

Computational Thinking for Actuaries and Financial Professionals

With Applications in Julia

Alec Loudenback and Yun-Tien Lee

2024-12-16

Table of contents

Preface	1
1. Draft of Cover	3
I. Introduction	5
The approach	8
What you will learn	8
The Journey Ahead	9
Prerequisites	10
The Contents of This Book	10
Notes on formatting	11
Colophon	12
2. Why Program?	15
2.1. In this Chapter	15
2.2. The Long View	15
2.3. What's coding got to do with this?	17
2.4. The 10x Actuary	18
2.5. Risk Governance	19
2.6. Managing and Leading the Transformation	20
2.7. Outlook	20
3. Why use Julia?	23
3.1. Expressiveness and Syntax	24
3.1.1. Example: Retention Analysis	24
3.1.2. Example: Random Sampling	26
3.2. The Speed	27
3.3. More of Julia's benefits	28
3.4. The Tradeoff	29
3.5. Package Ecosystem	30
3.6. Conclusion	31

Table of contents

II. Conceptual Foundations: Modeling and Programming	33
4. Elements of Financial Modeling	35
4.1. In this Chapter	35
4.2. What is a model?	35
4.2.1. “Small world” vs “Large world”	36
4.3. What is a <i>Financial Model</i> ?	37
4.4. The Process of Building a Financial Model	38
4.5. Predictive versus Explanatory Models	38
4.5.1. A Historical Example	38
4.5.2. Examples in the Financial Context	41
4.6. What makes a good model?	42
4.6.1. Achieving original purpose	42
4.6.2. Usability	42
4.6.3. Performance	43
4.6.4. Separation of Model Logic and Data	44
4.6.5. Abstraction of Modeled Systems	44
4.7. What makes a good modeler?	45
4.7.1. Domain Expertise	45
4.7.2. Model Theory	46
4.7.3. Curiosity	48
4.7.4. Rigor	49
4.7.5. Clarity	49
4.7.6. Humble	49
4.7.7. Architecture	49
4.7.8. Planning	50
4.7.9. Toolset	50
5. Elements of Programming	51
5.1. In this section	51
5.2. Computer Science, Programming, and Coding .	52
5.3. Expressions and Control Flow	54
5.3.1. Assignment and Variables	54
5.4. Expressions	55
5.4.1. Compound Expression	55
5.4.2. Conditional Expressions	56
5.4.3. Assignment and Variables	59
5.4.4. Loops	60
5.4.5. Performance of loops	62

Table of contents

5.5.	Data Types	62
5.5.1.	Numbers	62
5.5.2.	Type Hierarchy	66
5.5.3.	Collections	66
5.5.4.	Parametric Types	76
5.5.5.	Types for things not there	77
5.5.6.	Union Types	77
5.5.7.	Creating User Defined Types	79
5.5.8.	Mutable structs	82
5.5.9.	Constructors	83
5.6.	Functions	84
5.6.1.	Special Operators	85
5.6.2.	Defining Functions	86
5.6.3.	Defining Methods on Types	87
5.6.4.	Keyword Arguments	89
5.6.5.	Default Arguments	89
5.6.6.	Anonymous Functions	90
5.6.7.	First Class Nature	90
5.6.8.	Broadcasting	91
5.6.9.	Passing by Sharing	96
5.7.	Scope	97
5.7.1.	Modules and Namespaces	99
III.	Conceptual Foundations: Abstractions	103
6.	Functional Abstractions	105
6.1.	In this section	105
6.2.	Introduction	105
6.3.	Imperative Style	106
6.3.1.	Iterators	107
6.4.	Functional Techniques and Terminology	110
6.4.1.	map	112
6.4.2.	accumulate	114
6.4.3.	reduce	116
6.4.4.	mapreduce	117
6.4.5.	filter	118
6.4.6.	More Tips on Functional Styles	120
6.5.	Array-Oriented Styles	122
6.6.	Recursion	124

Table of contents

7. Data and Types	127
7.1. In this section	127
7.2. Using Types to Value a Portfolio	127
7.3. Benefits of Using Types	128
7.4. Defining Types for Portfolio Valuation	129
7.4.1. (Multiple) Dispatch	131
7.5. Objected Oriented Design	135
7.6. Assigning Behavior	135
7.7. Inheritance	136
7.7.1. Composition over Inheritance	137
8. Higher Levels of Abstraction	139
8.1. In this section	139
8.2. Introduction	139
8.3. Principles for Abstraction	140
8.3.1. Pragmatic Considerations for Model Design	142
8.4. Interfaces	143
8.4.1. Defining Good Interfaces	144
8.4.2. Interfaces: A Financial Modeling Case Study	145
8.5. Macros & Homoiconicity	150
8.5.1. Metaprogramming in Financial Modeling .	151
8.5.2. Commonly Encountered Macros	152
IV. Conceptual Foundations: Learning from Related Disciplines	155
9. Hardware and Its Implications	157
9.1. In this section	157
10. Elements of Computer Science	159
10.1. In this section	159
10.2. Computer Science for Financial Professionals .	159
10.3. Algorithms & Complexity	160
10.3.1. Computational Complexity	161
10.3.2. Expected versus worst-case complexity .	166
10.3.3. Complexity: Takeaways	167
10.4. Data Structures	167
10.4.1. Arrays	167

Table of contents

10.4.2. Linked Lists	168
10.4.3. Records/Structs	169
10.4.4. Dictionaries (Hash Tables)	170
10.4.5. Graphs	172
10.4.6. Trees	172
10.4.7. Data Structures Conclusion	174
10.5. Formal Verification	174
10.5.1. Basic Concept	174
10.5.2. Formal Verification in Practice	175
10.5.3. Related Topics	175
11. Applying Software Engineering Principles	177
11.1. In this section	177
11.2. Testing	177
12. Statistical Inference and Information Theory	179
12.1. In This Chapter	179
12.2. Information Theory	179
12.2.1. Example: Classificaiton	182
12.2.2. Maximum Entropy Distributions	185
12.3. Bayes' Rule	191
12.3.1. Example: Model Selection via Likelihoods	191
12.4. Modern Bayesian Statistics	199
12.4.1. Background	199
12.4.2. Implications for Financial Modeling . . .	203
12.4.3. Basics of Bayesian Modeling	204
12.4.4. Markov-Chain Monte Carlo	204
12.4.5. MCMC Algorithms	210
12.4.6. Rainfall Example (Continued)	211
12.4.7. Conclusion	223
12.4.8. Further Reading	224
V. Computational Thinking in an Actuarial and Financial Context	225
13. Modeling	227
13.1. In This Chapter	227
13.2. Parsimony	227
14. Automatic Differentiation	229
14.1. In This Chapter	229

