Sobol sequence while making each matrix a genuinely independent draw. Other randomization options such as MatousekScramble or OwenScramble also work but require additional parameters (base, pad).

```
using QuasiMonteCarlo, GlobalSensitivity

# reserve @ year 1/2, interest rate ∈ [0.01, 0.05], prem ∈ [500, 1500]
L, U = QuasiMonteCarlo.generate_design_matrices(
    10_000,
    [1, 0.01, 500],
    [2, 0.05, 1500], SobolSample(R=Shift())
)
s = gsa(r1_wrt_r, Sobol(), L, U)
println(s.S1)
println(s.ST)

[0.00016123403439898283, 0.08297764242619343, 0.03237349746877271]
[0.00018803191755323144, 0.08227943051041726, 0.03255829162491274]
```

S1 reports how much of the reserve variance comes from each input alone, while ST shows the total contribution including interaction terms. If rate and premium interact strongly, the gap ST - S1 will be large, signaling non-linear cross-effects. Note that with small sample sizes, Sobol index estimates can occasionally fall outside the theoretical [0, 1] range; increasing the number of design-matrix samples improves the estimates.

28.4.4. Morris Method

The Morris method, also known as Morris’s OAT method (where OAT stands for One At a Time), can be described in the following steps:

$$EE_i = \frac{f(x_1, x_2, \dots, x_i + \Delta, \dots, x_k) - y}{\Delta}$$

28. Sensitivity Analysis

We calculate local sensitivity measures known as “elementary effects”, which are calculated by measuring the perturbation in the output of the model on changing one parameter.

These are evaluated at various points in the input chosen such that a wide “spread” of the parameter space is explored and considered in the analysis, to provide an approximate global importance measure. The mean and variance of these elementary effects is computed. A high value of the mean implies that a parameter is important, a high variance implies that its effects are non-linear or the result of interactions with other inputs. This method does not evaluate separately the contribution from the interaction and the contribution of the parameters individually and gives the effects for each parameter which takes into consideration all the interactions and its individual contribution.

```
using GlobalSensitivity

# reserve @ year 1/2, interest rate ∈ [0.01, 0.05], prem ∈ [500, 1500]
m = gsa(r1_wrt_r, Morris(), [[1, 2], [0.01, 0.05], [500, 1500]])
println(m.means)
println(m.variances)

[0.0 -467438.5786591751 -22.681630357494786]
[0.0 1.2780615554351424e11 15.151055311448006]
```

From the means it can be observed which variables are more important, and the variances imply higher degree of nonlinearity or interactions with other variables.

28.4.5. Fourier Amplitude Sensitivity Tests

FAST (Fourier Amplitude Sensitivity Tests) offers a robust and computationally efficient procedure, especially at low sample sizes, to get the first and total order indices as discussed in Sobol. It utilizes monodimensional Fourier decomposition along a curve exploring the parameter space. Each input factor is varied along the curve via a parametric equation,

$$x_i(s) = G_i(\sin(\omega_i s))$$

where s is a scalar variable varying over the range $-\pi < s < \pi$, G_i are transformation functions and $\omega_i, \forall i = 1, 2, \dots, N$ is a set of different (angular) frequencies, to be properly selected, associated with each factor.

```
using GlobalSensitivity

# reserve @ year 1/2, interest rate ∈ [0.01, 0.05], prem ∈ [500, 1500]
fast = gsa(r1_wrt_r, eFAST(), [[1, 2], [0.01, 0.05], [500, 1500]],
            ↵ samples=1000)
println(fast.S1)
println(fast.ST)

[0.0014284012640849253 0.5938878712341962 0.3797879040012884]
[0.0026466170135476252 0.6163457880842091 0.402489411601634]
```

Interpretation mirrors Sobol: S1 is the main effect and ST the total effect. Differences between Sobol and FAST estimates mostly reflect Monte Carlo noise; agreement across both methods boosts confidence in your ranking.

28.4.6. Automatic Differentiation

By applying the chain rule repeatedly on elementary operations of computer calculations, automatic differentiation can be applied to measure impacts of small differences. More details in Chapter 16 on automatic differentiation.

28.4.7. Scenario Analyses

Quantitative sensitivities are only half the story. Boards and regulators still expect named scenarios that link narrative shocks to financial impacts. Use the techniques from Chapter 25 to build the paths, then layer on the following practices.

28.4.7.1. Reverse stress testing

Reverse stress tests start from an unacceptable outcome (e.g., statutory RBC < 250%) and work backward to the combination of market, actuarial, and operational events that would trigger it.

1. Define the failure point in business terms.
2. Enumerate the drivers that could push the system there (rate shocks, mass lapses, cyber loss).
3. Build a trajectory that connects today's state to the failure and quantify the cumulative impact.

Benefits include clearer vulnerability maps, stronger contingency plans, and alignment with supervisory expectations.

28.4.7.2. Stylistic scenarios

Stylistic (narrative) scenarios translate technical shocks into stories that executives recognize ("Higher-for-longer inflation with elevated credit spreads"). Describe the macro setting, policy responses, and customer behavior, then map those story elements to the stochastic blocks in your engine. This makes it easier to compare sensitivities across business units that may not share the same quantitative models.

28.4.7.3. Backtesting scenarios

Before anchoring capital plans on a scenario set, test whether similar historical episodes would have produced the modeled impacts.

1. Define the scenario family (base/up/down, climate pathways, liquidity crunch).
2. Pull historical periods that resemble each narrative and compute the actual KPI movements.
3. Run the scenario through your model for those historical start dates.
4. Compare modeled vs. realized outcomes and reconcile gaps.
5. Adjust assumptions or document why the future is expected to differ (structural breaks, new hedging programs).

Key reminders when using history: validate data quality, resist overfitting scenarios to the past, and note that structural changes (regulation, product mix, technology) can limit comparability.

28. Sensitivity Analysis

29. Automatic Differentiation for Asset Liability Management

CHAPTER AUTHORED BY: ALEC LOUDENBACK

29.1. Chapter Overview

Asset liability modeling requires computing derivatives of portfolio values with respect to yield curve changes. Traditional approaches use finite difference methods or analytical approximations, but automatic differentiation (“autodiff” or “AD”) provides exact derivatives with minimal additional computation. This chapter demonstrates how to implement ALM workflows using autodiff in Julia.

```
using FinanceCore: Cashflow      # provides Cashflow object
using DifferentiationInterface # autodiff
using DifferentiationInterface: AutoForwardDiff, gradient, hessian
using DifferentiationInterface: value_gradient_and_hessian
import ForwardDiff             # specific autodiff technique
using CairoMakie              # plotting
using DataInterpolations       # yield curve interpolation
using Transducers               # data aggregation
using JuMP, HiGHS              # portfolio optimization
using LinearAlgebra             # math
using BenchmarkTools            # benchmarking
using OhMyThreads              # multi-threading
```

29.2. Interest Rate Curve Setup

We start by constructing a yield curve using cubic spline interpolation:

The `curve` function creates a discount factor curve from zero rates and time points. This curve will serve as input to our `value` function, which makes it straightforward to compute sensitivities by differentiating with respect to the rate parameters.

```
zero_rates = [0.01, 0.02, 0.02, 0.03, 0.05, 0.055] # continuous, annualised
times = [1.0, 2.0, 3.0, 5.0, 10.0, 20.0]

function curve(zero_rates, times)
    discounts = [1.0; exp.(-zero_rates .* times)]
    DataInterpolations.CubicSpline(discounts, [0.; times])
end

c = curve(zero_rates, times)
```

CubicSpline with 7 points

| t | u |
|------|----------|
| 0.0 | 1.0 |
| 1.0 | 0.99005 |
| 2.0 | 0.960789 |
| 3.0 | 0.941765 |
| 5.0 | 0.860708 |
| 10.0 | 0.606531 |
| 20.0 | 0.332871 |

29.3. Asset Valuation Framework

The core valuation function operates on any instrument that produces cashflows:

```
function value(curve, asset)
    cfs = asset(curve)
    mapreduce(cf -> cf.amount * curve(cf.time), +, cfs)
end

value (generic function with 1 method)
```

This design separates the valuation logic from the instrument definition. Each asset type implements a callable interface that generates cashflows given a yield curve. Note how the asset itself gets passed the curve (the `asset(curve)` statement) to determine the cashflows.

For fixed bonds, we create a structure that generates periodic coupon payments:

```
struct FixedBond{T<:Real}
    coupon::T
    tenor::Int
    periodicity::Int
end

function (b::FixedBond)(curve)
    map(1//b.periodicity:1//b.periodicity:b.tenor) do t
        Cashflow(b.coupon / b.periodicity + (t == b.tenor ? 1. : 0.), t)
    end
end

function par_yield(discount_curve, tenor, periodicity)
    dfs = discount_curve.(1//periodicity:1//periodicity:tenor)

    (1 - last(dfs)) / sum(dfs) * periodicity
end
```

The `(b::FixedBond)(curve)` function (a **callable struct**, since we are using the `b` object itself as the function invocation) takes the curve and returns an array of `Cashflows`.

i Note

Cashflow objects are part of the JuliaActuary suite. This allows the cashflows to be tied with the exact timepoint that they occur, rather than needing a bunch of logic to pre-determine a timestep (annual, quarterly, etc.) for which cashflows would get bucketed. This is more efficient in many cases and much simpler code.

The `par_yield` function computes the coupon rate that prices the bond at par, which we'll use to construct our asset universe.

Here's an example of bond cashflows and valuing that bond using the curve `c` that we constructed earlier.

```
FixedBond(0.08, 10, 2)(c)
```

```
20-element Vector{Cashflow{Float64, Rational{Int64}}}:
Cashflow{Float64, Rational{Int64}}(0.04, 1//2)
Cashflow{Float64, Rational{Int64}}(0.04, 1//1)
Cashflow{Float64, Rational{Int64}}(0.04, 3//2)
Cashflow{Float64, Rational{Int64}}(0.04, 2//1)
Cashflow{Float64, Rational{Int64}}(0.04, 5//2)
Cashflow{Float64, Rational{Int64}}(0.04, 3//1)
Cashflow{Float64, Rational{Int64}}(0.04, 7//2)
Cashflow{Float64, Rational{Int64}}(0.04, 4//1)
Cashflow{Float64, Rational{Int64}}(0.04, 9//2)
Cashflow{Float64, Rational{Int64}}(0.04, 5//1)
Cashflow{Float64, Rational{Int64}}(0.04, 11//2)
Cashflow{Float64, Rational{Int64}}(0.04, 6//1)
Cashflow{Float64, Rational{Int64}}(0.04, 13//2)
Cashflow{Float64, Rational{Int64}}(0.04, 7//1)
Cashflow{Float64, Rational{Int64}}(0.04, 15//2)
Cashflow{Float64, Rational{Int64}}(0.04, 8//1)
Cashflow{Float64, Rational{Int64}}(0.04, 17//2)
Cashflow{Float64, Rational{Int64}}(0.04, 9//1)
Cashflow{Float64, Rational{Int64}}(0.04, 19//2)
Cashflow{Float64, Rational{Int64}}(1.04, 10//1)
```

```
value(c, FixedBond(0.09, 10, 2))
```

```
1.3526976075662451
```

29.4. Liability Modeling

Deferred annuities require more complex modeling than fixed bonds due to policyholder behavior (optionality). The surrender rate depends on the difference between market rates and the guaranteed rate. The surrender function chosen below is arbitrary, but follows a typical pattern with much higher surrenders if the market rate on competing instruments is higher than what's currently available. The account value accumulates at the guaranteed rate, and surrenders create negative cashflows representing benefit payments. Lastly, the `annuities` function is a wrapper function we will use to compute the portfolio value and ALM metrics later.

```
begin
    struct DeferredAnnuity{A,B}
```

```

        tenor::A
        rate::B
    end

    function (d::DeferredAnnuity)(curve)
        av = 1.
        map(1//12:1//12:d.tenor) do t

            # apply interest credit
            av *= exp(d.rate / 12)

            # determine market comparable rate and surrender rate
            mkt_rate = -log(curve(d.tenor) / curve(t)) / (d.tenor - t)
            rate_diff = mkt_rate - d.rate
            sr = t == d.tenor ? 1.0 : surrender_rate(rate_diff) / 12
            av_surr = av * sr
            av -= av_surr
            Cashflow(-av_surr, t)

        end
    end

    function surrender_rate(rate_diff)
        1 / (1 + exp(3 - rate_diff * 60))
    end
    function annuities(rates, portfolio)
        times = [1.0, 2.0, 3.0, 5.0, 10.0, 20.0]
        c = curve(rates, times)
        OhMyThreads.tmapreduce(
            +,
            1:length(portfolio),
            ntasks=Threads.nthreads(),
        ) do i
            value(c, portfolio[i])
        end
    end
end

annuities (generic function with 1 method)

```

Here's what the surrender rate behavior looks like for different levels of market rates compared to a 3% crediting rate.

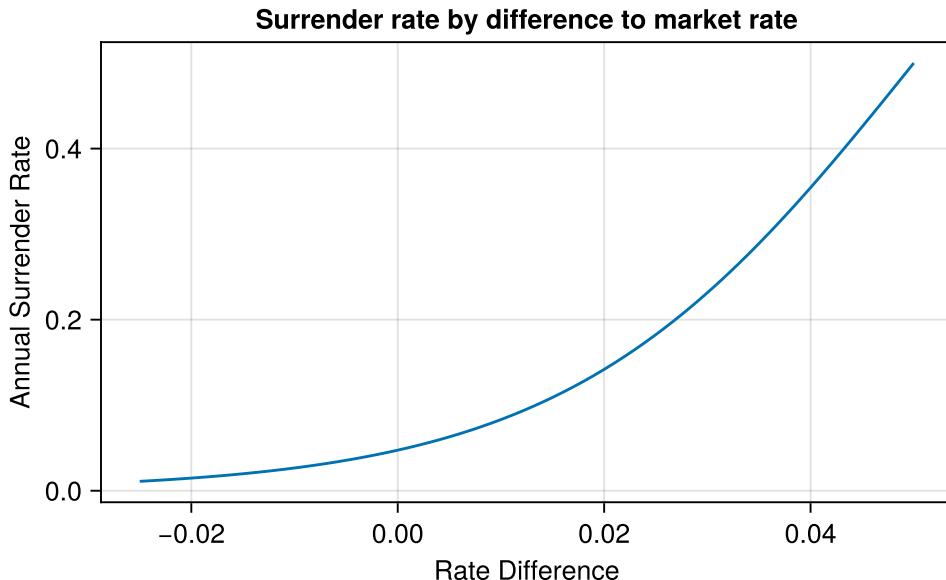
```

let
    cred_rate = 0.03
    mkt_rates = 0.005:0.001:0.08
    rate_diff = mkt_rates .- cred_rate

    lines(rate_diff, surrender_rate.(rate_diff),
        axis=(
            title="Surrender rate by difference to market rate",
            xlabel="Rate Difference",
            ylabel="Annual Surrender Rate"
        ))

```

```
end
```



i Note

In production ALM runs, one may typically evaluate hundreds of thousands of in-force contracts. The threaded annuities aggregator mirrors that setup: it delivers the same projected liability present value while keeping the code path fully differentiable.

We model a large portfolio of these annuities with random tenors:

```
liabilities = map(1:100_000) do i
    tenor = rand(1:20)
    DeferredAnnuity(tenor, par_yield(c, tenor, 12))
end

100000-element Vector{DeferredAnnuity{Int64, Float64}}:
DeferredAnnuity{Int64, Float64}(3, 0.019937936965500825)
DeferredAnnuity{Int64, Float64}(10, 0.04698961087662551)
DeferredAnnuity{Int64, Float64}(7, 0.03847750979086058)
DeferredAnnuity{Int64, Float64}(20, 0.051933558828553925)
DeferredAnnuity{Int64, Float64}(15, 0.05141229637424002)
DeferredAnnuity{Int64, Float64}(7, 0.03847750979086058)
DeferredAnnuity{Int64, Float64}(19, 0.05183909988436309)
DeferredAnnuity{Int64, Float64}(14, 0.05108046813021651)
DeferredAnnuity{Int64, Float64}(17, 0.051714618573584406)
DeferredAnnuity{Int64, Float64}(5, 0.029452531739357923)
:
DeferredAnnuity{Int64, Float64}(11, 0.048613487329580624)
DeferredAnnuity{Int64, Float64}(20, 0.051933558828553925)
DeferredAnnuity{Int64, Float64}(8, 0.0419258485355399)
```

```
DeferredAnnuity{Int64, Float64}(14, 0.05108046813021651)
DeferredAnnuity{Int64, Float64}(8, 0.0419258485355399)
DeferredAnnuity{Int64, Float64}(14, 0.05108046813021651)
DeferredAnnuity{Int64, Float64}(4, 0.023834377623749747)
DeferredAnnuity{Int64, Float64}(14, 0.05108046813021651)
DeferredAnnuity{Int64, Float64}(7, 0.03847750979086058)
```

Consolidating Cashflows

Later on we will generate vectors of vectors of cashflows without any guarantee that the timepoints will line up, making aggregating cashflows by timepoint a non-obvious task. There are many ways to accomplish this, but I like Transducers.

Transducers are unfamiliar to many people, and don't let their presence deter you from the main points of this chapter. The details aren't central to the point of this chapter so just skip over if confusing.

```
function consolidate(cashflows)

    cashflows ▷ # take the collection
    MapCat(identity) ▷ # flatten it out without changing elements
    # group by the time, and just keep and sum the amounts
    GroupBy(x -> x.time, Map(last) ; Map(x -> x.amount), +) ▷
        # perform the aggregation and keep the final grouped result
        foldxl(Transducers.right)
end

consolidate (generic function with 1 method)
```

Example:

```
cashflow_vectors = [l(c) for l in liabilities];
```

And running `consolidate` groups the cashflows into timepoint \Rightarrow amount pairs.

```
consolidate(cashflow_vectors);
```

Here's a visualization of the liability cashflows, showing that when the interest rates are bumped up slightly, there are more surrenders that occur earlier on (so there are fewer policies around at the time of each maturity). Negative cashflows are outflows:

```
let
    d = consolidate([p(c) for p in liabilities])
    ks = collect(keys(d)) ▷ sort!
    vs = [d[k] for k in ks]

    c2 = curve(zero_rates .+ 0.005, times)
    d2 = consolidate([p(c2) for p in liabilities])
    ks2 = collect(keys(d2)) ▷ sort!
    vs2 = [d2[k] for k in ks2]

    f = Figure(size = (900, 600))
    ax = Axis(f[1, 1],
              xlabel = "Time (Years)",
```

```

        ylabel = "Cashflow Amount (cumulative)",
        title = "Cumulative Liability Cashflows: Base vs +50bp Rate Shock",
    )

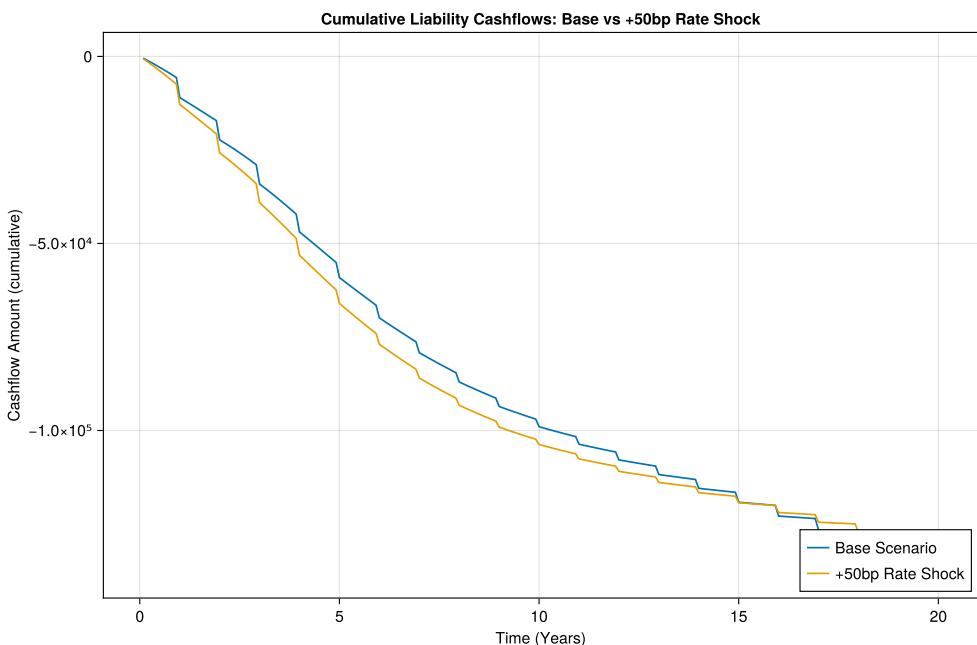
    lines!(ax, ks, cumsum(vs), label = "Base Scenario")

    lines!(ax, ks2, cumsum(vs2), label = "+50bp Rate Shock")

    axislegend(ax, position = :rb)

f
end

```



With an upward-sloping yield curve, without a surrender charge or market value adjustment, many mid-to-late-duration policyholders elect to surrender instead of holding to maturity.

29.5. Computing Derivatives with Autodiff

Rather than approximating derivatives through finite differences, autodiff computes exact values, gradients, and Hessians. The concepts and background are covered in Chapter 16.

The `value_gradient_and_hessian` function returns the present value, key rate durations (gradient), and convexities (Hessian diagonal) for the entire liability portfolio. We compute similar derivatives for each potential asset.

```

vgh_liab = let
    f = z -> annuities(z, liabilities)
    value_gradient_and_hessian(f, AutoForwardDiff(), zero_rates)
end

```

29. Automatic Differentiation for Asset Liability Management

```
(-102327.90959478273, [11590.158497585686, 29860.895743633464, 29869.614639329026,
↳ 255252.84637391486, 123705.03857983049, 6654.040166069367],
↳ [-167589.05983953836 197564.9531012995 ... 8765.080609848334 -41901.35835881232;
↳ 197564.95310129927 -1.677989126507484e6 ... 2.477337080789436e6
↳ 1.2669101394651898e6; ... ; 8765.080609848246 2.4773370807894384e6 ...
↳ -3.703461829065485e7 -2.7232761847430645e6; -41901.358358812366
↳ 1.2669101394651898e6 ... -2.7232761847430705e6 -2.1538880788064003e7])
```

29.5.1. Gradients and Hessians in ALM

Let's dive into the results here a little bit.

The first element of `vgh_liab` is the value of the liability portfolio using the yield curve constructed earlier:

```
vgh_liab[1]
```

```
-102327.90959478273
```

The second element of `vgh_liab` is the partial derivative with respect to each of the inputs (here, just the `zero_rates` that dictate the curve). The sum of the partials is the effective duration of the liabilities.

```
@show sum(vgh_liab[2])
vgh_liab[2]
```

```
sum(vgh_liab[2]) = 456932.5940003629
```

```
6-element Vector{Float64}:
11590.158497585686
29860.895743633464
29869.614639329026
255252.84637391486
123705.03857983049
6654.040166069367
```

This is the sensitivity relative to a full unit change in rates (e.g. 1.0). So if we wanted to estimate the dollar impact of a 50 basis point (0.50%) change, we would take 0.005 times the gradient/Hessian. Also note these are dollar durations but we could divide by the price to get effective or percentage durations:

```
-sum(vgh_liab[2]) / vgh_liab[1]
```

```
4.46537602311833
```

Additionally, note that this is the dynamic duration of the liabilities, not the static duration which ignores the effect of the interest-sensitive behavior of the liabilities.

```
let
    dynamic(zero_rates) = value(curve(zero_rates,times),liabilities[1])
    cfs = liabilities[1](c)
    static(zero_rates) = let
        c = curve(zero_rates,times)
        # note that `cfs` are defined outside of the function, so
        # will not change as the curve is sensitized
        mapreduce(cf -> c(cf.time) * cf.amount,+,cfs)
```

```

    end

    @show gradient(dynamic,AutoForwardDiff(),zero_rates) ▷ sum
    @show gradient(static,AutoForwardDiff(),zero_rates) ▷ sum
end

gradient(dynamic, AutoForwardDiff(), zero_rates) |> sum = 2.732406888412598
gradient(static, AutoForwardDiff(), zero_rates) |> sum = 2.7688749148072453

2.7688749148072453

```

Due to the steepness of the surrender function, the policy exiting sooner, on average, results in a higher change in value than if the policy was not sensitive to the change in rates. The increase in value from earlier cashflows outweighs the greater discount rate.

The third element of `vgh_liab` is the Hessian matrix, containing all second partial derivatives with respect to the yield curve inputs:

```

vgh_liab[3]

6x6 Matrix{Float64}:
 -1.67589e5 197565.0      -2.17473e5 ... 8765.08      -41901.4
 197565.0      -1.67799e6 2.86222e6      2.47734e6 1.26691e6
 -2.17473e5 2.86222e6 -9.86242e6      -4.21954e6 -2.75379e6
 6.51916e5 -3.30837e6 1.26571e7      1.88466e7 8.0982e6
 8765.08 2.47734e6 -4.21954e6      -3.70346e7 -2.72328e6
 -41901.4 1.26691e6 -2.75379e6 ... -2.72328e6 -2.15389e7

```

This matrix captures the convexity characteristics of the liability portfolio. The diagonal elements represent “key rate convexities”—how much the duration at each key rate changes as that specific rate moves:

```

@show diag(vgh_liab[3])
@show sum(diag(vgh_liab[3])) # Total dollar convexity

diag(vgh_liab[3]) = [-167589.05983953836, -1.677989126507484e6,
↪ -9.862424946467455e6, -2.8595528040819723e7, -3.703461829065485e7,
↪ -2.1538880788064003e7]
sum(diag(vgh_liab[3])) = -9.887703025235306e7

-9.887703025235306e7

```

Like duration, we can convert dollar convexity to percentage convexity by dividing by the portfolio value:

```

sum(diag(vgh_liab[3])) / vgh_liab[1]

966.2762646467116

```

The off-diagonal elements show cross-convexities—how the sensitivity to one key rate changes when a different key rate moves. For most portfolios, these cross-terms are smaller than the diagonal terms but can be significant for complex instruments.

This convexity measurement is also dynamic, capturing how the surrender behavior changes the second-order interest rate sensitivity:

29. Automatic Differentiation for Asset Liability Management

```

let
    dynamic(zero_rates) = value(curve(zero_rates,times),liabilities[1])
    cfs = liabilities[1](c)
    static(zero_rates) = let
        c = curve(zero_rates,times)
        mapreduce(cf -> c(cf.time) * cf.amount,+,cfs)
    end

    a = hessian(dynamic,AutoForwardDiff(),zero_rates) ▷ diag ▷ sum
    b = hessian(static,AutoForwardDiff(),zero_rates) ▷ diag ▷ sum

    a, b
end

```

(-117.02641335978038, -8.079255691536465)

The dynamic convexity differs from static convexity because the surrender function creates path-dependent behavior. As rates change, not only do the discount factors change, but the timing and magnitude of cashflows shift as well. This interaction between discount rate changes and cashflow timing changes produces the additional convexity captured in the dynamic measurement. Note how the convexity is larger in the dynamic case.

For ALM purposes, this convexity information helps quantify how well a duration-matched hedge will perform under large rate movements.

29.6. Optimizing an Asset Portfolio

29.6.1. Define Asset Universe

We will create a set of par bonds and select a portfolio of assets that matches the liabilities, subject to duration and KRD constraints:

```

asset_universe = [
    FixedBond(par_yield(c,t,4),t,4)
    for t in 1:20
]

```

20-element Vector{FixedBond{Float64}}:

```

FixedBond{Float64}(0.009998004795647176, 1, 4)
FixedBond{Float64}(0.019932158064569137, 2, 4)
FixedBond{Float64}(0.01997170973543043, 3, 4)
FixedBond{Float64}(0.02388292451655035, 4, 4)
FixedBond{Float64}(0.02952635925170046, 5, 4)
FixedBond{Float64}(0.03446708593197626, 6, 4)
FixedBond{Float64}(0.038602669873830896, 7, 4)
FixedBond{Float64}(0.04207416430705521, 8, 4)
FixedBond{Float64}(0.04493178862970366, 9, 4)
FixedBond{Float64}(0.04717539935714873, 10, 4)
FixedBond{Float64}(0.04881213298628277, 11, 4)
FixedBond{Float64}(0.04996932107971216, 12, 4)
FixedBond{Float64}(0.050768208246499, 13, 4)
FixedBond{Float64}(0.051299382922034086, 14, 4)
FixedBond{Float64}(0.051633982591327676, 15, 4)
FixedBond{Float64}(0.051830458100536235, 16, 4)

```

```
FixedBond{Float64}(0.051938805267308895, 17, 4)
FixedBond{Float64}(0.05200329735010632, 18, 4)
FixedBond{Float64}(0.0520643078824987, 19, 4)
FixedBond{Float64}(0.052159576852118396, 20, 4)
```

And we capture the measures for each of the available assets for the portfolio selection:

```
vgh_assets = [
    value_gradient_and_hessian(
        x -> value(curve(x, times), a), AutoForwardDiff(), zero_rates)
    for a in asset_universe
];
```

29.6.2. Optimization Routine

This optimization function uses functionality from JuMP, a robust optimization library in Julia.

With derivatives available, we can optimize the asset portfolio to match liability characteristics. The optimization maximizes asset yield while constraining the difference between asset and liability key rate durations. This ensures that small yield curve movements don't create large changes in surplus.

```
function optimize_portfolio(
    assets, vgh_assets, liabs, vgh_liabs, constraints)
    n = length(assets)

    # Create model
    model = Model(HiGHS.Optimizer)
    set_silent(model) # Suppress solver output

    # Decision variables: weight vector w
    @variable(model, w[1:n])

    @constraint(model, w .>= 0) # Long-only constraint
    # Budget/asset value constraint
    asset_val = sum(w[i] * vgh_assets[i][1] for i in 1:n)
    @constraint(model, asset_val + vgh_liabs[1] == 0)

    # Objective: Maximize total yield
    @objective(model, Max, sum(w[i] * assets[i].coupon for i in 1:n))

    # Gradient component (krd) constraints
    for j in 1:length(vgh_liabs[2])
        asset_grad = sum(w[i] * vgh_assets[i][2][j] for i in 1:n)
        gradient_gap = asset_grad + vgh_liabs[2][j]
        @constraint(model, gradient_gap >= constraints[:krd][:lower])
        @constraint(model, gradient_gap <= constraints[:krd][:upper])
    end

    # total duration constraint
    asset_dur = sum(w[i] * sum(vgh_assets[i][2]) for i in 1:n)
    duration_gap = asset_dur + sum(vgh_liabs[2])
    @constraint(model, duration_gap <= constraints[:duration][:upper])
    @constraint(model, duration_gap >= constraints[:duration][:lower])
```

```

# Solve
optimize!(model)

status = termination_status(model)
success = string(status) == "OPTIMAL"
weights = success ? JuMP.value.(w) : nothing
obj = success ? objective_value(model) : missing
return (status = status, weights = weights, objective_value = obj)
end

# Define gradient constraints
constraints = Dict(
    :krd => Dict(:lower => -0.35e6, :upper => 0.35e6),
    :duration => Dict(:lower => -0.05e6, :upper => 0.05e6)
)

# Optimize
result = optimize_portfolio(
    asset_universe, vgh_assets, liabilities, vgh_liab, constraints)

(status = OPTIMAL, weights = [-0.0, 45565.3902223364, -0.0, -0.0, -0.0, -0.0,
    -0.0, -0.0, 56762.51937244633, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0,
    -0.0, -0.0, -0.0], objective_value = 3458.6580827176)

```

29.6.3. Results

The optimization produces asset weights that hedge the liability portfolio. We can visualize both the resulting cashflow patterns and the key rate duration matching:

```

let
    d = consolidate([p(c) for p in liabilities])
    ks = collect(keys(d)) ▷ sort!
    vs = -cumsum([d[k] for k in ks])

    f = Figure(size = (900, 600))
    ax = Axis(f[1, 1],
        xlabel = "Time (Years)",
        ylabel = "Cashflow Amount (cumulative)",
        title = "Cumulative Asset vs Liability Cashflows",
    )
    lines!(ax, ks, vs, label = "Liabilities")

    asset_cfs = map(1:length(asset_universe)) do i
        cfs =
            result.weights[i] * asset_universe[i](c)
    end

    d = consolidate(asset_cfs)