Table of contents

14.2. Motivation for (Automatic) Derivatives	229
14.3. Finite Differentiation	229
14.4. Automatic Differentiation	233
14.4.1. Dual Numbers	234
14.5. Performance of Automatic Differentiation	237
14.6. Automatic Differentiation in Practice	239
14.6.1. Performance	241
14.7. Forward Mode and Reverse Mode	242
14.8. Practical tips for Automatic Differentiation	242
14.8.1. Choosing between Reverse Mode and Forward Mode	243
14.8.2. Mutation	243
14.8.3. Custom Rules	243
14.8.4. Available Libraries	243
14.9. References	244
15. Optimization	245
15.1. In This Chapter	245
15.2. Setup	245
15.3. Differentiable programming	247
15.4. Model fitting	249
15.4.1. Root finding	249
15.4.2. Best fitting curve	250
16. Sensitivity Analysis	253
16.1. In This Chapter	253
16.2. Setup	253
16.3. The Data	254
16.4. Common Sensitivity Analysis Methodologies	256
16.4.1. Finite Differences	256
16.4.2. Scenario Analyses	256
16.4.3. Regression Analyses	257
16.4.4. Sobol Indices	257
16.4.5. Morris Method	258
16.4.6. Fourier Amplitude Sensitivity Tests	258
16.5. Benchmarking	258
17. Stochastic Modeling	259
18. Visualizations	261
18.1. In This Chapter	261

Table of contents

19. Matrices and Their Uses	263
19.1. In This Chapter	263
19.2. Setup	263
19.3. Matrix manipulation	263
19.3.1. Multiplication	263
19.3.2. Inversion	264
19.4. Matrix decomposition	264
19.4.1. Eigenvalues	264
19.4.2. Singular values	265
19.4.3. Matrix factorization and fatorization ma- chines	266
19.4.4. Principal component analysis	268
20. Learning from Data	269
20.1. In this chapter	269
20.2. Setup	269
20.3. Applications	270
20.3.1. Parameter fitting	270
20.3.2. Forecasting	270
20.3.3. Static and dynamic validation	270
20.3.4. Implied rate analysis	272
VI. Applications in Practice	275
21. Stochastic Mortality Projections	277
21.1. In This Chapter	277
21.2. Setup	277
21.3. The Data	278
21.4. Running the projection	281
21.4.1. Stochastic Projection	281
21.5. Benchmarking	283
21.6. Further Optimization	284
22. Scenario Generation	285
22.1. In This Chapter	285
22.2. Setup	285
22.3. The Data	285
22.4. Pseudo Random Number Generators	285
22.4.1. Common PRNGs	286
22.4.2. Consistent Interface	287

Table of contents

22.5. Common Economic Scenario Generation Approaches	288
22.5.1. Interest Rate Models	288
22.5.2. Stock Models	293
22.5.3. Copulas	297
22.6. Benchmarking	298
23. Similarity Analysis	299
23.1. In This Chapter	299
23.2. Setup	299
23.3. The Data	299
23.4. Common Similarity Measures	301
23.4.1. Euclidean Distance (L ₂ norm)	301
23.4.2. Manhattan Distance (L ₁ Norm)	302
23.4.3. Cosine Similarity	302
23.4.4. Jaccard Similarity	303
23.4.5. Hamming Distance	303
23.5. k-Nearest Neighbor (kNN) Clustering	304
23.6. Benchmarking	305
24. Portfolio Optimization	307
24.1. In This Chapter	307
24.2. Setup	307
24.3. The Data	307
24.4. Theory	307
24.5. Mathematical tools	308
24.5.1. Mean-variance optimization model	308
24.5.2. Efficient frontier analysis	309
24.5.3. Black-Litterman	310
24.5.4. Risk Parity	312
24.5.5. Sharpe Ratio Maximization	313
24.5.6. Robust Optimization	314
25. Bayesian Mortality Modeling	317
25.1. Generating fake data	317
25.2. 1: A single binomial parameter model	321
25.2.1. Sampling from the posterior	322
25.3. 2. Parametric model	327
25.3.1. Plotting samples from the posterior	329
25.4. 3. Parametric model	334
25.5. Handling non-unit exposures	338

Table of contents

25.6. Predictions	342
26. Other Useful Techniques	345
26.1. In this chapter	345
26.2. Conceptual Techniques	345
26.2.1. Taking things to the Extreme	345
26.2.2. Range Bounding	345
26.3. Modeling Techniques	346
26.3.1. Serialization	346
VII. Appendices	347
27. Set up Julia and the Computing Environment	349
27.1. Installation	349
27.2. Package Management	349
27.3. Editors	350
27.3.1. Visual Studio Code	350
27.3.2. Notebooks	351
27.4. REPL	351
27.4.1. Help Mode	351
28. Environment and Package Management	353
28.1. In This Section	353
28.2. Projects, Manifests, and Dependencies	353
28.2.1. Project.toml	353
28.2.2. Manifest.toml	356
28.2.3. Reproducibility	356
28.3. Environments	357
28.4. Packages	358
28.4.1. Packages versus Projects	358
28.4.2. Basic Package Structure	358
28.4.3. Extension Packages	358
28.5. Registries	358
28.5.1. Local Registries	358
29. The Julia Ecosystem Today	359
29.0.1. Data	360
29.0.2. Plotting	360
29.0.3. Statistics	360
29.0.4. Machine Learning	361
29.0.5. Differentiable Programming	361

Table of contents

29.0.6. Utilities	362
29.0.7. Other packages	363
29.0.8. Actuarial packages	363
30. Debugging and Performance Measurement	365
30.1. Benchmarking	365
References	367

Preface

This book is intended to enable practitioners and advanced students of financial disciplines to utilize the tools, language, and ideas of computational and related sciences in their own work.