```

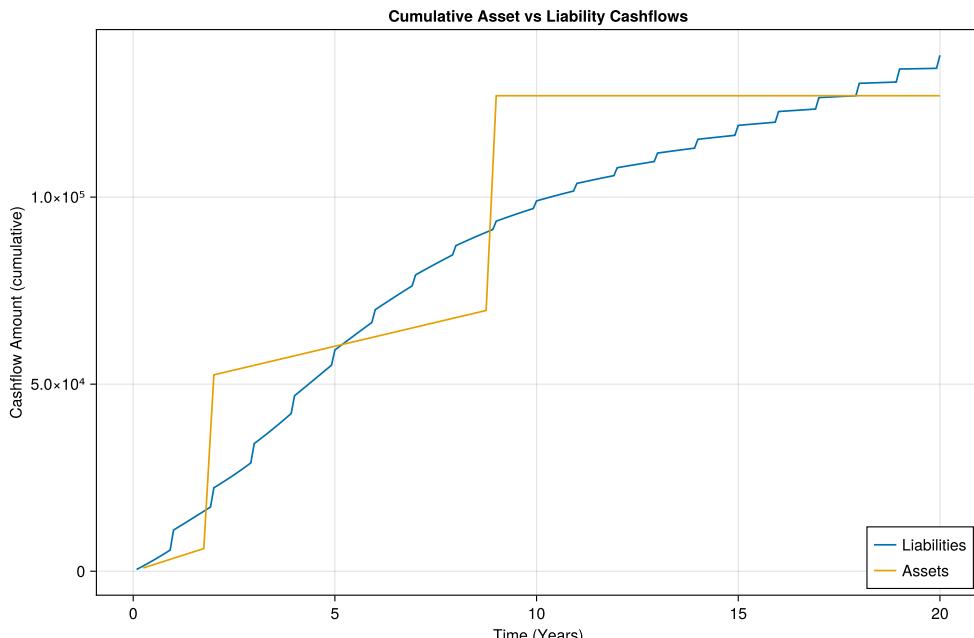
```

ks2 = collect(keys(d)) ▷ sort!
vs2 = cumsum([d[k] for k in ks2])
lines!(ax, ks2, vs2, label = "Assets")

axislegend(ax, position = :rb)

f
end

```



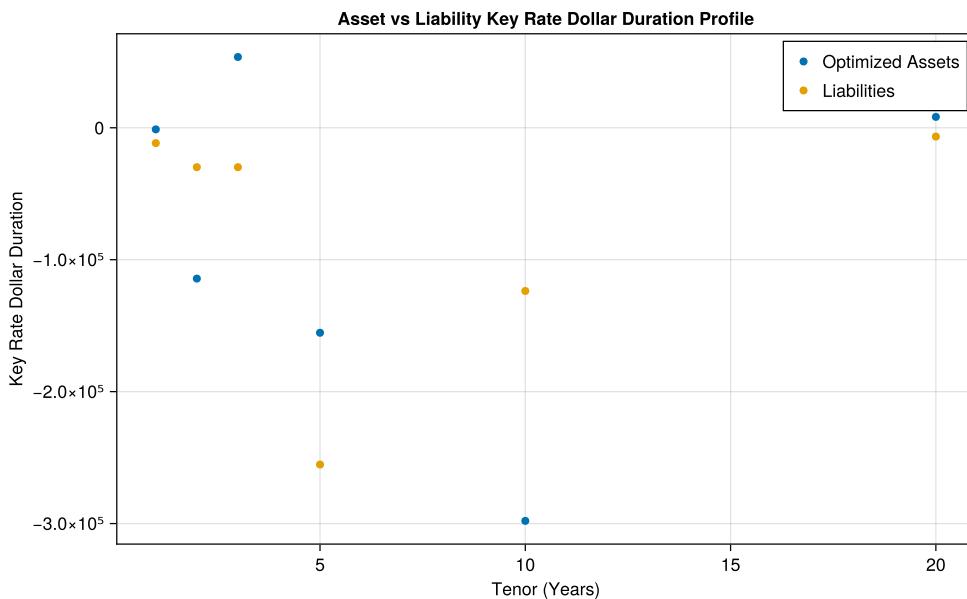
```

let
    asset_krds = sum(getindex.(vgh_assets,2) .* result.weights)
    liab_krds = -vgh_liab[2]

    f = Figure(size = (800, 500))
    ax = Axis(f[1, 1],
              xlabel = "Tenor (Years)",
              ylabel = "Key Rate Dollar Duration",
              title = "Asset vs Liability Key Rate Dollar Duration Profile",
              )
    scatter!(ax, times, asset_krds, label = "Optimized Assets")
    scatter!(ax, times, liab_krds, label = "Liabilities")
    axislegend(ax, position = :rt)
    f

```

```
end
```



The first plot shows the distribution of asset cashflows over time. The second compares the key rate duration profiles of the optimized asset portfolio and the liability portfolio, demonstrating how well the hedge performs across different points on the yield curve.

i Note

Production risk teams could rerun this optimization after every major market move. Having gradients and Hessians on tap means the optimizer converges in seconds, so traders can rebalance hedges intra-day without resorting to coarse stress grids.

29.7. Computational Benefits

Autodiff provides several advantages over traditional finite difference approaches:

- Exact derivatives rather than approximations
- Efficient computation of derivatives without separate finite difference passes for each parameter
- No tuning of step sizes or dealing with numerical artifacts
- Scales efficiently to high-dimensional parameter spaces

For ALM applications, this means more accurate risk measurement and the ability to optimize portfolios with complex constraints that would be computationally expensive using traditional methods.

Here, we value 100,000 interest-sensitive policies with a monthly timestep for up to 20 years *and* compute 1st and 2nd order partial sensitivities extremely quickly:

```
f = z -> annuities(z, liabilities)
@btime value_gradient_and_hessian(\$f, AutoForwardDiff(), \$zero_rates);

3.769 s (127123986 allocations: 35.96 GiB)
```

However, there's still some performance left on the table! The `(d::DeferredAnnuity)(curve)` function defined above is not type stable. In the appendix to this chapter, we'll cover a way to improve the performance even more.

29.8. Conclusion

The Julia ecosystem supports this workflow through packages like `DifferentiationInterface` for autodiff, `JuMP` for optimization, and `FinanceCore` for financial mathematics. This combination enables sophisticated ALM implementations that are both mathematically precise and computationally efficient.

29.9. Appendix: Even more performance (Advanced)

Julia is fastest when all functions are type stable (i.e. the return type can be inferred at compile time). Looking back at the function defined above, the issue is that the `av` variable is defined outside of the scope used within the `map` block. This means that the compiler can't be sure that `av` won't be modified while being used within the `map`. Therefore, `av` gets boxed and held as an `Any` type. This type uncertainty propagates to the value returned from the `(d::DeferredAnnuity)(curve)` function:

```
function (d::DeferredAnnuity)(curve)
    av = 1.0
    map(1//12:1//12:d.tenor) do t
        # apply interest credit
        av *= exp(d.rate / 12)
        # determine market comparable rate and surrender rate
        mkt_rate = -log(curve(d.tenor) / curve(t)) / (d.tenor - t)
        rate_diff = mkt_rate - d.rate
        sr = t == d.tenor ? 1.0 : surrender_rate(rate_diff) / 12
        av_surr = av * sr
        av -= av_surr
        Cashflow(-av_surr, t)
    end
end
```

An alternative would be to write a `for` loop and initialize an array to hold the cashflows. The challenge with that is to concretely define the output type of the resulting array. Particularly when combined with AD, the types within the program are no longer basic floats and integers; we have dual numbers and more complex types running through our functions.

To maintain most of the simplicity, an alternative approach¹ is to use small, immutable containers from `MicroCollections.jl` and combine them with `BangBang.jl`. Then, instead of using `map`, we will write a regular loop. The macro `@unroll` is defined to unroll the first `N` iterations of the loop. This means that the macro transforms the source code to explicitly write out the first two loops. An example of this might be as follows, where two iterations of the loop are unrolled.

¹With thanks to the helpful people on the Julia Zulip and in particular Mason Protter for this approach.

```
function basic_loop()
    out = []
    for i ∈ 1:10
        push!(out,i)
    end
    out
end

function partially_unrolled_loop()
    out = []
    push!(out,1)
    push!(out,2) # two steps unrolled
    for i ∈ 3:10
        push!(out,i)
    end
    out
end
```

Here's the macro that does this transformation for us. It takes a loop and unrolls the first N iterations, then continues with a regular loop for the remaining iterations:

i @unroll macro

```
"""
@unroll N for_loop

Unroll the first 'N' iterations of a for loop,
with remaining iterations handled by a regular loop.

This macro takes a for loop and explicitly expands
the first 'N' iterations, which can improve performance
and type stability, particularly when building collections
where the first few iterations determine the container's type.

# Arguments
- 'N:Int': # of loop iterations to unroll (must be a compile-time
  ↵ constant)
- 'for_loop': A standard for loop expression

"""

macro unroll(N:Int, loop)
    Base.isexpr(loop, :for) || error("only works on for loops")
    Base.isexpr(loop.args[1], :(=)) || error("Loop pattern not
      ↵ supported")
    val, itr = esc.(loop.args[1].args)
    body = esc(loop.args[2])
    @gensym loopend
    label = :(@label $loopend)
    goto = :(@goto $loopend)
    out = Expr(:block, :(itr = $itr), :(next = iterate(itr)))
    unrolled = map(1:N) do _
        quote
            isnothing(next) && @goto loopend
            $val, state = next
            $body
            next = iterate(itr, state)
        end
    end
    append!(out.args, unrolled)
    remainder = quote
        while !isnothing(next)
            $val, state = next
            $body
            next = iterate(itr, state)
        end
        @label loopend
    end
    push!(out.args, remainder)
    out
end
```

Main.Notebook.@unroll

Then, we re-write and redefine (d::DeferredAnnuity)(curve) to utilize this technique.

```

using BangBang, MicroCollections

function (d::DeferredAnnuity)(curve)
    times = 1//12:1//12:d.tenor
    out = UndefVector{Union{}}(length(times)) # 1
    av = 1.0
    @unroll 2 for (i, t) ∈ enumerate(times) # 2
        mkt_rate = -log(curve(d.tenor) / curve(t)) / (d.tenor - t)
        av *= exp(d.rate / 12)
        rate_diff = mkt_rate - d.rate
        sr = t == d.tenor ? 1.0 : surrender_rate(rate_diff) / 12
        av_surr = av * sr
        av -= av_surr
        cf = Cashflow(-av_surr, t)
        out = setindex!!(out, cf, i) # 3
    end
    out
end;

```

1. We tell the `out` vector how many elements to expect.
2. We unroll two iterations of the loop so that the compiler can use the calculated result to determine the type of the output container.
3. We use `setindex!!` from `BangBang` to efficiently update the output vector and its type.

Using this technique, we can see that we achieve a significant speedup (less than half the runtime) from the earlier version due to improving the type stability of the code:

```

f = z -> annuities(z, liabilities)
@btime value_gradient_and_hessian($f, AutoForwardDiff(), $zero_rates);

```

```
1.683 s (600186 allocations: 5.40 GiB)
```

30. Stochastic Mortality Projections

CHAPTER AUTHORED BY: ALEC LOUDENBACK

30.1. Chapter Overview

A term life insurance policy is used to illustrate: selecting key model features, design tradeoffs between a few different approaches, and a discussion of the performance impacts of the different approaches to parallelism.

30.2. Introduction

Monte Carlo simulation is common in risk and valuation contexts. This worked example will create a population of term life insurance policies and simulate the associated claims stochastically. For this chapter, the focus is not so much on the outcomes of the model, but instead *how* and *why* the model was chosen to be set up in the way that it is.

The general structure of the example is:

1. Define the datatypes and sample data
2. Define the core logic that governs the projected outcomes for the modeled policies
3. Evaluate a few ways to structure the simulation, including:
 - allocating and non-allocating approaches
 - single threaded and multi-threaded approaches

As will be shown, the number of simulations able to be completed on modern CPU hardware is really remarkable!

```
using CSV, DataFrames
using MortalityTables, FinanceCore
using Dates
using ChunkSplitters
using BenchmarkTools
using Random
using CairoMakie
```

30.3. Data and Types

30.3.1. @enums and the Policy Type

Our core unit of simulation will be a single life insurance Policy. Important characteristics include: the age a policy was issued at, the sex of the insured, and risk class to determine the assumed mortality rate. To make the example more realistic and demonstrate how it might look for a real block of inforce policies, additional fields have been included such as ID (not really used, but a common identifier) and COLA which is a cost-of-living-adjustment, used to modify the policy

30. Stochastic Mortality Projections

benefit through time. To be clear: the `Policy` type has more fields than will actually be used in the calculations, with the purpose to show how typical fields used in practice might be defined.

Before we define the core `Policy` type, there are a couple of types we might consider defining that would subsequently get used by `Policy`: types representing sex and risk class. A typical approach might be to simply define associated types, like this:

```
abstract type Sex end

struct Male <: Sex end
struct Female <: Sex end
```

Then, if we were to include a `sex` field in `Policy`, we could write it like this:

```
struct Policy
    # ...
    sex::Sex
    # ...
end
```

This would be a totally valid and logical approach! However, high performance is a top priority for this simulation, and therefore this approach would sacrifice being able to keep `Policy` data on the stack instead of the heap. This is because `Sex` is of unknown concrete type, which could be as simple as our definition above (with no fields in `Male` and `Female`) or someone could add a new `Alien` subtype of `Sex` with a number of associated data fields! This is why Julia cannot assume that subtypes of `Sex` will always be simple singletons with no associated data.

Instead, we can utilize Enums, which are a sort of lightweight type where the only thing that matters with it is distinguishing between categories. Enums in Base Julia are basically a set of constants grouped together that reference an associated integer.

```
@enum Sex Female = 1 Male = 2
@enum Risk Standard = 1 Preferred = 2
@enum PaymentMode Annual = 1 Quarterly = 4 Monthly = 12
```

Enums are convenient because they let us use human-meaningful names for integer-based categories. Julia will also keep track of valid options: we cannot now use anything other than `Female` or `Male` where we have said a `Sex` must be specified.

Note

There exist Julia packages which are more powerful versions of Enums, essentially leaning into the ability to use the type system instead of just nice names for categorical variables.

Moving on to the definition of the policy itself, here's what that looks like. Note that every field has a type annotation associated with it.

```
struct Policy
    id::Int
    sex::Sex
    benefit_base::Float64
    COLA::Float64
    mode::PaymentMode
    issue_date::Date
    issue_age::Int
```

```
risk::Risk
end
```

The benefit of the way we have defined it here, using simple bits-types for each field is that our new composite `Policy` type is also a bitstype:

```
let
    p = Policy(1, Male, 1e6, 0.02, Annual, today(), 50, Standard)

    isbits(p)
end

true
```

Note

Type annotations are optional, but providing them is able to coerce the values to be all plain bits (i.e. simple, non-referenced values like arrays are) when the type is constructed. We could have defined `Policy` without the types specified:

```
struct Policy
    id
    sex
    ...
    risk
end
```

Leaving out the annotations here forces Julia to assume that Any type could be used for the given field. Having the field be of type Any makes the instantiated struct data be stored in the heap, since Julia can't know the size of `Policy` in bits in advance.

We would also find that the un-annotated type is about 50 times slower than the one with annotation due to the need to utilize runtime lookup and reference memory on the heap instead of the stack.

30.3.2. The Data

To partially illustrate a common workflow, we'll pretend that the data we are interested in comes from a CSV file, which will be defined inline using an `IOBuffer` so that the structure of the source data is clear to the reader. Only two policies will be listed for brevity, but we will duplicate them for simulation purposes later on.

```
sample_csv_data =
    IOBuffer(
        raw"id,sex,benefit_base,COLA,mode,issue_date,issue_age,risk
        1,M,100000.0,0.03,12,1999-12-05,30,Std
        2,F,200000.0,0.03,12,1999-12-05,30,Pref"
    );
```

We will now load the sample data using a common pattern:

1. Load the source file into a `DataFrame`
2. `map` over each row of the dataframe, and return an instantiated `Policy` object
3. Within the map, apply basic data parsing and translation logic as needed.

30. Stochastic Mortality Projections

```
policies = let
    # read CSV directly into a dataframe
    df = CSV.read(sample_csv_data, DataFrame) ①

    # map over each row and construct an array of Policy objects
    map(eachrow(df)) do row
        Policy(
            row.id,
            row.sex == "M" ? Male : Female,
            row.benefit_base,
            row.COLA,
            PaymentMode(row.mode),
            row.issue_date,
            row.issue_age,
            row.risk == "Std" ? Standard : Preferred,
        )
    end
end
```

- ① `CSV.read("sample_inforce.csv",DataFrame)` could be used if the data really was in a CSV file named `sample_inforce.csv` instead of our demonstration IOBuffer.

```
2-element Vector{Policy}:
Policy(1, Male, 100000.0, 0.03, Monthly, Date("1999-12-05"), 30, Standard)
Policy(2, Female, 200000.0, 0.03, Monthly, Date("1999-12-05"), 30, Preferred)
```

30.4. Model Assumptions

30.4.1. Mortality Assumption

`MortalityTables.jl` provides common life insurance industry tables, and we will use two tables: one each for male and female policies respectively.

```
mort = Dict(
    Male => MortalityTables.table(988).ultimate, ①
    Female => MortalityTables.table(992).ultimate,
)

function mortality(pol::Policy, params)
    return params.mortality[pol.sex]
end
```

- ① `ultimate` refers to not differentiating the mortality by a ‘select’ underwriting period, which is common but unnecessary for the demonstration in this chapter.

```
mortality (generic function with 1 method)
```

30.5. Model Structure and Mechanics

30.5.1. Core Model Behavior

The overall flow of the model loop will be as follows:

1. Determine some initialized values for each policy at the start of the projection.
2. Step through annual timesteps and simulate whether a death has occurred.
 1. If a death has occurred, log the benefit paid out.
 2. If a death has not occurred, keep track of the remaining lives in force and increment policy values.

The code is shown first and then discussion will follow it:

```
function pol_project!(out, policy, params)
    # some starting values for the given policy
    dur = length(policy.issue_date:Year(1):params.val_date) + 1
    start_age = policy.issue_age + dur - 1
    COLA_factor = (1 + policy.COLA)
    cur_benefit = policy.benefit_base * COLA_factor^(dur - 1)

    # get the right mortality vector
    qs = mortality(policy, params)

    w = lastindex(qs)

    @inbounds for t in 1:min(params.proj_length, w - start_age)           (1)

        q = qs[start_age+t] # get current mortality

        if (rand() < q)
            # if dead then just return and don't increment the results
            # anymore
            out.benefits[t] += cur_benefit                                (2)
            return
        else
            # pay benefit, add a life to the output count,
            # and increment the benefit for next year
            out.lives[t] += 1
            cur_benefit *= COLA_factor
        end
    end
end
```

① `inbounds` turns off bounds-checking, which makes hot loops faster but first write the loop without it to ensure you don't create an error (will crash if you have the error without bounds checking)

② Note that the loop is iterating down a column (i.e. across rows) for efficiency (since Julia is column-major).

`pol_project!` (generic function with 1 method)

30.5.2. Inputs and Outputs

30.5.2.1. Inputs

The general approach for non-allocating model runs is to provide a previously instantiated container for the function to write the results to. Here, the incoming argument `out(put)` will be a named tuple with associated vectors as the `lives` and `benefits` fields. We know how many periods the model will simulate for and can therefore size the array appropriately at creation.

Other inputs include: `params` which defines some global-like parameters and `policy` which is a single `Policy` object.

i Note

Note that the unit of the core model logic is a single policy. This simplifies the logic and reduces the chance for error due to needing to code for entire arrays of policies at a single time (as would be the case for array oriented programming style, as described in Section 6.5).

30.5.3. Threading

The simulation uses a threaded parallelism approach with `ChunkSplitters` to divide work across available CPU threads. This pattern ensures thread safety by:

1. Splitting the policy array into chunks, one per thread
2. Each thread maintains its own local output buffer
3. Results are safely combined after all threads complete

Multi-processor (multi-machine) or GPU-based computation would require some modifications; see Chapter 11. For the scale and complexity of this example, thread-based parallelism on a single CPU is all one should need for compute.

30.5.4. Simulation Control

Parameters for our projection:

```
params = (
    val_date=Date(2021, 12, 31),
    proj_length=100,
    mortality=mort,
);
```

Having defined the model behavior at the level of the policy above, we now need to define how the model should iterate over the entire population of interest. Given a vector of `Policies` in the `policies` argument, the `project` function will:

1. Create output containers for each thread chunk.
2. Split work across threads using `ChunkSplitters`, with each thread processing its chunk independently.
3. Spawn tasks for parallel execution and safely combine results.

```
function project(policies, params)
    # Split policies into chunks, one per thread
```

```

tasks = map(chunks(policies; n=Threads.nthreads())) do chunk
    Threads.@spawn begin
        # Each thread gets its own output buffer
        thread_out = (
            benefits=zeros(params.proj_length),
            lives=zeros(Int, params.proj_length)
        )
        # Process all policies in this chunk
        for pol in chunk
            pol_project!(thread_out, pol, params)
        end
        thread_out
    end
end

# Initialize final output
output = (
    benefits=zeros(params.proj_length),
    lives=zeros(Int, params.proj_length)
)

# Safely combine results from all threads
for task in tasks
    thread_result = fetch(task)
    output.benefits .+= thread_result.benefits
    output.lives .+= thread_result.lives
end

output
end

project (generic function with 1 method)

```

30.6. Running the projection

Example of a single projection:

```

project(repeat(policies, 100_000), params);
```

(1)

(1) repeat creates a vector that repeats the two demonstration policies many times.

30.6.1. Stochastic Projection

This defines a loop to calculate the results n times (this is only running the two policies in the sample data n times). This emulates running our population of policies through n stochastic scenarios, similar to what might be done for a risk or pricing exercise.

```

function stochastic_proj(policies, params, n)
    # Split scenarios into chunks for parallel processing
    tasks = map(chunks(1:n; n=Threads.nthreads())) do chunk
        Threads.@spawn begin
            [project(policies, params) for _ in chunk]

```

30. Stochastic Mortality Projections

```
        end
    end

    # Collect all results
    results = []
    for task in tasks
        append!(results, fetch(task))
    end

    results
end

stochastic_proj (generic function with 1 method)
```

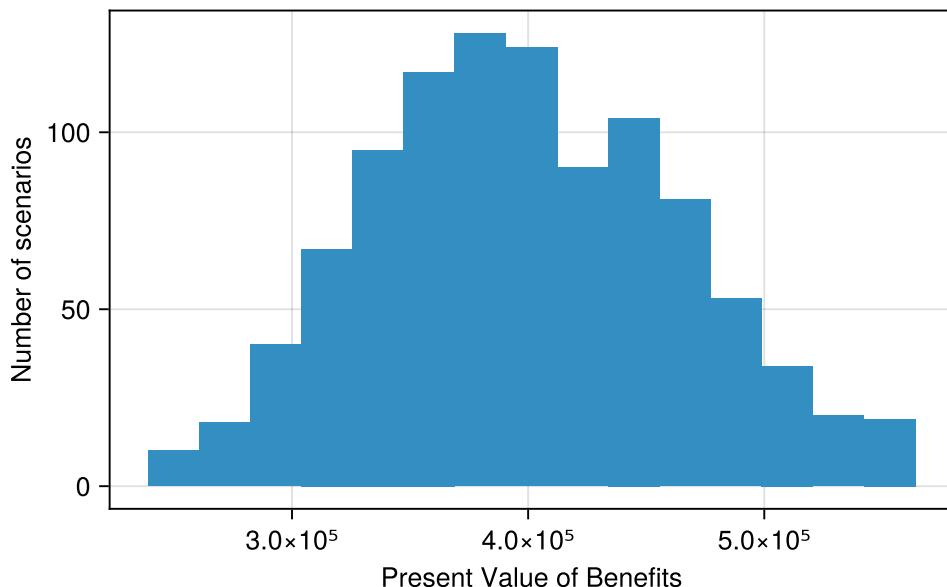
30.6.1.1. Demonstration

We'll simulate the two policies' outcomes 1,000 times and visualize the resulting distribution of claims value:

```
stoch = stochastic_proj(policies, params, 1000);

let
    v = [pv(0.05, s.benefits) for s in stoch] ①
    hist(v,
        bins=15,
        axis=(
            xlabel="Present Value of Benefits",
            ylabel="Number of scenarios"
        )
    )
end
```

- ① The discount rate (5%) exceeds the COLA (3%), so the timing of death affects the present value. Earlier deaths produce higher present values because the discounting effect dominates the benefit growth: while early benefits are smaller, they are discounted less, and the net effect favors earlier payouts.



30.7. Benchmarking

Many millions of policies are able to be stochastically projected per second using a laptop computer¹:

```
policies_to_benchmark = 10_000_000
# adjust the 'repeat' depending on how many policies are already in the array
# to match the target number for the benchmark
n = policies_to_benchmark / length(policies)

@benchmark project(p, r) setup = (p = repeat($policies, $n); r = $params)
```

BenchmarkTools.Trial: 28 samples with 1 evaluation per sample.
Range (min ... max): 137.350 ms ... 180.008 ms GC (min ... max): 0.00% ... 0.00%
Time (median): 138.163 ms GC (median): 0.00%
Time (mean ± σ): 143.900 ms ± 12.610 ms GC (mean ± σ): 0.00% ± 0.00%



Memory estimate: 24.92 KiB, allocs estimate: 126.

¹See the Colophon section in the Introduction for details on the hardware used to render this book.

30.8. Conclusion

This example has worked through a recommended pattern of setting up and running a stochastic simulation using a threaded approach to parallelism with proper thread safety guarantees. The `ChunkSplitters` pattern ensures that:

- Each thread works on its own data without conflicts
- Results are safely combined after parallel execution completes
- The code is maintainable and follows Julia best practices

The results show that quite a bit of simulation power is available using even consumer laptop hardware!

31. Bayesian Mortality Modeling

CHAPTER AUTHORED BY: ALEC LOUDENBACK

"After a year of intense mental struggle, however, [Arthur Bailey] realized to his consternation that actuarial sledgehammering worked. He even preferred [the Bayesian underpinnings of credibility theory] to the elegance of frequentism. He positively liked formulae that described 'actual data. . . . I realized that the hard-shelled underwriters were recognizing certain facts of life neglected by the statistical theorists.' He wanted to give more weight to a large volume of data than to the frequentists' small sample; doing so felt surprisingly 'logical and reasonable.' He concluded that only a 'suicidal' actuary would use Fisher's method of maximum likelihood, which assigned a zero probability to nonevents." - Sharon Bertsch McGrayne, Excerpt From The Theory That Would Not Die

31.1. Chapter Overview

An example of using a Bayesian MCMC approach with Turing.jl to fit a mortality curve to sample data, with multi-level models and censored data.

31.2. Generating fake data

The problem of interest is to look at mortality rates, which are given in terms of exposures (whether or not a life experienced a death in a given year).

We'll grab some example rates from an insurance table, which has a "selection" component: When someone enters observation, say at age 50, their mortality is path dependent (so someone who started being observed at 50 will have a different risk/mortality rate at age 55 than someone who started being observed at 45).

Additionally, there may be additional groups of interest, such as:

- high/medium/low risk classification
- sex
- group (e.g. company, data source, etc.)
- type of insurance product offered

The example data will start with only the risk classification above.

```
using MortalityTables
using Turing
using DataFramesMeta
using MCMCChains
using LinearAlgebra
using CairoMakie
using StatsBase
```

31. Bayesian Mortality Modeling

```
n = 10_000
inforce = [(issue_age=rand(30:70), risk_level=rand(1:3)) for _ in 1:n]

10000-element Vector{@NamedTuple{issue_age::Int64, risk_level::Int64}}:
(issue_age = 62, risk_level = 1)
(issue_age = 46, risk_level = 3)
(issue_age = 59, risk_level = 2)
(issue_age = 31, risk_level = 3)
(issue_age = 65, risk_level = 2)
(issue_age = 53, risk_level = 1)
(issue_age = 43, risk_level = 2)
(issue_age = 58, risk_level = 2)
(issue_age = 43, risk_level = 3)
(issue_age = 31, risk_level = 1)
(issue_age = 54, risk_level = 1)
(issue_age = 68, risk_level = 2)
(issue_age = 65, risk_level = 3)
:
(issue_age = 48, risk_level = 1)
(issue_age = 35, risk_level = 3)
(issue_age = 37, risk_level = 2)
(issue_age = 45, risk_level = 2)
(issue_age = 57, risk_level = 1)
(issue_age = 30, risk_level = 2)
(issue_age = 40, risk_level = 3)
(issue_age = 57, risk_level = 3)
(issue_age = 66, risk_level = 1)
(issue_age = 69, risk_level = 2)
(issue_age = 59, risk_level = 3)
(issue_age = 39, risk_level = 2)

tbl_name = "2001 VBT Residual Standard Select and Ultimate - Male Nonsmoker,
    ↪ ANB"
base_table = MortalityTables.table(tbl_name)

# Risk level multipliers: 1 = preferred (0.7x), 2 = standard (1.0x), 3 =
    ↪ substandard (1.5x)
const RISK_MULTIPLIERS = (0.7, 1.0, 1.5)

function tabular_mortality(params, issue_age, att_age, risk_level)
    params.ultimate[att_age] * RISK_MULTIPLIERS[risk_level]
end

tabular_mortality (generic function with 1 method)

"""
Simulate mortality outcomes for a portfolio of policies.
Returns a DataFrame with one row per policy-year exposure.
"""

function model_outcomes(inforce, assumption, assumption_params; n_years=5)
    # Pre-allocate result vectors
    result_issue_age = Int[]
    result_risk_level = Int[]
    result_att_age = Int[]
```

```

result_death = Int[]

sizehint!(result_issue_age, length(inforce) * n_years)
sizehint!(result_risk_level, length(inforce) * n_years)
sizehint!(result_att_age, length(inforce) * n_years)
sizehint!(result_death, length(inforce) * n_years)

for pol in inforce
    for t in 1:n_years
        att_age = pol.issue_age + t - 1
        q = assumption(assumption_params, pol.issue_age, att_age,
        ← pol.risk_level)
        died = rand() < q

        push!(result_issue_age, pol.issue_age)
        push!(result_risk_level, pol.risk_level)
        push!(result_att_age, att_age)
        push!(result_death, died ? 1 : 0)

        # If died, no more exposures for this policy
        died && break
    end
end

DataFrame(
    issue_age=result_issue_age,
    risk_level=result_risk_level,
    att_age=result_att_age,
    death=result_death,
    exposures=ones(Int, length(result_death)) # each row is one exposure
)
end

exposures = model_outcomes(inforce, tabular_mortality, base_table)

# Aggregate by issue_age and att_age
data = @chain exposures begin
    groupby([:issue_age, :att_age])
    @combine(:exposures = length(:death),
        :deaths = sum(:death),
        :fraction = sum(:death) / length(:death))
end

# Aggregate including risk_level for multi-level modeling
data2 = @chain exposures begin
    groupby([:issue_age, :att_age, :risk_level])
    @combine(:exposures = length(:death),
        :deaths = sum(:death),
        :fraction = sum(:death) / length(:death))
end

```

31. Bayesian Mortality Modeling

| | issue_age | att_age | risk_level | exposures | deaths | fraction |
|-----|-----------|---------|------------|-----------|--------|-----------|
| | Int64 | Int64 | Int64 | Int64 | Int64 | Float64 |
| 1 | 30 | 30 | 1 | 81 | 0 | 0.0 |
| 2 | 30 | 30 | 2 | 96 | 0 | 0.0 |
| 3 | 30 | 30 | 3 | 88 | 0 | 0.0 |
| 4 | 30 | 31 | 1 | 81 | 0 | 0.0 |
| 5 | 30 | 31 | 2 | 96 | 0 | 0.0 |
| 6 | 30 | 31 | 3 | 88 | 0 | 0.0 |
| 7 | 30 | 32 | 1 | 81 | 0 | 0.0 |
| 8 | 30 | 32 | 2 | 96 | 0 | 0.0 |
| 9 | 30 | 32 | 3 | 88 | 0 | 0.0 |
| 10 | 30 | 33 | 1 | 81 | 0 | 0.0 |
| 11 | 30 | 33 | 2 | 96 | 1 | 0.0104167 |
| 12 | 30 | 33 | 3 | 88 | 0 | 0.0 |
| 13 | 30 | 34 | 1 | 81 | 0 | 0.0 |
| 14 | 30 | 34 | 2 | 95 | 0 | 0.0 |
| 15 | 30 | 34 | 3 | 88 | 0 | 0.0 |
| 16 | 31 | 31 | 1 | 89 | 1 | 0.011236 |
| 17 | 31 | 31 | 2 | 77 | 0 | 0.0 |
| 18 | 31 | 31 | 3 | 81 | 0 | 0.0 |
| 19 | 31 | 32 | 1 | 88 | 0 | 0.0 |
| 20 | 31 | 32 | 2 | 77 | 0 | 0.0 |
| 21 | 31 | 32 | 3 | 81 | 0 | 0.0 |
| 22 | 31 | 33 | 1 | 88 | 0 | 0.0 |
| 23 | 31 | 33 | 2 | 77 | 0 | 0.0 |
| 24 | 31 | 33 | 3 | 81 | 0 | 0.0 |
| 25 | 31 | 34 | 1 | 88 | 0 | 0.0 |
| 26 | 31 | 34 | 2 | 77 | 0 | 0.0 |
| 27 | 31 | 34 | 3 | 81 | 1 | 0.0123457 |
| 28 | 31 | 35 | 1 | 88 | 0 | 0.0 |
| 29 | 31 | 35 | 2 | 77 | 0 | 0.0 |
| 30 | 31 | 35 | 3 | 80 | 0 | 0.0 |
| ... | ... | ... | ... | ... | ... | ... |