1. Draft of Cover

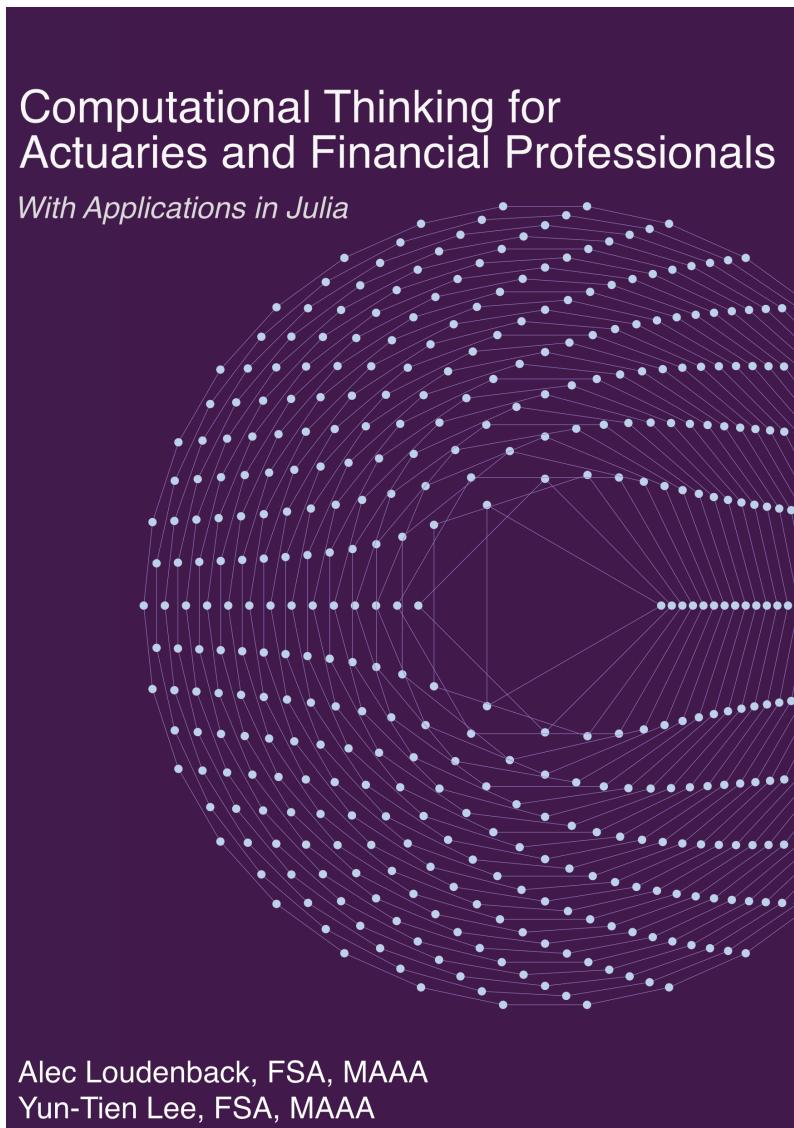


Figure 1.1.: Draft of Cover

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 incredibly complex and variegated. It, along with many of the sciences, is a place where practical goals harness computational tools to arrive at answers that (we hope) are meaningful in a way that tells us more about the world we live in. What this usually means specifically is that practitioners utilize computers to do the heavy work of processing data or running simulations which reveal the something about the complex systems we seek to represent. In this way, then, financial modelers must also be a craftsman who seeks not only to design new products, but must also think carefully about the tools and the process used therein.

This book seeks to aid the practitioner in developing 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 and bring it back up to the most abstract levels, and develop 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 better for the journey we will take.

This book looks at programming for the applied financial professional and we will start by answering a very basic 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 with some basic training in a richer set of tools will build a better house. This book will enhance your toolkit with experience with multiple tools: a specific programming language, yes, but much more than that: a language to talk about solving problems, a deeper understanding of specific problem solving techniques, how to make decisions about what the architecture of a solution looks like, 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 are applicable to a wide variety of related disciplines.

We will pull from examples on both sides of the balance sheet: the left (assets) and right (liabilities). We may also take the liberty to, at times, abuse traditional accounting notions: 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 take examples that use examples of assets (fixed income, equity, derivatives) or 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,

The Journey Ahead

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 gotten to be very proficient *in spite of* not having substantive formal training on problem solving, algorithms, or model architecture. This book serves to fill that gap and provide the “missing semester” (or “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 direct responsibility does not entail hands-on-coding, be it management or “low-code”, the ideas and language should prove useful in guiding 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 others, and there are community support resources 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.

Further, 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 and the trick to encouraging adoption is finding the right place to plug a new idea or suggestion. One of the secondary, but still important, things this book should provide is the language and technical knowledge to partner

with others (such as peers and IT) to make pragmatic decisions about the tradeoffs that will need to be made.

Prerequisites

Basic experience with financial modeling is not strictly required, but it will benefit the reader to be familiar so that the examples will not be attempting to teach both financial maths and computer science simultaneously.

Advanced financial maths (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 either: Chapter 5 introduces the basic syntax and concepts while Chapter 27 covers setting up your environment to follow along. For readers with background in programming, we recommend skimming Chapter 5 and reading in full the sections which have a \square symbol in the margin, which is our way of highlighting Julia-specific content to be aware of.

TODO

Create a venn diagram showing financial modeling at the intersection of statistics, financial math, computer science.

The Contents of This Book

Part 1 of the book addresses the theoretical and technical foundations of programming, as well as the conceptual basis for financial modelling. It familiarizes the readers with key functional programming principles, alongside introducing important aspects of software engineering relevant to financial modelling.

The Contents of This Book

Parts 2 and 3 bridge the gap between theory and practical applications, underlining the features of Julia that make it a robust tool for real-world financial and actuarial contexts. Through a careful exploration of topics like sensitivity analysis, optimization, stochastic modeling, visualization, and practical financial applications, the book demonstrates how Julia's high-level, high-performance programming capabilities can enhance accuracy and efficiency in financial modelling. As an up-and-coming language loved for its speed and simplicity, Julia is ripe for wide adoption in the financial sector. The time for this book is ripe, as it will satiate the growing demand for professionals who want to blend programming skills with financial modelling acumen.

While we have chosen to use Julia for the examples in this book, the vast majority of the concepts presented are not Julia-specific. We will attempt to motivate why Julia works so well as a language for financial modeling but like mathematics and applied mathematics, the concepts are portable even if the numbers (language) changes. Readers are encouraged to follow along the examples on their own computer (see instructions for Julia in Chapter 27) and the entire book is available on GitHub at [#TODO: determine book URL].

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 are to be sent to Pkg mode in the REPL (see Section 28.1), such as 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:

```
(@v1.10) 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 book were rendered using Quarto and Quarto's open source dependencies like PanDoc.

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 Loudeback using Affinity Designer with the graphic used under permission by user cormulion on Github.

This book was rendered on May 19, 2024. The system used to generate the code and benchmarks was:

```
versioninfo()
```

```
Julia Version 1.10.2
Commit bd47eca2c8a (2024-03-01 10:14 UTC)
Build Info:
  Official https://julialang.org/ release
Platform Info:
  OS: macOS (arm64-apple-darwin22.4.0)
  CPU: 8 × Apple M3
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-15.0.7 (ORCJIT, apple-m1)
```

The Contents of This Book

Threads: 4 default, 0 interactive, 2 GC (on 4 virtual cores)

Environment:

JULIA_NUM_THREADS = auto

2. Why Program?

"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)

[Drafting Note: This chapter is pulled from the article published in 2020 and needs to be adapted for the book's audience. Also to include: why not low-code solutions?]

2.1. In this Chapter

We motivate why a financial professional should adopt programming skills which will improve their own capabilities and enjoyment of the discipline, whilst allowing themselves to better themselves and the industry we work in.

2.2. The Long View

It might be odd to say that technology and its use in insurance is on a one-hundred-year cycle, but that seems to be the case.

130 years ago, actuaries crowded into a room at a meeting of the Actuarial Society of America to watch a demonstration that would revolutionize the industry: Herman Hollerith's tabulating punch card machine¹.

For the next half-century, the increasing automation — from tabulating machines to early-adopting mainframes and computers — was a critical competitive differentiator. Companies like Prudential, MetLife, and others partnered with technology companies in the development of hardware and software².

¹ Co-evolution of Information Processing Technology and Use: Interaction Between the Life Insurance and Tabulating Industries

² From Tabulators to Early Computers in the U.S. Life Insurance Industry

2. Why Program?

The dramatic embodiment of this information-driven cycle was portrayed in the infamous Billion Dollar Bubble movie, which showcased the power and abstraction of the computer to commit millions of dollars of fraud by creating and maintaining fake insurance policies.

The movie also starts to hint at the oscillation away from the technological-competitive focus of insurance companies. I argue that the focus on technology was lost over the last 50 years with the rise of Wall Street finance, investment-oriented life insurance, industry consolidation, and the explosion of financial structuring like derivatives, reserve financing, or other advanced forms of reinsurance.

Value-add came from the C-Suite, not from the underlying business processes, operations, and analysis. The result is, e.g., ever-more complicated reinsurance treaties layered into mainframes and admin systems older than most of the actuaries interfacing with them.

The pace of *strategic value-add* isn't slowing, though it must stretch further (in complexity and risk) to find comparable opportunities as the past. Having more agile, data-oriented operations enables companies to be able to react to and implement those opportunities. *Technological value-add* can improve a company's bottom line through lower expenses and higher top-line growth, but often with a more favorable risk profile than some of the "strategic" opportunities.

Today, there is a trend reverting back to technological value-creation and 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 sharing³; Airlines are so dependent on their systems that the skies become quieter on the rare occasion that their computers give way.

Why is it, that companies that are so involved in *things* (cars, shopping) and *physical services* (flights) are so much more focused on improving their technological operations than insurance companies *whose very focus is 'information-based'?* **The market has rewarded those who have prioritized their internal technological solutions.**

³ Have you had your Bezos moment?
What you can learn from Amazon

2.3. What's coding got to do with this?

Commoditized investing services and low yield environments have reduced insurance companies' comparative advantage to "manage money". Yield compression and the explosion of consumer-oriented investment services means a more competitive focus on the ability to manage the entire policy lifecycle efficiently (digitally), perform more real-time analysis of experience and risk management, and handle the 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:

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

Once those two walls are breached, traditional insurance companies without a strong technology core will struggle to keep up. The key to thriving is not just adding "developers" to an organization; it's going to be **getting domain experts like actuaries to be an integral part of the technology transformation**.

2.3. What's coding got to do with this?

Everything. Programming is the optimal way to interact between the computer and actuary — and importantly between computer and computer. Programming is the actionable expression of ideas, math, analysis, and information. Think of programming as the 21st-century leap in the actuary's toolkit, just as spreadsheets were in the preceding 40 years. Versus a spreadsheet-oriented workflow:

- More natural automation of, and between processes
- Better reproducibility
- Scaling to fit any size dataset and workload
- Statistics and machine learning capabilities

2. Why Program?

- Advanced visualizations to garner new views into your data

This list isn't comprehensive and some benefits are subtle — when you are code-oriented instead of spreadsheet-oriented, you tend to want to structure your data in a portable and shareable way. For example, relying more on data warehouses instead of email attachments. This, in turn, enables data discovery and insights that otherwise wouldn't be there. Investing in a code-oriented workflow is playing the long-game.

The actuary of the future needs to have coding as one of their core skills. Already today, the advances of business processes, insurance products, and financial ingenuity are written with lines of code — *not* spreadsheets. Not being able to code *necessarily* means that you are *following* what others are doing today.

It's commonly accepted now that to gather insights from your data, you need to know how to code. Similar to your data, your business architecture, modeling needs, and product peculiarities are often better suited to customized solutions. Why stop at data science when learning how to solve problems with a computer?

2.4. The 10x Actuary

As we swing back to a technological focus, we do not leave the finance-driven complexity behind. The increasingly complex business needs will highlight a large productivity difference between an actuary who can code and one who can't — simply because the former can react, create, synthesize, and model faster than the latter. From the efficiency of transforming administration extracts, summarizing and aggregating valuation output, to analyzing claims data in ways that spreadsheets simply can't handle, you can become a "10x Actuary"⁴.

Flipping switches in a graphical user interface versus being able to *build models* is the difference between having a surface-level familiarity and having full command over the analysis

⁴ The 10x [Rockstar] developer is NOT a myth

2.5. Risk Governance

and the 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 visualize, perform sensitivity analysis, statistics, stochastic analysis, or process automation. Things that, when done programmatically, are often just a few lines of additional code.

Do I advocate dropping the license for your software vendor? No, not yet anyway. But the ability to supplement and break out of the modeling box has been an increasingly important part of most actuaries' work.

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

30-40 years ago, there were no vendor-supplied modeling solutions and so you had no choice but to build models internally. This shifted with the advent of vendor-supplied modeling solutions. Today, it's never been better for companies to leverage open source to support their custom modeling, risk analysis/monitoring, and reporting workflows.

2.5. 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 and approval processes
- Version control, version management, and reproducibility
- Continuous testing and validation of results
- Open and transparent design
- Minimization of manual overrides, intervention, and opportunity for user error

2. Why Program?

- Automated trending analysis, system metrics, and summary statistics
- Continuously updated, integrated, and self-generating documentation
- Integration with other business processes through a formal boundary (e.g. via an API)
- Tools to manage collaboration in parallel and in sequence

2.6. Managing and Leading the Transformation

The ability to understand the concepts, capabilities, challenges, and lingo is not a dichotomy, it's a spectrum. Most actuaries, even at fairly high levels, are still often involved in analytical work. Still above that, it's difficult to lead something that you don't understand.

Conversely, the skill and practice of coding enhances managerial capabilities. When you are really skilled at pulling apart a problem or process into its constituent parts and designing optimal solutions; that's a core attribute of leadership: having the vision of where the organization *should be* instead of thinking about where it is now.

Nor is the skillset described here limiting in any other aspect of career development any more than mathematical ability, project collaboration, or financial acumen — just to name a few.

2.7. Outlook

It will increasingly be essential for companies to modernize to remain competitive. That modernization isn't built with big black-box software packages; it will be with domain experts who can translate the expertise into new forms of analysis - doing it faster and more robustly than the competition.

SpaceX doesn't just hire rocket scientists - they hire rocket scientists who code.

2.7. Outlook

Be an actuary who codes.

3. Why use Julia?

[Drafting Note: This chapter is pulled from the article published in 2021 and needs to be adapted for the book's audience.
]

Julia is relatively new⁵, and *it shows*. It 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 and there's a growing body of online references and 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 modeling⁶. The last of these has a great talk on YouTube by Aviva's Tim Thornham, which 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 not just great for data science — but also modeling, ETL, visualizations, package control/version 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.

⁵ Python first appeared in 1990. R is an implementation of S, which was created in 1976, though depending on when you want 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.

⁶ Aviva Case Study

3. Why use Julia?

3.1. 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.

3.1.1. Example: Retention Analysis

This is a really simple example relating `Cessions`, `Policies`, and `Lives` to do simple retention analysis.

First, let's define our data:

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

struct Policy
    face
    cessions
end

struct Cession
    ceded
end
```