31.3. 1: A single binomial parameter model

Estimate q , the average mortality rate, not accounting for any variation within the population/sample. Our model is defined as a Beta prior on q with a Binomial likelihood:

$$q \sim \text{Beta}(1, 1)$$

$$p(\text{death}) \sim \text{Binomial}(n, q)$$

```
@model function mortality(exposures, deaths)
    q ~ Beta(1, 1)
    # Vectorized: observe all deaths at once
    for i in eachindex(deaths)
```

```

        deaths[i] ~ Binomial(exposures[i], q)
    end
end

m1 = mortality(data.exposures, data.deaths)

DynamicPPL.Model{typeof(mortality), (:exposures, :deaths), (), ()},
    ↵ Tuple{Vector{Int64}, Vector{Int64}}, Tuple{}, DynamicPPL.DefaultContext,
    ↵ false}(Main.mortality, (exposures = [265, 265, 265, 265, 264, 247, 246, 246,
    ↵ 246, 245 ... 267, 262, 254, 250, 240, 234, 228, 222, 216, 205], deaths = [0,
    ↵ 0, 0, 1, 0, 1, 0, 0, 1, 0 ... 5, 8, 4, 10, 8, 6, 6, 6, 11, 6]), NamedTuple(),
    ↵ DynamicPPL.DefaultContext())

```

31.3.1. Sampling from the posterior

We use a No-U-Turn-Sampler (NUTS) technique to sample multiple chains at once:

```

num_chains = 4
chain = sample(m1, NUTS(), MCMCThreads(), 400, num_chains)

```

Chains MCMC chain (400×15×4 Array{Float64, 3}):

```

Iterations          = 201:1:600
Number of chains   = 4
Samples per chain  = 400
Wall duration      = 1.18 seconds
Compute duration   = 4.7 seconds
parameters         = q
internals          = n_steps, is_accept, acceptance_rate, log_density,
    ↵ hamiltonian_energy, hamiltonian_energy_error, max_hamiltonian_energy_error,
    ↵ tree_depth, numerical_error, step_size, nom_step_size, logprior,
    ↵ loglikelihood, logjoint

```

Use `describe(chains)` for summary statistics and quantiles.

Here, we have asked for the outcomes to be modeled via a single parameter for the population. We see that the posterior distribution of q is very close to the overall population mortality rate:

```

# Posterior mean of q should be close to the pooled fraction
sum(data.deaths) / sum(data.exposures)

```

0.008031802693493429

However, we can see that the sampling of possible posterior parameters doesn't really fit the data very well since our model was so simplified. The lines represent the posterior binomial probability.

This is saying that for the observed data, if there really is just a single probability p that governs the true process that came up with the data, there's a pretty narrow range of values it could possibly be:

```

let
    data_weight = sqrt.(data.exposures) / 2
    f = Figure()
    ax = Axis(f[1, 1],

```

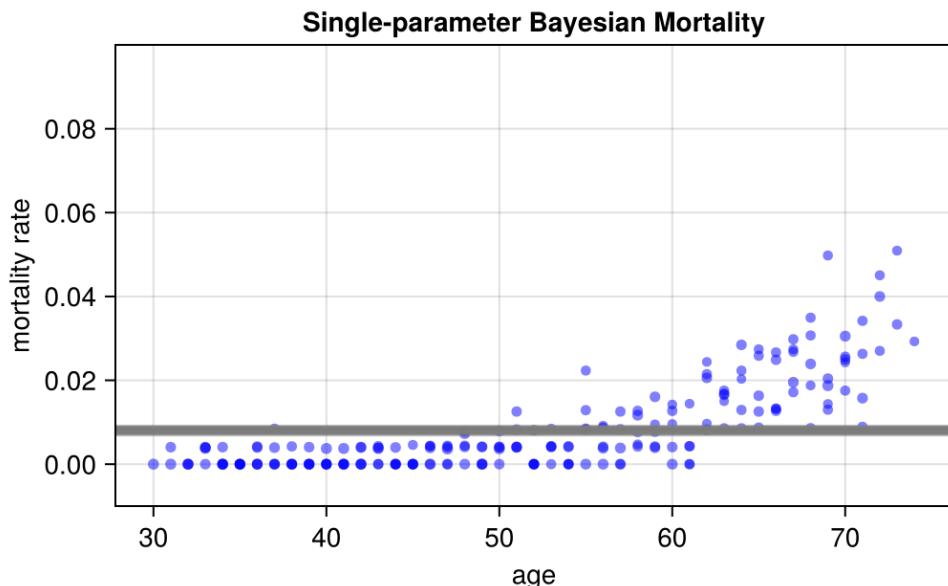
31. Bayesian Mortality Modeling

```
    xlabel="age",
    ylabel="mortality rate",
    limits=(nothing, nothing, -0.01, 0.10),
    title="Single-parameter Bayesian Mortality"
)
scatter!(ax,
    data.att_age,
    data.fraction,
    markersize=data_weight,
    color=(:blue, 0.5),
    label="Experience data (size ~ exposure)")

# Sample from posterior once (more efficient than sampling 1 at a time)
n_samples = 300
q_posterior = vec(Array(sample(chain, n_samples)[:q]))

# Draw horizontal lines for each posterior sample
for q in q_posterior
    hlines!(ax, [q], color=:grey, 0.1)
end

f
end
```



31.4. 2. Parametric model

In this example, we utilize a MakehamBeard parameterization because it's already very similar in form to a logistic function. This is important because our desired output is a probability (i.e., the probability of a death at a given age), so the value must be constrained to be in the interval between zero and one.

The **prior** values for a , b , c , and k are chosen to constrain the hazard (mortality) rate to be between zero and one.

This isn't an ideal parameterization (e.g. we aren't including information about the select underwriting period), but is an example of utilizing Bayesian techniques on life experience data.

```
@model function mortality2(ages, exposures, deaths)
    a ~ Exponential(0.1)
    b ~ Exponential(0.1)
    c = 0.0
    k ~ truncated(Exponential(1), 1, Inf)

    # Create parametric mortality model once
    m = MortalityTables.MakehamBeard(; a, b, c, k)

    # Observe deaths for each age/exposure combination
    for i in eachindex(deaths)
        q = MortalityTables.hazard(m, ages[i])
        if !isfinite(q) || q < 0 || q > 1
            Turing.@addlogprob! -Inf
            return
        end
        deaths[i] ~ Binomial(exposures[i], q)
    end
end

mortality2 (generic function with 2 methods)
```

Guarding against invalid probabilities

During sampling, NUTS uses gradient-based proposals that can explore extreme parameter values. For example, a large b may cause $\exp(b * \text{age})$ to overflow, producing a NaN hazard rate. Since $\text{Binomial}(n, p)$ requires $0 \leq p \leq 1$, passing an invalid value would throw a `DomainError`. The guard clause handles this by calling `Turing.@addlogprob! -Inf`, which sets the log-density to negative infinity and tells the sampler to reject that proposal. This is the idiomatic `Turing.jl` pattern for enforcing domain constraints that arise from complex likelihood computations.

We combine the model with the data and sample from the posterior using a similar call as before:

```
m2 = mortality2(data.att_age, data.exposures, data.deaths)

chain2 = sample(m2, NUTS(), MCMCThreads(), 400, num_chains)
```

Chains MCMC chain (400×17×4 Array{Float64, 3}):

```
Iterations          = 201:1:600
Number of chains   = 4
Samples per chain  = 400
Wall duration      = 4.06 seconds
Compute duration   = 15.26 seconds
parameters         = a, b, k
internals          = n_steps, is_accept, acceptance_rate, log_density,
                     hamiltonian_energy, hamiltonian_energy_error, max_hamiltonian_energy_error,
                     tree_depth, numerical_error, step_size, nom_step_size, logprior,
                     loglikelihood, logjoint
```

31. Bayesian Mortality Modeling

Use `describe(chains)` for summary statistics and quantiles.

31.4.1. Plotting samples from the posterior

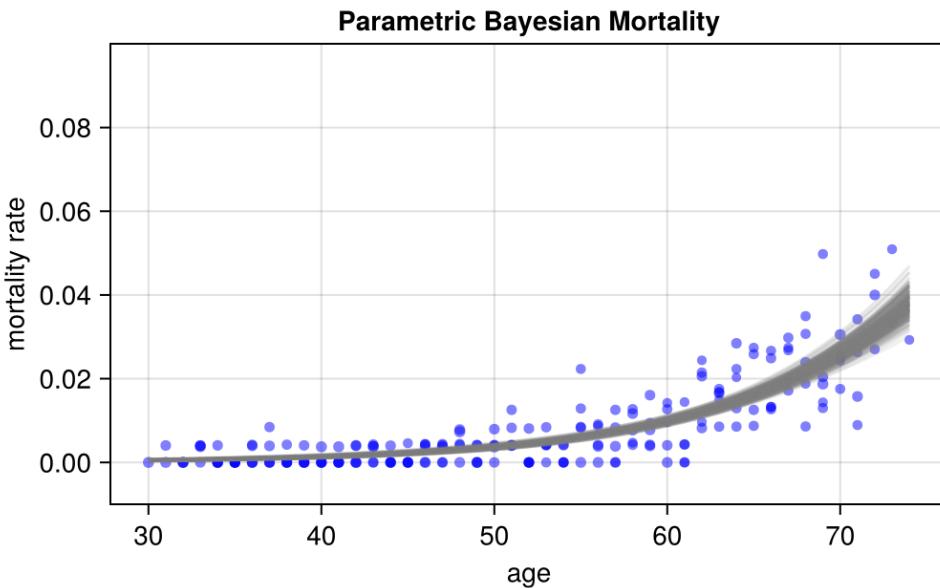
We can see that the sampling of possible posterior parameters fits the data well:

```
let
    data_weight = sqrt.(data.exposures) / 2
    f = Figure()
    ax = Axis(f[1, 1],
        xlabel="age",
        ylabel="mortality rate",
        limits=(nothing, nothing, -0.01, 0.10),
        title="Parametric Bayesian Mortality")
)
scatter!(ax,
    data.att_age,
    data.fraction,
    markersize=data_weight,
    color=(:blue, 0.5),
    label="Experience data (size ~ exposure)")

# Sample from posterior once (much faster than sampling 1 at a time in
# loop)
n_samples = 300
posterior = sample(chain2, n_samples)
a_samples = vec(Array(posterior[:a]))
b_samples = vec(Array(posterior[:b]))
k_samples = vec(Array(posterior[:k]))

ages = sort!(unique(data.att_age))

for i in 1:n_samples
    m = MortalityTables.MakehamBeard(; a=a_samples[i], b=b_samples[i],
    c=0.0, k=k_samples[i])
    qs = MortalityTables.hazard.(m, ages)
    lines!(ax, ages, qs, color=(:grey, 0.1))
end
f
end
```



Recall that the lines are not plotting the possible outcomes of the claim rates, but the *mean* claim rate for the given age.

31.5. 3. Multi-level model

This model extends the prior to create a multi-level model. Each risk class (`risk_level`) gets its own a parameter in the `MakehamBeard` model. The prior for a_i is determined by the hyper-parameter \bar{a} .

```

@model function mortality3(ages, exposures, risk_levels, deaths,
    ↳ n_risk_levels)
    b ~ Exponential(0.1)
    ā ~ Exponential(0.1)
    a ~ filldist(Exponential(ā), n_risk_levels)
    c = 0.0
    k ~ truncated(Exponential(1), 1, Inf)

    # Observe deaths for each row
    for i in eachindex(deaths)
        m = MortalityTables.MakehamBeard(; a=a[risk_levels[i]], b, c, k)
        q = MortalityTables.hazard(m, ages[i])
        if !isfinite(q) || q < 0 || q > 1
            Turing.@addlogprob! -Inf
            return
        end
        deaths[i] ~ Binomial(exposures[i], q)
    end
end

n_risk_levels = length(unique(data2.risk_level))

```

31. Bayesian Mortality Modeling

```
m3 = mortality3(data2.att_age, data2.exposures, data2.risk_level,
← data2.deaths, n_risk_levels)

chain3 = sample(m3, NUTS(), 1000)

summarize(chain3)
```

| parameters | mean | std | mcse | ess_bulk | ess_tail | rhat | e ... |
|------------|---------|---------|---------|----------|----------|---------|-------|
| Symbol | Float64 | Float64 | Float64 | Float64 | Float64 | Float64 | ... |
| b | 0.1007 | 0.0057 | 0.0003 | 296.7129 | 293.7857 | 1.0045 | ... |
| ā | 0.0001 | 0.0001 | 0.0000 | 371.5190 | 275.9341 | 1.0044 | ... |
| a[1] | 0.0000 | 0.0000 | 0.0000 | 298.9985 | 313.7426 | 1.0032 | ... |
| a[2] | 0.0000 | 0.0000 | 0.0000 | 304.0853 | 305.7593 | 1.0024 | ... |
| a[3] | 0.0000 | 0.0000 | 0.0000 | 302.3984 | 253.5755 | 0.9998 | ... |
| k | 1.9784 | 0.9698 | 0.0427 | 388.9334 | 332.9308 | 1.0049 | ... |

1 column omitted

```
let
colors = Makie.wong_colors()
data_weight = sqrt.(data2.exposures)

p, ax, _ = scatter(
    data2.att_age,
    data2.fraction,
    markersize=data_weight,
    alpha=0.5,
    color=[(colors[c], 0.7) for c in data2.risk_level],
    label="Experience data point",
    axis=(
        xlabel="age",
        limits=(nothing, nothing, -0.01, 0.10),
        ylabel="mortality rate",
        title="Multi-level Bayesian Mortality"
    )
)

# Sample from posterior once (much faster than sampling 1 at a time in
# loop)
n_samples = 100
posterior = sample(chain3, n_samples)
b_samples = vec(Array(posterior[:b]))
k_samples = vec(Array(posterior[:k]))

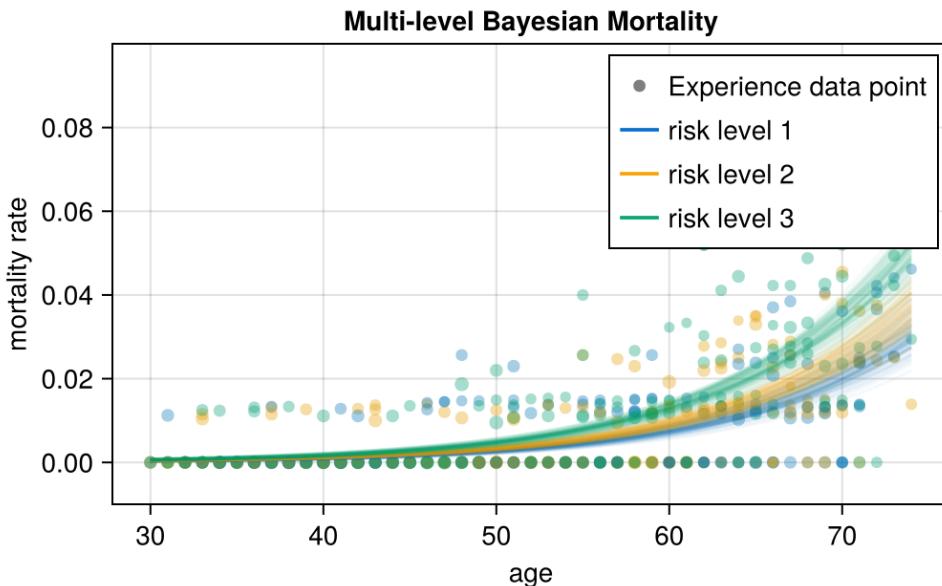
ages = sort!(unique(data2.att_age))

for r in 1:3
    a_samples = vec(Array(posterior[Symbol("a[$r]")])))
    for i in 1:n_samples
        m = MortalityTables.MakehamBeard(; a=a_samples[i],
        ← b=b_samples[i], c=0.0, k=k_samples[i])
```

```

        qs = MortalityTables.hazard.(m, ages)
    lines!(ages, qs, label="risk level $r", alpha=0.2,
          ↵ color=(colors[r], 0.2))
end
end
axislegend(ax, merge=true)
p
end

```



Again, the lines are not plotting the possible outcomes of the claim rates, but the *mean* claim rate for the given age and risk class.