3.1. Expressiveness and Syntax

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
```

And then what retention means for a `Life`:

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

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

We defined `retained` and adapted it to mean related, but different things depending on the specific context. That is, we didn't have to define `retained_life(...)` and `retained_pol(...)` because Julia can be *dispatch* based on what you give it. This is, as some would call it, unreasonably effective.

Let's use the above code in practice then.

The `julia>` syntax indicates that we've moved into Julia's interactive mode (REPL mode):

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

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

julia> retained(pol_1)
400

julia> retained(life)
1900
```

3. Why use Julia?

And for the last trick, something called “broadcasting”, which automatically vectorizes any function you write, no need to write loops or create if statements to handle a single vs repeated case:

```
julia> retained.(life.policies) # retained amount for each policy  
[400 , 1500]
```

3.1.2. Example: Random Sampling

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

We generate 100:

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

Table 3.1.: A comparison of random outcome generation in Julia, R, and Python.

Julia	R	Python
<pre>using Distribution unif(100) rnorm(100)</pre>	<pre>rand(100) rnorm(100)</pre>	<pre>import scipy.stats as sps import numpy as np</pre>

rand(Normal(), 100) sample(c("Preferred", "Standard"),
rand(Bernoulli(0.5), 100, place=TRUE)) ps.uniform.rvs(size=100)
rand(["Preferred", "Standard"], 100) sps.norm.rvs(size=100)
sps.bernoulli.rvs(p=0.5, size=100)
np.random.choice(["Preferred", "S", size=100)

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.

3.2. The Speed

3.2. The Speed

As the journal Nature said, “Come for the Syntax, Stay for the Speed”.

Recall the Solvency II compliance which ran 1000x faster than the prior vendor solution mentioned earlier: 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.

What analysis would you like to do if it took less time? A stochastic analysis of life-level claims? Machine learning with your experience data? Daily valuation instead of quarterly?

Speaking from experience, speed is not just great for production time improvements. During development, it’s really helpful too. When building something, I can see that I messed something up in a couple of seconds instead of 20 minutes. 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 vs R or Python or MATLAB.

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 an example of 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?

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!

3. Why use Julia?

3.3. 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: vectorizable (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 PyCall 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.

3.4. The Tradeoff

For those who are enterprise-minded: in addition to the liberal licensing mentioned above, there are professional products from organizations like Julia Computing that provide hands-on support, training, IT governance solutions, behind-the-firewall package management, and deployment/scaling assistance.

3.4. The Tradeoff

Julia is fast because it's compiled, unlike R and Python where (loosely speaking) the computer just reads one line at a time. Julia compiles code "just-in-time": right before you use a function for the first time, it will take a moment to pre-process the code section for the machine. Subsequent calls don't need to be re-compiled 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, but each 9,999 remaining projections only take 10ms.
- Python runs in 17 minutes: 100ms of a second for each computation.

Typically, the compilation is very fast (milliseconds), but in the most complicated cases it can be several seconds. One of these is the "time-to-first-plot" issue because it's the most common one users encounter: super-flexible plotting libraries have a lot of things to pre-compile. So in the case of plotting, 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. The time-to-first plot is a solvable problem that's receiving a lot of attention from the core developers and will get better with future 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.

3. Why use Julia?

3.5. Package Ecosystem

Using packages as dependencies in your project is assisted by Julia's bundled package manager.

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

Packages can be one of the thousands of publicly available, or private 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 often interop without any additional coordination.

For example, packages that operate on data tables work without issue together 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 want to load a CSV you might not want to transform the data into a dataframe (maybe you want a matrix or a plot instead). To load data into a dataframe, in Julia the practice is to use both the CSV and DataFrames packages, which help separate concerns. Some users may prefer the Python/R approach of less modular but more all-inclusive packages.

3.6. Conclusion

Looking at other great tools like R and Python, it can be difficult to summarize a single reason to motivate a switch to Julia, but hopefully this article piqued an interest to try it for your next project.

That said, Julia shouldn't be the only tool in your tool-kit. SQL will remain an important way to interact with databases. R and Python aren't going anywhere in the short term and will always offer a different perspective on things!

In an earlier article, I talked about becoming a **10x Actuary** which meant being proficient in the language of computers so that you could build and implement great things. In a large way, the choice of tools and paradigms shape your focus. Productivity is one aspect, expressiveness is another, speed one more. There are many reasons to think about what tools you use and trying out different ones is probably the best way to find what works best for you.

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 innovated ways to solve problems. In this way, you reward your intellect while building more meaningful and relevant models and analysis.

Part II.

Conceptual Foundations: Modeling and Programming

4. Elements of Financial Modeling

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

4.1. In this Chapter

We explain what constitutes a financial model and what are common uses of a model. We explain what makes an adept practitioner.

4.2. What is a model?

A **model** represents aspects of the world around us distilled down into simpler, more tractable components. It is impossible to fully capture the everything that may affect the objects of our interest. We may build models for a variety of reasons, as listed in Table 4.1.

Table 4.1.: 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.

4. Elements of Financial Modeling

Use	Description
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 the returns for the stocks in our retirement portfolio. It would be impossible to try to build a model which would capture all of the individual people working jobs and making decisions, weather events that damage property, political machinations, etc. Instead, we try to capture certain fundamental characteristics. For example, it is common to model equity returns as cumulative pluses and minuses from random movements where those movements have certain theoretical or historical characteristics.

Whether we are 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 eight percent return on stocks and conclude that such a return will be sufficient to retire at age 53. Alternatively, we may assume that future returns will follow a stochastic path with a certain distribution of volatility and drift. These two assumption sets will 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. Lastly, to be effective practitioners we must be able to contextualize the “small world” results within the “large world” that exists around us.

4.2.1. “Small world” vs “Large world”

What do we mean by “small world” vs “large world”? Say that our model is one that discounts a fixed set of future cashflows using the US Treasury rate curve. If I run my model using current rates today, and then re-run my model tomorrow with the same future cashflows and the present value of those cashflows has increased by 5% I may ask why the result

4.3. *What is a Financial Model?*

has changed so much in such a short period of time! In the “small”, mechanistic world of the model I may be able to see that the rates I used to discount the cashflows with have fallen substantially. The “small world” answer is that the inputs have changed which produced a mechanical change in the output. The “large world” answer may be that the Federal Reserve lowered the Federal Funds Rate to prevent the economy from entering a deflationary recession. Of course, we can’t completely explain the relation between our model and the real world (otherwise we could capture that relationship in our model!). An effective practitioner will always try to look up from the immediate work and take stock of how the world at large *is* or *is not* reflected in the model.

4.3. 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 is transacted 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 pattern of cashflows or obligations at future timepoints
2. Reducing the projected obligations into a current 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)

4. Elements of Financial Modeling

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).

We should also distinguish a financial model from a purely statistical model, where often the inputs and output data are known and the intention is to estimate relationships between variables (example: linear regressions). That said, a financial model may have statistical components and many aspects of modeling is shared between the two kinds.

4.4. The Process of Building a Financial Model

i TODO: Describe model building process and make associated diagram

4.5. Predictive versus Explanatory Models

Given a set of inputs, our model will generate an output and we are generally interested in its accuracy. *The model need not have a realistic mechanism for how the world works.* That is, we may primarily be 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.

4.5.1. A Historical Example

Consider the classic underdog story where Copernicus overthrew the status quo when he proposed (correctly) that the earth orbited the sun instead of the other way around⁷.

⁷ Prof. 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>

4.5. Predictive versus Explanatory Models

The existing Ptolemy 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 4.1). 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. This was accurate enough to match the observational data that described the position of the planets in the sky.

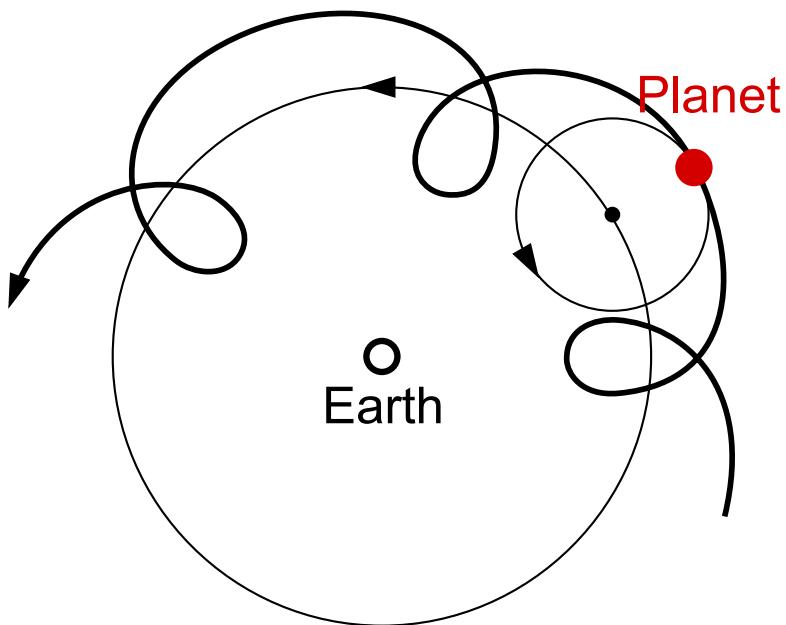


Figure 4.1.: In the Ptolemaic solar model, the retrograde motion of the planets was explained by adding an epicycle to the circular orbit around the earth.

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

4. Elements of Financial Modeling

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 (in favor of 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 criteria 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 interrelationships of a system and much care and thought needs to be given in 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 fit (or 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 help achieve our goals.

4.5. Predictive versus Explanatory Models

4.5.2. Examples in the Financial Context

4.5.2.1. Home Prices

American home prices which have a strong degree of seasonality and have the strongest prices around April of each year. We may find that including a simple oscillating term in our model captures the variability in prices *better* than if we tried to imperfectly capture the true market dynamics of home sales: supply and demand curves varying by personal (job bonus payment timing, school calendars), local (new homes built, company relocation), and national (monetary policy, tax incentives for home-ownership). In other words, one could likely predict a stable pattern like this with a model that contains a simple sinusoidal periodic component. One could likely spend months trying to build a more scientific model and not achieve as good of fit, *even though the latter tries to be more conceptually accurate.*



4.5.2.2. Replicating Portfolio

Another example in the financial modeling realm: in attempting to value a portfolio of insurance contracts a **replicating portfolio** of hypothetical assets will sometimes be constructed⁸. The point of this is to create a basket of assets that can be more quickly (minutes to hours) valued in response to changing market conditions than it would take to run the actuarial model (hours to days). This is an example where the basket of assets has no ability to explain why the projected cashflows are what they are - but retains strong predictive accuracy.

⁸ See, e.g., SOA Investment Symposium March 2010. *Replicating Portfolios in the Insurance Industry* (Curt Burmeister Mike Dorsel Patricia Matson)

4. Elements of Financial Modeling

4.6. What makes a good model?

The answer is: *it depends.*

4.6.1. 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 if we want to use it outside of what it was intended to do. This includes: contents of output and required level of accuracy.

A model may have been created to 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. If we try to add a never-before-seen asset class or use the model to order trades we may be extending the design scope of the original model.