31.6. Handling non-unit exposures

The key is to use the Poisson distribution, which is a limiting approximation to the Binomial distribution:

```

@model function mortality4(ages, exposures, risk_levels, deaths,
    ↵ n_risk_levels)
    b ~ Exponential(0.1)
    ā ~ Exponential(0.1)
    a ~ filldist(Exponential(ā), n_risk_levels)
    c ~ Beta(4, 18)
    k ~ truncated(Exponential(1), 1, Inf)

# Observe deaths for each row using Poisson likelihood
for i in eachindex(deaths)
    m = MortalityTables.MakehamBeard(; a=a[risk_levels[i]], b, c, k)
    q = MortalityTables.hazard(m, ages[i])

```

31. Bayesian Mortality Modeling

```
    deaths[i] ~ Poisson(exposures[i] * q)
end
m4 = mortality4(data2.att_age, data2.exposures, data2.risk_level,
                 data2.deaths, n_risk_levels)
chain4 = sample(m4, NUTS(), 1000)
```

Chains MCMC chain (1000×21×1 Array{Float64, 3}):

```
Iterations      = 501:1:1500
Number of chains = 1
Samples per chain = 1000
Wall duration    = 31.73 seconds
Compute duration = 31.73 seconds
parameters       = b, ā, a[1], a[2], a[3], c, k
internals        = n_steps, is_accept, acceptance_rate, log_density,
                   hamiltonian_energy, hamiltonian_energy_error, max_hamiltonian_energy_error,
                   tree_depth, numerical_error, step_size, nom_step_size, logprior,
                   loglikelihood, logjoint
```

Use `describe(chains)` for summary statistics and quantiles.

```
# Extract posterior means for risk factors and compute relative factors
risk_factors4 = [mean(chain4[Symbol("a[$f]")]) for f in 1:3]
println("Risk factors relative to standard (level 2): ", risk_factors4 ./ risk_factors4[2])

let
    colors = Makie.wong_colors()
    data_weight = sqrt.(data2.exposures) / 2

    p, ax, _ = scatter(
        data2.att_age,
        data2.fraction,
        markersize=data_weight,
        alpha=0.5,
        color=data2.risk_level,
        label="Experience data point",
        axis=(
            xlabel="age",
            limits=(nothing, nothing, -0.01, 0.10),
            ylabel="mortality rate",
            title="Poisson Multi-level Bayesian Mortality"
        )
    )

    # Sample from posterior once (much faster than sampling 1 at a time in
    # loop)
    n_samples = 100
    posterior = sample(chain4, n_samples)
    b_samples = vec(Array(posterior[:b]))
    c_samples = vec(Array(posterior[:c]))
```

```

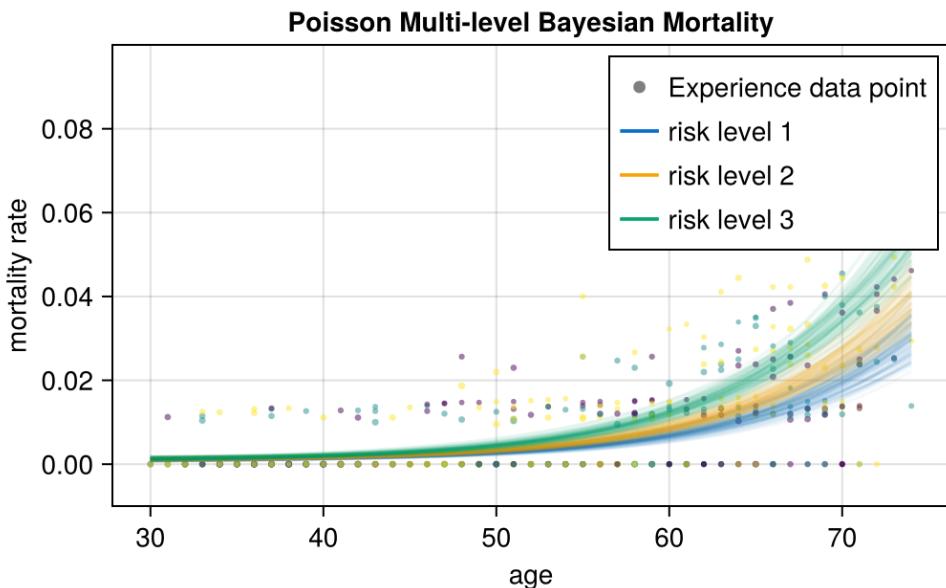
k_samples = vec(Array(posterior[:,k]))

ages = sort!(unique(data2.att_age))

for r in 1:3
    a_samples = vec(Array(posterior[Symbol("a[$r]")])))
    for i in 1:n_samples
        m = MortalityTables.MakehamBeard(; a=a_samples[i],
        ← b=b_samples[i], c=c_samples[i], k=k_samples[i])
        qs = MortalityTables.hazard.(m, ages)
        lines!(ages, qs, label="risk level $r", alpha=0.2,
        ← color=(colors[r], 0.2))
    end
end
axislegend(ax, merge=true)
p
end

Risk factors relative to standard (level 2): [0.7808100858141713, 1.0,
← 1.452909663263604]

```



31.7. Model Predictions

We can generate predictive estimates by passing a vector of `missing` in place of the outcome variables and then calling `predict`.

We get a table of values where each row is the prediction implied by the corresponding chain sample, and the columns are the predicted value for each of the outcomes in our original dataset.

31. Bayesian Mortality Modeling

```
# Create model with missing deaths to generate predictions
pred_model = mortality4(
  data2.att_age,
  data2.exposures,
  data2.risk_level,
  fill(missing, length(data2.deaths)),
  n_risk_levels
)
preds = predict(pred_model, chain4);
```

32. More Useful Techniques

CHAPTER AUTHORED BY: ALEC LOUDENBACK

"All models are wrong, but some are useful." - George Box (1976)

32.1. Chapter Overview

A grab-bag of practical techniques for keeping models honest: sanity checks, serialization patterns for reproducibility, validation workflows, and ways to think about whether a model is doing what you think it's doing.

32.2. General Modeling Techniques

32.2.1. Taking Things to the Extreme

Before trusting any model, ask: what happens at the edges? Set interest rates to zero, or negative. Assume 100% lapse. Perfectly correlated defaults. An illiquid market with zero trades. These extreme scenarios often reveal assumptions you didn't know you had made.

Consider a simple loan loss model. It might work perfectly well under normal conditions, but what happens when recovery rates hit zero? Does the code handle that gracefully, or does it divide by something that's now zero? Extreme thought experiments surface these hidden assumptions before production does.

32.2.2. Range Bounding

Sometimes you don't need the answer—you just need to know that the answer is good enough. If both a pessimistic and an optimistic estimate clear your hurdle, you're done.

Here's a classic example from interview lore: you need to determine whether a mortgaged property's value exceeds the \$100,000 loan balance. No appraisal available. But you know that a comparable house in worse condition sold for \$100 per square foot, and from the floor plan this house must be at least 1,000 square feet. So:

$$\frac{\$100}{\text{sq. ft}} \times 1000\text{sq. ft} = \$100,000$$

The property almost certainly exceeds the loan balance. No complex modeling required.

This technique is particularly useful in early scoping meetings or ad-hoc regulatory requests where a directional answer is all you need.

32.2.3. Pseudo-Monte Carlo Sanity Checks

Before committing to a massive simulation run, do a miniature version first. Fix the random seed, use a handful of scenarios, and verify that everything works end to end. This catches problems like:

- Configuration files that aren't being read correctly
- Aggregation logic that breaks on edge cases
- Performance bottlenecks that will be painful at scale

A ten-scenario dry run that takes five seconds can save you from discovering bugs halfway through an overnight batch job.

32.2.4. Model Validation

32.2.4.1. Static vs. Dynamic

Model validation is essential. The most common validation approach is static: split your data chronologically, fit on the earlier period, and test on the later period. This tells you how well the model generalizes to unseen data.

```
using Random, Statistics, LinearAlgebra

T = 200
x = rand(T)
y = 1.0 .+ 2.0 .* x .+ 0.1 .* randn(T)

# Chronological holdout
cut = 150
Xtrain = hcat(ones(cut), x[1:cut])
ytrain = y[1:cut]
Xtest = hcat(ones(T - cut), x[(cut+1):end])
ytest = y[(cut+1):end]

θ = Xtrain \ ytrain
ŷ = Xtest * θ

println("MSE: ", mean((ŷ .- ytest) .^ 2))
println("MAE: ", mean(abs.(ŷ .- ytest)))
```

MSE: 0.010348591599606136
MAE: 0.08064975520002537

Dynamic validation (sometimes called walk-forward validation) is more demanding: at each time step, you only use data available up to that point. This mimics how the model would actually be used in production.

```
Random.seed!(42)

T = 200
x = rand(T)
y = 1.0 .+ 2.0 .* x .+ 0.1 .* randn(T)

initial_window = 60
```

```

sqerrs = Float64[]

for t in (initial_window+1):T
    Xtr = hcat(ones(t - 1), x[1:(t-1)])
    ytr = y[1:(t-1)]
    θ = Xtr \ ytr
    ŷt = [1.0, x[t]]' * θ
    push!(sqerrs, (ŷt - y[t])^2)
end

println("Walk-forward MSE: ", mean(sqerrs))

```

Walk-forward MSE: 0.012102241884186694

Note

In some contexts, “static” and “dynamic” validation mean something different: static validation checks whether the model reproduces time-zero prices or balances, while dynamic validation checks whether projected cashflows match historical trends.

32.2.4.2. Implied Rates

Implied rates are a form of model inversion: given an observed price, what rate would produce that price? If your pricing function and your implied-rate function don’t round-trip consistently, something is wrong.

```

using Zygote

function present_value(rate, cash_flows)
    sum(cf / (1 + rate)^i for (i, cf) in enumerate(cash_flows))
end

function implied_rate(cash_flows, price)
    f(r) = present_value(r, cash_flows) - price
    # Newton's method using autodiff for the derivative
    x = 0.05
    for _ in 1:100
        fx = f(x)
        abs(fx) < 1e-6 && return x
        x -= fx / gradient(f, x)[1]
    end
    return NaN # didn't converge
end

cash_flows = [100, 100, 100, 100, 1100]
prices = [950, 1000, 1050]

for price in prices
    r = implied_rate(cash_flows, price)
    println("Price $price → rate $(round(r*100, digits=2))%")
end

```

Price 950 → rate 11.37%

32. More Useful Techniques

Price 1000 → rate 10.0%

Price 1050 → rate 8.72%

💡 Tip

JuliaActuary's FinanceCore.jl provides a robust `irr` function that handles edge cases better than a hand-rolled Newton's method.

32.2.5. Predictive vs. Explanatory Models

Models serve different masters. A *predictive* model needs to forecast accurately; an *explanatory* model needs to tell a coherent story about why things happen. The validation approach should match the purpose.

For prediction, pick a loss function that matches how the forecast will be used. If you're forecasting claims payments, RMSE or MAE make sense. If you're estimating Value-at-Risk, use a quantile loss that rewards accurate tail placement. If you're producing full distributions, consider the Brier score or CRPS.

For explanation, the bar is different. Coefficients should have sensible signs and magnitudes—a lapse elasticity of -0.3 per 100 bps rate change is something you can discuss with product actuaries. The model should be stable across different time periods, and it should remain plausible under counterfactual scenarios (“what if we changed surrender charges?”).

💡 Financial Modeling Pro Tip

Align the loss you optimize with the metric you report. If the risk committee cares about 99th percentile losses, train and evaluate on quantile losses—not just RMSE.

32.2.6. Causal Modeling

Causal modeling addresses an important distinction: most financial models capture correlation, not causation. That's often fine for prediction, but dangerous for “what-if” analysis. If you want to know what happens when you *change* something, you need causal reasoning.

Judea Pearl's work on directed acyclic graphs (DAGs) provides a framework for this. The basic idea: draw arrows between variables to represent direct causal influence, then use the graph to determine what you need to control for (and what you shouldn't).

A few patterns come up repeatedly:

Confounders drive both the treatment and the outcome. Macro growth affects both lending standards and default rates. If you don't account for it, you'll see a spurious relationship between standards and defaults.

Mediators sit on the causal pathway. A capital rule affects lending supply, which affects loan growth. If you control for lending supply, you block part of the effect you're trying to measure.

Colliders are caused by two other variables. Regulation intensity and market stress both affect media coverage. If you control for media coverage, you *create* a spurious correlation between regulation and stress.

This matters because financial regulators and boards increasingly ask “what happens if we do X?” Answering that question requires thinking carefully about causal structure, not just fitting the best predictive model. See Pearl (2009) for more on this topic.

32.2.7. Other Techniques Worth Knowing

A few topics we won't cover in depth but are worth exploring:

Quasi-Monte Carlo uses low-discrepancy sequences (Sobol, Halton) instead of pseudo-random numbers. For high-dimensional integrals like exotic option pricing or nested ALM, this can dramatically reduce variance.

Variance reduction techniques—control variates, antithetic paths, stratification—shrink simulation error without adding more scenarios. Useful when estimating Greeks or tail percentiles.

Scenario reduction algorithms compress thousands of economic scenarios into a representative subset while preserving risk metrics. Kantorovich distance pruning is one approach.

Reverse stress testing inverts the usual question: instead of “what's the loss under scenario X?”, ask “what scenario produces loss Y?” This can surface vulnerabilities that standard stress grids miss.

32.3. Programming Techniques

32.3.1. Serialization

Serialization is important because in most finance workflows, the slow part isn't the regression—it's the data prep, calibration, and scenario generation that come before. If you're running the same expensive calibration every time you tweak something downstream, you're wasting compute and making audits harder.

Serialization lets you checkpoint expensive intermediate results. The question is which format to use:

| Format | Good for | Watch out for |
|----------------------|---------------------------------------|-------------------------------|
| Serialization stdlib | Quick caches, memoization | Breaks across Julia versions |
| JLD2 | Persisting results across sessions | Still Julia-specific |
| Arrow/Parquet | Large tables, cross-language sharing | Not for arbitrary Julia types |
| CSV/JSON/TOML | Configs, small tables, human-readable | Slow, lossy for binary data |

Here's a pattern for saving model state with atomic writes (so you don't end up with half-written files if something crashes):

```
using Dates, Serialization

struct ModelState
    θ::Vector{Float64}           # fitted parameters (example)
    seed::Int64                  # RNG seed used for the run
    timestamp::DateTime          # when the snapshot was created
    note::String                  # short description
end

# Atomic write to avoid half-written files
function atomic_serialize(path::AbstractString, obj)
```

32. More Useful Techniques

```
dir = dirname(path)
makedirs(dir)
tmp = tempfile(dir)
serialize(tmp, obj)
mv(tmp, path; force=true)
return path
end

# Example: save/load a state
θ = [1.0, 2.0] # pretend these were estimated
state = ModelState(θ, 42, now(), "OLS on 2025-08-11")

path = joinpath("artifacts", "model_state.jls")
atomic_serialize(path, state)
restored = deserialize(path)

ModelState([1.0, 2.0], 42, DateTime("2026-02-09T18:57:39.820"), "OLS on
↪ 2025-08-11")
```

For cross-session persistence where you might share artifacts with colleagues, JLD2 is more robust.

```
using JLD2, Random, LinearAlgebra, Dates

X = hcat(ones(100), rand(100))
y = X * [1.0, 2.0] .+ 0.1 .* randn(100)
θ = X \ y

meta = (
    julia_version=string(VERSION),
    created_at=string(now()),
    description="OLS fit example",
)
makedirs("artifacts")
jldsave("artifacts/example.jld2"; θ, meta)

θ_loaded, meta_loaded = JLD2.load("artifacts/example.jld2", "θ", "meta")

([0.9770282860377026, 2.059377931238816], (julia_version = "1.12.4", created_at =
↪ "2026-02-09T18:57:40.310", description = "OLS fit example"))
```

The key insight: serialize fitted parameters, calibrated curves, and expensive intermediate results. Don't serialize raw data—keep that in efficient columnar formats and reference it by path (and ideally by content hash) in your artifact metadata. And remember that the CPU is fast enough that in many cases it's faster to compute an answer than it is to retrieve it from memory.