4.6.2. Usability

How easy is it for someone to use? Does it require pages and pages of documentation, weeks of specialized training and an on-call help desk? *All else equal*, it is an indicator of how usable the model is by the amount of support and training. However, one may sometimes wish to create a highly capable, complex model which is known to require a high amount of experience and expertise. An analogy here might be the cockpit of a small Cessna aircraft versus a fighter jet: the former is a lot simpler and takes less training to master but is also more limited.

Figure 4.2 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.

4.6. What makes a good model?

```
[Warning: Found 'resolution' in the theme when creating a 'Scene'. The 'resolution' keyword for 'Scene'  
@ Makie ~/.julia/packages/Makie/iRM0c/src/scenes.jl:220
```

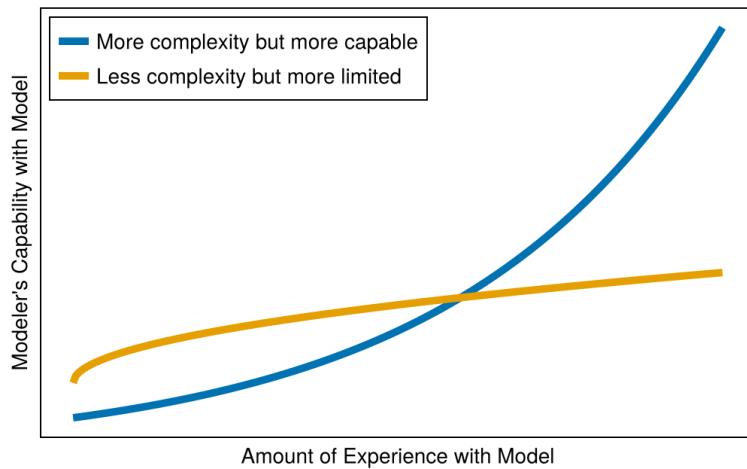


Figure 4.2.: Tradeoff between complexity and capability

4.6.3. Performance

Financial models are generally not used for their awe-inspiring beauty - users are results oriented and the faster a model returns the requested results, the better. Aside from direct computational costs such as server runtime, a shorter model runtime means that one 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 model overnight, (2) see results failed in the morning, (3) spend 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 valuations are different when it comes to performance.

4. Elements of Financial Modeling

4.6.4. Separation of Model Logic and Data

When business logic is embedded within data, or data inputs are spread out across multiple locations it's tough to keep track of things. Using a spreadsheet as an example, often times it's incredibly difficult to ascertain a model's operation if inputs are spread out across locations on many 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.

4.6.5. Abstraction of Modeled Systems

At different times we are interested in different **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 4.2.: 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.

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

4.7. What makes a good modeler?

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 model that effectively exist on the same level of vertical abstraction. For example, both asset and liability mechanics sit at the most granular level in Table 4.2, 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.

4.7. 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?

4.7.1. Domain Expertise

An expert who knows enough about all of the domains that are applicable 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 business to attempt to pair a business expert with an information technologist in the same way.

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 more

4. Elements of Financial Modeling

a more general and flexible skill-set. There's more technologists starting banks than there are financiers starting technology companies.

4.7.2. Model Theory

If it is granted that financial modeling must involve, as the essential part, a building up of modeler's knowledge, the next issue is to characterize that knowledge more explicitly. The modeler's knowledge should be regarded as a theory, in the sense of Ryle's⁹ "Concept of the Mind." Very briefly: a person who has or possesses a theory in this sense knows how to do certain things and in addition can support the actual doing with explanations, justifications, and answers to queries, about the model and its results¹⁰.

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. And 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 then it cannot successfully be maintained through time:

The conclusion seems inescapable that at least with certain kinds of large programs, the continued adaption, 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.

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

4.7. What makes a good modeler?

- 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.
- Analogize how the system works to themselves or 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:

4. Elements of Financial Modeling

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 bright now. However, we do want to emphasize that the process itself being able to 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.

4.7.3. Curiosity

A model never answers all of the questions and many times find itself overdrawn: sometimes more questions arise than answers provided. It is our experience that you modeler who

4.7. What makes a good modeler?

continues to pursue questions that arise as a result of the analysis and in particular possesses an Insatiable itch for resolving apparent contradictions in model conclusions. That is, if an incomplete understanding or an incorrect model allows one to arrive at contradictory conclusions it's suggest that a deeper understanding or model revision is required.

4.7.4. Rigor

When developing a model it's important to ensure that assumptions and parameters are very clear, the methodology is in line with established theory, inappropriate thought has been given to how the model will be used. Additionally one should be mindful of standards of practice. For example, professional actuarial societies have a long list of Actuarial Standards of Practice ("ASOPs"), some of which apply to modeling and the use of data that models ultimately rely on.

4.7.5. Clarity

A rigorous understanding of the fundamentals is important as it is all too easy to let imprecise communication and terminology interfere with the task at hand. Many terms in finance are overloaded with multiple meanings depending on the context such as the speaker's background or company norms. When there is a term that is prone to misunderstanding because of its multiple overloaded meanings, a practitioner should take care to use that term and convey which definition is intended either explicitly or through the appropriate context clues.

4.7.6. Humble

Irreducible & epistemic/reducible uncertainty...

4.7.7. Architecture

Any sufficiently complex project benefits from architectural thinking

4. Elements of Financial Modeling

4.7.8. Planning

When tackling a large problem, it helps

4.7.9. Toolset

An experience professional is aware of a number of approaches that can be used in solving a problem. From heuristics that are able to be calculated on a napkin to complex economic models, the ability to draw on a wide tool set allows a practitioner to find the right solution for a given problem. Further, it is the intention of this book to enumerate a number of additional approaches that may prove useful in practice.

5. Elements of Programming

“Programming is not about typing, it’s about thinking.” — Rich Hickey (2011)

5.1. In this section

Start building up computer science concepts by introducing tangible programming essentials. Data types, variables, control flow, functions, and scope are introduced.

💡 On Your First Readthrough

This chapter is intended to be an introductory reference for most of the basic building blocks which we will build abstractions on top of in 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 begin directly searching or reading the documentation of a language or library itself. However, it may be intimidating or frustrating reading reference documentation due to the density and terminology - 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 5.5.4 through Section 5.5.9 which covers advanced and custom data types
- After Section 5.6.3 which deals with advanced func-

5. Elements of Programming

tion usage and program organization via scope

5.2. Computer Science, Programming, and Coding

¹¹ Said 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 ($3^2 + 4^2 = 5^2$).

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 concepts¹¹. 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 and 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 problem characteristics. Having the language to describe a concept will also help see aspects of the problem in new ways, opening one up to more innovative solutions.