32.3.2. Memoization

Memoization is caching function results keyed by their inputs. For expensive computations that get called repeatedly with the same arguments, this can be a huge win.

```
using SHA, Serialization
```

```

function cachekey(label, args...; kwargs...)
    io = IOBuffer()
    print(io, label, '|', args, '|', kwargs)
    bytes2hex(sha1(take!(io)))
end

function memoize_to_disk(f; label="f", cache_dir="cache")
    mkpath(cache_dir)
    function (args...; kwargs...)
        key = cachekey(label, args...; kwargs...)
        path = joinpath(cache_dir, "$key.jls")
        if isfile(path)
            return deserialize(path)
        end
        result = f(args...; kwargs...)
        tmp = tempfile(cache_dir)
        serialize(tmp, result)
        mv(tmp, path; force=true)
        return result
    end
end

# Wrap an expensive function
ols = (X, y) -> X \ y
ols_cached = memoize_to_disk(ols; label="ols_v1")

# First call computes and caches; second call loads from disk
θa = ols_cached([ones(3) [1.0, 2.0, 3.0]], [1.0, 3.0, 5.0])
θb = ols_cached([ones(3) [1.0, 2.0, 3.0]], [1.0, 3.0, 5.0])
@assert θa == θb

```

 Financial Modeling Pro Tip

For recurring production runs, use a directory convention like `artifacts/ YYYY-MM-DD/` and clean old caches on a schedule. Otherwise disk usage creeps up over time.

32.3.3. Automated Benchmarks

If you have a pricing engine or cash-flow projection that runs nightly, maintain a small set of benchmark portfolios with known expected outputs. Run them automatically and alert if results drift. This catches numerical regressions before they reach production—and gives you confidence when refactoring.

32. More Useful Techniques

Part VIII.

Appendices

33. The Julia Ecosystem for Financial Modeling

CHAPTER AUTHORED BY: ALEC LOUDENBACK

33.1. Chapter Overview

One of Julia's strengths is its rich ecosystem of packages relevant to financial modeling. Rather than building everything from scratch, you can leverage community-maintained libraries that have been tested, optimized, and documented. This chapter provides a tour of packages that financial modelers should know about—many of which appear elsewhere in this book.

33.2. Why Composability Matters

The Julia ecosystem favors composability and interoperability, enabled by multiple dispatch. Because it's easy to automatically specialize functionality based on the type of data being used, there's much less need to bundle a lot of features within a single package.

Julia packages tend to be less vertically integrated because it's easier to pass data around. Contrast this with other ecosystems:

- NumPy-compatible packages in Python are designed to work with a subset of numerically fast libraries
- Pandas includes special functions to read CSV, JSON, database connections, etc.—all bundled together
- The Tidyverse in R has a tightly coupled set of packages that work well together but have limitations with some other R packages

Julia is not perfect in this regard, but it's remarkable how frequently things *just work*. It's not magic—it's the result of language features (particularly multiple dispatch) that make it easy for package developers to write code that composes naturally with other packages.

Julia also has language-level support for documentation, so packages follow a consistent style of help-text that can be auto-generated into web pages available locally or online.

Evaluating Packages

When evaluating packages, look at the GitHub repository: Is it actively maintained? Are issues being addressed? Does it have tests? A package that was last updated three years ago may still work perfectly, but you should understand what you're getting into before building production systems on it.

33.3. Data Handling

Julia offers a rich data ecosystem. At the center sit `CSV.jl` and `DataFrames.jl`.

- **DataFrames.jl:** The standard tabular data structure in Julia, analogous to `pandas` in Python or `data.frame` in R. Essential for working with structured financial data. A mature package with excellent documentation.
- **CSV.jl:** Fast, flexible CSV reading and writing. Handles the messy reality of real-world data files and offers top-class read and write performance.
- **JSON.jl:** Parse and write JSON, the lingua franca of web APIs and configuration files.
- **XLSX.jl:** Read and write Excel files, because spreadsheets aren't going away anytime soon.
- **Arrow.jl:** Interface to the Apache Arrow columnar format, useful for large datasets and interoperability with other languages.
- **Tidier.jl:** A meta-package providing tidyverse-style data manipulation for those coming from R.
- **ODBC.jl:** Connect to any database (if you have the right drivers installed).

Check out the JuliaData organization for more packages and information.

33.4. Data Structures and Utilities

These packages extend Julia's built-in data structures and iteration patterns:

- **DataStructures.jl:** Additional data structures including heaps, queues, deques, sorted containers, and more.
- **StructArrays.jl:** Arrays of structs stored as structs of arrays—provides better memory layout and performance for tabular-like data while maintaining convenient access patterns.
- **Accessors.jl:** Functional lenses for updating immutable data structures. Particularly useful when working with nested configurations or model parameters.
- **IterTools.jl:** Extended iteration utilities including partitioning, grouping, and combining iterators.

33.5. Time and Dates

- **Dates** (standard library): Julia's built-in date and time handling, which is surprisingly capable for most financial applications. Straightforward and robust.
- **BusinessDays.jl:** Holiday calendars and business day calculations—essential for anything involving settlement dates, coupon schedules, or trading calendars.
- **DayCounts.jl:** Day count conventions (30/360, ACT/ACT, etc.) used in fixed income calculations.

33.6. Statistics and Probability

Julia has first-class support for `missing` values, which follows the rules of three-valued logic, so other packages don't need to do anything special to incorporate missing values.

- **Distributions.jl:** A comprehensive library of probability distributions with a consistent interface. Fitting, sampling, and computing densities for dozens of distributions. Essential for any probabilistic work.
- **StatsBase.jl:** Basic statistical functions beyond what's in the standard library.
- **HypothesisTests.jl:** Statistical hypothesis testing, if you need to compute p-values.

- **Turing.jl:** Probabilistic programming and Bayesian inference. Outstanding in its combination of clear model syntax with performance. Described in detail in Chapter 14.
- **TimeSeries.jl:** Time series data structures and analysis.
- **GLM.jl:** Generalized linear models, mimicking R's `glm` functionality.
- **LsqFit.jl:** Fitting data to non-linear models.
- **MultivariateStats.jl:** Multivariate statistics, including PCA.

You can find more packages at JuliaStats.

33.7. Optimization

- **Optimization.jl:** A unified interface to dozens of optimization backends. This is the recommended starting point for most optimization problems—it lets you switch between solvers (`Optim.jl`, `NLOpt`, `Ipopt`, etc.) without changing your code.
- **JuMP.jl:** A modeling language for mathematical optimization (linear, quadratic, mixed-integer, nonlinear). If you've used AMPL, GAMS, or similar tools, JuMP provides comparable expressiveness with Julia's performance.
- **Optim.jl:** Unconstrained and box-constrained optimization algorithms.
- **BlackBoxOptim.jl:** Derivative-free optimization for when you can't (or don't want to) compute gradients.
- **Roots.jl:** Root-finding algorithms for scalar functions—useful for yield-to-maturity calculations, implied volatility, and similar inversions.

33.8. Automatic Differentiation

Sensitivity testing is very common in financial workflows: understanding the change in one variable in relation to another. In other words, the derivative. Julia has unique capabilities where almost across the entire language and ecosystem, you can take the derivative of entire functions or scripts—not by finite differences, but through automatic application of the chain rule.

Key packages include:

- **ForwardDiff.jl:** Forward-mode automatic differentiation. Fast when you have few inputs and many outputs.
- **Mooncake.jl:** A newer reverse-mode AD package with strong performance characteristics.
- **Zygote.jl:** Reverse-mode AD, useful when you have many inputs and few outputs (like training neural networks or computing Greeks).
- **Enzyme.jl:** High-performance AD that works at the LLVM level, supporting mutation and other patterns that trip up source-to-source AD.
- **DifferentiationInterface.jl:** A unified API for multiple AD backends, so you can switch implementations without rewriting code.

Automatic differentiation has uses in optimization, machine learning, sensitivity testing, and risk analysis. See Chapter 16 for a deeper treatment of automatic differentiation with worked examples, and the JuliaDiff organization for more.

33.9. JuliaActuary

The JuliaActuary organization maintains a suite of packages for actuarial and financial calculations. These packages are designed to work together and integrate well with the broader Julia ecosystem.

33. The Julia Ecosystem for Financial Modeling

- **MortalityTables.jl:** Work with standard mortality tables from mort.SOA.org and parametric mortality models. Provides common survival calculations and makes it easy to load, manipulate, and extend mortality assumptions.
- **LifeContingencies.jl:** Calculations for life insurance and annuities, including actuarial present values, net premiums, and reserves. Builds on MortalityTables.jl for mortality assumptions.
- **ActuaryUtilities.jl:** Robust and fast calculations for `internal_rate_of_return`, `duration`, `convexity`, `present_value`, `breakeven`, and more. Useful well beyond actuarial work—these are standard financial math functions.
- **FinanceModels.jl:** Composable contracts, models, and yield curves. Allows modeling of both simple and complex financial instruments with a consistent interface for discounting and valuation.
- **ExperienceAnalysis.jl:** Exposure calculations for actuarial experience studies. Handles the details of calculating exposure periods, handling policy anniversaries, and producing the data needed for A/E analysis.
- **EconomicScenarioGenerators.jl:** Interest rate and equity scenario generation for stochastic modeling. Integrates with FinanceModels.jl for consistent discounting across generated scenarios.

33.10. Visualization

- **Makie.jl:** A powerful, flexible plotting ecosystem. CairoMakie produces publication-quality static figures; GLMakie provides GPU-accelerated interactive 3D graphics. Can create beautiful visualizations.
- **Plots.jl:** A meta-package providing a consistent interface to several plotting backends, depending on whether you're emphasizing interactivity on the web or print-quality output. You can easily add animations or change almost any feature of a plot.
- **AlgebraOfGraphics.jl:** Grammar-of-graphics style plotting built on Makie, similar to ggplot2 in R.
- **StatsPlots.jl:** Extends Plots.jl with a focus on data visualization and compatibility with DataFrames.

33.11. Numerical Computing

- **LinearAlgebra** (standard library): BLAS and LAPACK bindings for matrix operations, eigenvalue decomposition, and linear system solving.
- **DifferentialEquations.jl:** A comprehensive suite for solving ordinary, stochastic, and partial differential equations. Useful for option pricing (Black-Scholes PDE), interest rate models (CIR, Vasicek), and other dynamic systems.
- **Interpolations.jl:** Interpolation methods for curves and surfaces—essential for yield curve construction and volatility surfaces.
- **QuadGK.jl:** Adaptive Gauss-Kronrod numerical integration.

33.12. Parallel and Concurrent Computing

Julia has built-in support for multithreading and distributed computing, but these packages provide higher-level abstractions that make parallel code easier to write and reason about:

- **OhMyThreads.jl**: Simple, composable multithreading. Provides `tmap`, `tforeach`, `treduce` and other parallel primitives that are easier to use correctly than raw `@threads`.
- **Dagger.jl**: Task-based parallelism with automatic scheduling. Define your computation as a graph of dependent tasks and let Dagger figure out how to execute them efficiently across threads or machines.
- **Transducers.jl**: Composable data transformations that can be executed sequentially, in parallel, or distributed. The same transformation pipeline works whether you're processing a small array or a massive distributed dataset.

33.13. Machine Learning

- **Flux.jl**: Neural networks in pure Julia. Flexible and composable. Written entirely in Julia, so it's easier to adapt or see what's going on in the entire stack.
- **MLJ.jl**: A machine learning framework providing a consistent interface to many algorithms, similar to scikit-learn.
- **DecisionTree.jl**: Decision trees and random forests.

One advantage for users is that Julia packages are written in Julia, so automatic differentiation can work across the entire stack. In contrast, PyTorch and TensorFlow are built primarily with C++.

33.14. Developer Utilities

There are also a lot of quality-of-life packages for development:

- **Revise.jl**: Edit code on the fly without needing to re-run entire scripts. Essential for interactive development.
- **BenchmarkTools.jl**: Makes it incredibly easy to benchmark your code—simply add `@benchmark` in front of what you want to test.
- **JET.jl**: Static analysis for Julia code. Catches type errors, unbound variables, and other issues before runtime.
- **Cthulhu.jl**: Interactive exploration of type inference and code generation. When you need to understand why your code is slow, Cthulhu lets you descend into the compiler's view of your functions.
- **Infiltrator.jl**: Drop into an interactive REPL at any point in your code with `@infiltrate`. Useful for debugging complex control flow.
- **Julia VS Code Extension**: The primary IDE experience for Julia development. Provides integrated REPL, debugging, profiling, plot viewing, and documentation lookup.
- **Test** (standard library): Built-in package for performing test sets.
- **Documenter.jl**: Build high-quality documentation based on inline documentation.
- **ClipData.jl**: Copy and paste from spreadsheets to Julia sessions.

33.15. Finding More Packages

This is not an exhaustive list—the Julia ecosystem grows steadily, and new packages appear regularly. Key organizations for financial work include:

33. The Julia Ecosystem for Financial Modeling

- **JuliaActuary:** Actuarial science and financial modeling packages
- **JuliaMath:** Mathematical computing
- **JuliaStats:** Statistics
- **SciML:** Scientific machine learning and differential equations

Julia is a general-purpose language, so you'll also find packages for web development, graphics, game development, audio production, and much more. You can explore packages (and their dependencies) at JuliaHub.

References

- Balbás, Alejandro, José Garrido, and Silvia Mayoral. 2009. "Properties of Distortion Risk Measures." *Methodology and Computing in Applied Probability* 11 (3): 385–99. <https://doi.org/10.1007/s11009-008-9089-z>.
- Brown University CS 0300 Course Staff. 2022. "Lab 4: Caching." 2022. <https://cs.brown.edu/courses/csci0300/2022/assign/labs/lab4.html>.
- Buscemi, Alessio. 2023. "A Comparative Study of Code Generation Using ChatGPT 3.5 Across 10 Programming Languages." <https://arxiv.org/abs/2308.04477>.
- Center, Netherlands eScience. 2019. "Automatic Differentiation from Scratch." 2019. <https://blog.esciencecenter.nl/automatic-differentiation-from-scratch-23d50c699555>.
- Duracz, Jan. 2009. "Derivation of Probability Distributions for Risk Assessment." https://www.researchgate.net/publication/239752412_Derivation_of_Probability_Distributions_for_Risk_Assessment.
- ENCCS. 2023. "Julia for High-Performance Computing." 2023. <https://enccs.github.io/julia-for-hpc/>.
- Hardy, Mary R. 2006. "An Introduction to Risk Measures for Actuarial Applications." *SOA Syllabus Study Note* 19.
- Heer, Jeffrey, Michael Bostock, and Vadim Ogievetsky. 2010. "A Tour Through the Visualization Zoo." *Communications of the ACM*. <https://homes.cs.washington.edu/~jheer/files/zoo/>.
- Julia Documentation. 2024. "Parallel Computing." 2024. <https://docs.julialang.org/en/v1/manual/parallel-computing/>.
- Knuth, Donald E. 1984. "Literate Programming." *Comput. J.* 27 (2): 97–111. <https://doi.org/10.1093/comjnl/27.2.97>.
- Leemis, Lawrence M, and Jacquelyn T McQueston. 2008. "Univariate Distribution Relationships." *The American Statistician* 62 (1): 45–53. <https://doi.org/10.1198/000313008x270448>.
- Lewis, N D. 2013. *100 Statistical Tests*. Createspace.
- MacKay, David J. C. 2003. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press. <https://www.inference.org.uk/mackay/itila/>.
- Matthews, Dylan. 2014. "The Ice Bucket Challenge, and Why We Give to Charity." 2014. <https://web.archive.org/web/20140823152725/https://www.vox.com/2014/8/20/6040435/als-ice-bucket-challenge-and-why-we-give-to-charity-donate>.
- Morey, Richard D., Rink Hoekstra, Jeffrey N. Rouder, Michael D. Lee, and Eric-Jan Wagenmakers. 2016. "The Fallacy of Placing Confidence in Confidence Intervals." *Psychonomic Bulletin & Review* 23 (1): 103–23. <https://doi.org/10.3758/s13423-015-0947-8>.
- Murphy, Hannah, and Cristina Criddle. 2024. "Meta AI Chief Says Large Language Models Will Not Reach Human Intelligence." <https://www.ft.com/content/23fab126-f1d3-4add-a457-207a25730ad9>.
- Pearl, Judea. 2009. *Causality: Models, Reasoning, and Inference*. 2nd ed. New York: Cambridge University Press.
- Rackauckas, Chris. 2020a. "Forward-Mode Automatic Differentiation (AD) via High Dimensional Algebras." 2020. [https://book.sciml.ai/notes/08-Forward-Mode_Automatic_Differentiation_\(AD\)_via_High_Dimensional_Algebras/](https://book.sciml.ai/notes/08-Forward-Mode_Automatic_Differentiation_(AD)_via_High_Dimensional_Algebras/).
- . 2020b. "The Different Flavors of Parallelism." 2020. https://book.sciml.ai/notes/06-The_Different_Flavors_of_Parallelism/.
- Ryle, Gilbert. 1949. *The Concept of Mind*. Harmondsworth, England: Penguin.
- Schwarz, C. J. 2016. "A Short Tour of Bad Graphs." Online. <http://www.stat.sfu.ca/~cschwarz/posters/1999/absenteeism.pdf>.

References

- Shen, Judy Hanwen, and Alex Tamkin. 2026. "How AI Impacts Skill Formation." <https://arxiv.org/abs/2601.20245>.
- Tufte, Edward. 2001. *The Visual Display of Quantitative Information*. 2nd ed. Graphics Press.
- Wang, Shaun S. 2002. "A Universal Framework for Pricing Financial and Insurance Risks." *ASTIN Bulletin* 32 (2): 213–34. <https://doi.org/10.2143/AST.32.2.1027>.

Index

Index

- @code_warntype, 409
- @enter, 394
- @inbounds, 175
- 10x modeler, 17
- AArch64, 158
- abstract type, 90
- abstraction
 - ladder of, 49
- Accessors.jl, 524
- accumulate (function), 113
- ActuaryUtilities.jl, 526
- agent-based models, 43
- aleatory uncertainty, 285
- AlgebraOfGraphics.jl, 526
- algorithm, 232
- Amdahl's Law, 177
- anonymous function, 96
- Anscombe's Quartet, 337
- antithetic variates, 295
- API (Application Programming Interface),
 - 144, 216
- Aqua.jl, 400
- argmax, 317
- argmin, 317
- ARM, 158
- array, 80, 241
 - array-based programming, 190
 - array-oriented programming, 119
- Arrow.jl, 524
- artifacts, 229
- artificial intelligence, 13
- Artificial Intelligence (AI), 16
- ASCII, 84
- asset liability management (ALM), 471
- assumptions, 37
- AST (Abstract Syntax Tree), 159
- atomics, 188
- automatic differentiation, 306
- automation, 15
- AVX-512, 181
- AVX2, 181
- backpropagation, 313
- bar chart, 343
- Bayes factor, 259
- Bayes' Rule, 258
- Bayesian inference, 499
- Bayesian networks, 43
- Bayesian optimization, 329
- Bayesian statistics, 265
- BenchmarkTools.jl, 406, 527
- Beta distribution, 502
- BFGS algorithm, 319
- bias-variance tradeoff, 363
- Big-O notation, 234
- Binomial distribution, 502
- bit, 76
- Black-Litterman model, 455
- Black-Scholes-Merton formula, 26
- BlackBoxOptim.jl, 525
- boolean, 70
- boxplot, 349
- branch (Git), 218
- branch prediction, 162
- branch-and-bound, 237
- breadth-first search (BFS), 245
- breakpoint, 393
- broadcasting, 24, 97
- Brownian motion, 288
- BusinessDays.jl, 524
- byte, 76
- cache line, 156
- cache locality, 163
- CairoMakie.jl, 349
- calibration, 39
- callable struct, 472
- cashflows, 38
- causal modeling, 516
- Central Limit Theorem, 257
- chain rule, 306
- change management, 60
- channels, 183
- character, 84
- choropleth maps, 347
- ChunkSplitters.jl, 494
- CISC, 158
- ClipData.jl, 527
- code coverage, 399
- coding, 68
- colliders, 516

INDEX

- column-major, 163
- commit, 217
- compiler, 158
- composability, 142
- composition over inheritance, 137
- computational complexity, 232
- computer science, 67
- concrete type, 90
- conditional, 70
- Conditional Tail Expectation (CTE), 292
- confounders, 516
- constant time, 233
- constrained optimization, 330
- constructor, 91
- content hash, 225
- continuous integration (CI), 223, 398
- control variates, 295
- conventional commits, 220
- convexity, 479
- Copernican model, 40
- copulas, 438
- cosine similarity, 446
- covariance matrix, 451
- Cox-Ingersoll-Ross (CIR) model, 427
- CPU, 156
- CPU cache, 156
- credibility theory, 499
- cross-validation, 362
- CSV.jl, 524
- Cthulhu.jl, 527
- CUDA, 190
- CUDA.jl, 418
- CUSIP, 138
- DAG (directed acyclic graph), 218
- Dagger.jl, 527
- data drift, 59
- data science, 15, 39
- data structures, 241
- data type, 76
- data version control, 223
- data-oriented design, 140
- DataFrames.jl, 524
- DataStructures.jl, 524
- Dates, 524
- DayCounts.jl, 524
- Debugger.jl, 394
- decision tree, 251
- decision trees, 43
- DecisionTree.jl, 527
- default arguments, 96
- deferred annuity, 473
- dependency management, 30
- depth-first search (DFS), 245
- determinant, 352
- dictionary, 86, 244
- DifferentialEquations.jl, 526
- DifferentiationInterface.jl, 314, 471, 525
- dispatch, 128
- Distributions.jl, 524
- docsites, 216
- docstrings, 215
- documentation, 52
- Documenter.jl, 401, 527
- domain expertise, 50
- domain-specific language, 149
- DomainError, 390
- dual numbers, 307
- duration, 478
 - effective, 478
 - key rate, 479
- econometric models, 43
- EconomicScenarioGenerators.jl, 526
- effective sample size (ESS), 277
- efficient frontier, 454
- eigenvalue decomposition, 356
- eigenvalues, 356
- eigenvector, 356
- empirical models, 43
- encapsulation, 142
- entropy, 250
- enum, 490
- environment stacking, 377
- Enzyme.jl, 314, 525
- epistemic uncertainty, 285
- equality
 - egal, 73
 - equal, 73
- Euclidean distance, 445
- evolutionary algorithm, 326
- Excel, 8
- ExperienceAnalysis.jl, 526
- explanatory models, 40
- exploratory data analysis (EDA), 362
- Exponential distribution, 254
- expression, 69
- expression problem, 132
- expressiveness, 22
- FAST (Fourier Amplitude Sensitivity Tests), 468
- feature engineering, 58, 362
- feature scaling, 443
- Fibonacci sequence, 122
- filter (function), 116
- FinanceModels.jl, 144, 526
- financial model

- definition, 38
- financial modeling, 7
- craft, 47
- finite differences, 465
- finite differentiation, 303
- first-class function, 97
- flame graph, 407
- floating-point, 76
- `Flux.jl`, 527
- forecasting, 364
- formal verification, 246
- forward mode (automatic differentiation), 313
- `ForwardDiff.jl`, 314, 525
- Frequentist statistics, 265
- function, 92
- functional programming, 109
- fuzzing, 247
- garbage collector, 157, 405
- garbage in, garbage out, 58
- GARCH, 436
- generalization, 142
- Geometric Brownian Motion (GBM), 434
- Git, 217
- GitHub, 14
- GitHub Actions, 223, 398
- `GLM.jl`, 525
- `GlobalSensitivity.jl`, 466
- GPU, 189
- GPU programming, 418
- gradient, 311
- gradient-based optimization, 317
- graph (data structure), 244
- Greeks (finance), 311
- Hamming distance, 446
- hash, 243
- hash function, 243
- heap, 157
- heap allocation, 405
- heatmaps, 344
- Hessian, 311
- histogram, 343
- homoiconicity, 148
- Hull-White model, 431
- hyper-parameters, 507
- hyperparameters, 363
- `HypothesisTests.jl`, 524
- IEEE 754, 78
- imperative programming, 106
- `Infiltrator.jl`, 393, 527
- information content, 250
- information gain, 252
- information theory, 249
- inheritance, 137
- inner source, 18
- integer, 76
- integer overflow, 394
- integer programming, 331
- integration testing, 213
- interface, 143
- interoperability, 31
- `Interpolations.jl`, 526
- ISA (Instruction Set Architecture), 158
- iterator, 106
- `IterTools.jl`, 524
- Jaccard similarity, 446
- Jacobian, 311
- `JET.jl`, 400, 527
- `JLD2.jl`, 518
- `JSON.jl`, 524
- Julia, 21
- Julia VS Code Extension, 527
- JuliaActuary, 525, 528
- `JuliaFormatter.jl`, 400
- JuliaHub, 29
- JuliaMath, 528
- JuliaStats, 528
- `JuMP.jl`, 334, 452, 525
- Jupyter notebooks, 28
- just-in-time compilation, 29
- k-nearest neighbor (kNN), 447
- `KDTree`, 447
- kernels (GPU), 191
- key rate duration, 479
- keyword argument, 96
- Large Language Models (LLMs), 16
- LaTeX, 11
- lazy evaluation, 107
- `LifeContingencies.jl`, 526
- line chart, 343
- linear algebra, 351
- linear programming, 330
- linear regression, 43
- linear time, 233
- `LinearAlgebra`, 526
- linked list, 242
- Lisp, 119
- literate programming, 375
- `Literate.jl`, 401
- `LiveServer.jl`, 401
- LLVM, 159
- `LocalRegistry.jl`, 403
- locks, 188
- logging, 390

INDEX

logistic regression, 43
loop, 75
 for, 75
 while, 75
loss function, 317
`LsqFit.jl`, 525

machine instructions, 159
machine learning, 43
macro, 148
Makeham-Beard model, 504
`Makie.jl`, 349, 526
Manhattan distance, 445
`Manifest.toml`, 379
map (function), 112
MAP (maximum a posteriori), 280
map-reduce, 199
mapreduce (function), 116
Markov chain Monte Carlo (MCMC), 499
Markov property, 287
martingale, 288
matrix, 82
matrix factorization, 358
matrix inversion, 355
matrix multiplication, 354
maximum entropy distribution, 254
maximum likelihood estimation (MLE),
 262
MCMC (Markov Chain Monte-Carlo),
 269
mean reversion, 426
mean-variance optimization, 452
mediators, 516
Meltdown, 162
memoization, 518
memory bottleneck, 155
memory hierarchy, 156
memory profiling, 407
Mersenne Twister, 299
Metal, 190
metaprogramming, 148
method, 93
`MethodError`, 390
Metropolis-Hastings, 269
missing, 87
`MLJ.jl`, 527
model
 definition, 37
model architecture, 56
model inversion, 515
model theory, 50
model validation, 61, 514
modern portfolio theory, 451
modular design, 56

module, 103
Monte Carlo methods, 284
Monte Carlo simulation, 43, 489
`Mooncake.jl`, 525
Morris method, 467
`MortalityTables.jl`, 492, 526
multi-device computing, 196
multi-level model, 507
multi-threading, 494
multiple dispatch, 23, 130
`MultivariateStats.jl`, 525
mutability, 74

named tuple, 86
namespace, 103
Naur, Peter, 50
Nelder-Mead method, 322
NEON, 181
neural networks, 43
Newton's method, 319
norm (matrix), 353
Normal distribution, 254
nothing, 87
`NUTS` (No-U-Turn Sampler), 274, 503

object-oriented programming, 131
objective function, 317
`ODBC.jl`, 524
`OhMyThreads.jl`, 416, 527
one-hot encoding, 443
open source, 18
operator
 binary, 93
 unary, 93
`Optim.jl`, 525
optimal point, 317
`Optimization.jl`, 334, 525
overfitting, 43, 363

p-value, 260
package, 103
package manager, 30
`Pandoc`, 11
parallelism, 162
parametric type, 86
particle swarm optimization (PSO), 324
pass-by-sharing, 100
peer review, 61
`Pkg.jl`, 373
`PkgTemplates.jl`, 382
`Plots.jl`, 349, 526
`Pluto.jl`, 375
Poisson process, 289
polymorphism, 126
posterior, 269

- posterior (Bayesian), 506
- precompilation, 413
- predictive models, 40
- present value, 38, 116
- principal component analysis (PCA), 360
- prior, 250
- prior (Bayesian), 505
- probabilistic models, 43
- procedural programming, 106
- process, 184
- profiling, 407
- programming, 7, 68
- `Project.toml`, 377
- property-based testing, 247
- pseudo-random number generator (PRNG), 298
- Ptolemaic model, 40
- pull request, 221
- `PythonCall.jl`, 31
-
- `QuadGK.jl`, 526
- quadratic time, 233
- quantitative analysts, 14
- Quarto, 11, 376
- quasi-Monte Carlo, 296
- quasi-Newton methods, 319
-
- R-hat, 277
- race condition, 186
- RAM, 156
- random forests, 43
- random walk, 255
- range, 83
- `RCall.jl`, 31
- real-world scenarios, 425
- recursion, 122
- REDCAPE framework, 37
- reduce (function), 115
- registers, 156
- registry, 226
- regression testing, 213
- reinforcement learning, 362
- REPL, 372
- replicating portfolio, 42
- reverse mode (automatic differentiation), 313
- reverse stress testing, 469
- `Revise.jl`, 375, 527
- RISC, 158
- risk governance, 18, 60
- risk measures, 290
- risk parity, 457
- risk-neutral scenarios, 425
- robust optimization, 459
-
- ROCM, 190
- root finding, 318
- `Roots.jl`, 525
- `Runic.jl`, 400
-
- scatter plot, 343
- scenario analysis, 38
- scenario generation, 425
- `SciML`, 528
- scope, 101
 - global, 101
 - local, 101
- semantic versioning, 227, 378
- sensitivity analysis, 463
 - global methods, 463
 - local methods, 463
- separation of concerns, 142
- serialization, 517
- `SharedArrays`, 201
- Sharpe ratio, 458
- short-rate models, 426
- SIMD, 180, 418
- similarity analysis, 443
- simulated annealing, 322
- simulation models, 43
- singular value decomposition (SVD), 357
- Sklar's theorem, 438
- small world vs large world, 38
- `SnoopCompile.jl`, 413
- Sobol indices, 466
- space complexity, 236
- `Spectre`, 162
- speculative execution, 162
- SSD, 156
- `StableRNGs.jl`, 403
- stack, 157
- stacktrace, 390
- staging (Git), 220
- startup file, 383
- state, 106
- state space, 285
- `StaticArrays.jl`, 173, 419
- stationarity, 287
- statistical modeling, 39
- `StatsBase.jl`, 524
- `StatsPlots.jl`, 526
- stochastic differential equations (SDEs), 288
- stratified sampling, 296
- string, 84
- struct, 88
- `StructArrays.jl`, 524
- supervised learning, 362
- surrender rate, 473

INDEX

- SVE (Scalable Vector Extension), 181
- syntax, 22
- sysimage, 381
- t-SNE, 346
- tangency portfolio, 459
- task-based parallelism, 202
- tasks, 182
- term life insurance, 489
- Test, 527
- test coverage, 212
- test-driven development (TDD), 211
- testing, 210
- thread, 185
- thread safety, 494
- Tidier.jl, 524
- time series forecasting, 43
- time-to-first-plot, 29
- TimeSeries.jl, 525
- TPU, 189
- trace (matrix), 353
- Transducers.jl, 527
- transistors, 158
- transpose, 352
- tree (data structure), 245
- tuple, 85
- Turing.jl, 274, 499, 525
- two culture problem, 22
- two language problem, 22
- type hierarchy, 79, 126
- type inference, 405
- type stability, 173, 405
- type system, 29
- uncertainty
 - aleatory, 54
 - epistemic, 54
- underfitting, 363
- Unicode, 84, 90
- union type, 88
- unit testing, 213
- unsupervised learning, 362
- valuation, 38
- Value at Risk (VaR), 292
- variable, 69
- variance reduction, 295
- Vasicek model, 427
- vector, 82
- vectorization, 180
- version control, 14, 56, 217
- violin plot, 349
- Visual Studio Code, 374
- walk-forward validation, 514
- Wiener process, 288
- x86-64, 158
- XLSX.jl, 524
- Xorshift, 299
- Xoshiro, 299
- yield curve, 144, 471
- Yields.jl, 146
- Zygote.jl, 314, 525