In the context of this financial modeling that we do, we can consider a financial model to be a type of computer program. It takes as input abstract information (data), performs calculations (an algorithm), and returns new data as an output. In this context, we generally do not need to consider many things that a software engineer may contemplate such as a graphical user interface, networking, or access restrictions. But there are many similarities: a good financial modeler must understand data types, algorithms, and some hardware details.

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

5.2. Computer Science, Programming, and Coding

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 Chapter 27 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:

```
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. Elements of Programming

🔥 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.

5.3. Expressions and Control Flow

5.3.1. Assignment and 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

0.56818181818182
```

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.

We will dive a little bit deeper into variables and assignment in Section 5.4.3.

5.4. Expressions

Having already seen some more illustrative examples above, we can zoom in onto smaller pieces called **expressions** which are effectively the basic block of code that gets evaluated. 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.

5.4.1. Compound Expression

There's 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)
```

The variables inside the `begin` block are evaluated in the same scope as `c` and therefore have the assigned values when we call `a` and `b` in the last line. Contrast that with the `let` block below, where `d` and `e` are not available when we try to get the value of `f`. This is because `let` creates a new inner scope that's not available in `f`'s scope. More on scope later in the chapter.

5. Elements of Programming

```
f = let
    d = 1
    e = 2
    d + e
end
f

3

d

LoadError: UndefVarError: `d` not defined
UndefVarError: `d` not defined
```

5.4.2. 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
Float64 isa Rational
(5 > 0) & (-1 < 2) # "and" expression
(5 > 0) | (-1 > 2) # "or" expression
1 != 2
```

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 (in some languages, any non-zero number is “truthy”).
- `true` is not equal to 1 (`egal` is defined later in this chapter).

5.4. Expressions

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")
```

5.4.2.1. Equality

The “Ship of Theseus¹²” problem is an example of how equality can be 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

¹² The 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. Elements of Programming

- **Equal** equality is when the values of two objects are the same

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

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

```
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 equal because there is no way for us to make them different:

5.4. Expressions

```
(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”.

5.4.3. Assignment and Variables

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 something 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]                                ①  
x[1] = 5                                     ②  
x
```

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                                              ①  
x[1] = 6  
y
```

5. Elements of Programming

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

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

Note that variables can be re-assigned unless they are marked as `const`:

```
const PHI = π * 2 # <1>
```

- ① Capitalizing constant variables is a convention in Julia.

If we tried to re-assign `PHI`, we would get an error.

⚠ 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.

5.4.4. 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`):

```
# use fixed indices
for i in 1:3
    println(v[i])
end
```

5.4. Expressions

```
# use 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
end
```

- ① `global` is used to increment *i* by 1. *i* is defined outside the scope of the `while` loop (see Section 5.7).

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

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

5. Elements of Programming

5.4.5. Performance of loops

Loops are highly performant in Julia and often the fastest way to accomplish things. Those coming from Python or R may have developed a habit to avoid writing loops. *Fear the for loop not!*

5.5. Data Types

Data types are a way of categorizing information by intrinsic characteristics. We instinctively know that 13.24 is different than "this set of words" and types are how we will formalize this distinction. This is a key conceptual point, and mathematically it's like we have different sets of objects to perform specialized operations on. Beyond this set-like abstraction is 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 that a computer can understand if it should interpret 01000010 as B or as 66¹³.

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 handle¹⁴.

5.5.1. Numbers

Numbers are usually grouped into two categories: **integers** and **floating-point**¹⁵ 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)¹⁶ and the result is **printed** below the input:

2

2

5.5. Data Types

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 since most architectures these days are 64-bit¹⁷, while on a 32-bit system 0.2 would be interpreted as a `Float32`. 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

¹⁷ This means that their central processing units (CPUs) use instructions that are 64 bits long.

5. Elements of Programming

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 5.4.3.
- ② `big(x)` is a 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 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:

- -32,767 to 32,767 for `Int32`
- -2,147,483,647 to 2,147,483,647 for `Int64`

5.5. Data Types

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 standard C++ type by default (that is, Int(30) will create a 64 bit integer on 64-bit systems and 32-bit on 32-bit systems). 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.

5. Elements of Programming

5.5.2. 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.

Figure 5.1 shows the type hierarchy for most built-in Julia number types.

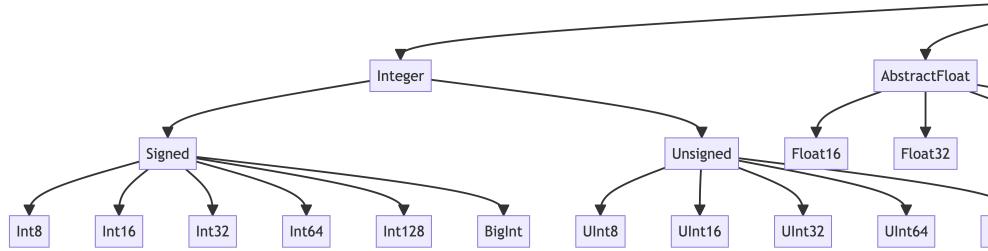


Figure 5.1.: Numeric Type Hierarchy in Julia. Leafs of the tree are concrete types.

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.

5.5.3. Collections

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

5.5. Data Types

5.5.3.1. 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 represented¹⁸ in floating point.

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

4-element Vector{Float64}:

```
1.0  
1.3  
300.0  
21.0
```

¹⁸ Accurate only to a limited precision, as described in Section 5.5.1.

5. Elements of Programming

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] =
 0.416769  0.153191  0.641468
 0.449059  0.657702  0.353166
 0.957003  0.872806  0.192663

[:, :, 2] =
 0.769644  0.0383001  0.419858
 0.967838  0.19581    0.0077235
 0.690151  0.039209  0.432719

[:, :, 3] =
 0.242051  0.507494  0.543275
 0.577441  0.782059  0.729001
 0.580694  0.303957  0.0664385

[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
```

5.5. Data Types

```
[:, :, 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 by semi-colons (;):

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

2×3 Matrix{Int64}:

```
1 2 3  
4 5 6
```

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.

5.5.3.2. Array indexing

Array elements are accessed with the integer position, starting at 1 for the first element¹⁹ ²⁰:

```
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]
```

¹⁹ Whether 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.

²⁰ Arrays in Julia can actually be indexed with an arbitrary starting point: see the package `OffsetArrays.jl`

5. Elements of Programming

```
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
```

5.5.3.3. Array Alignment

When you have an MxN 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

5.5. Data Types

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 Nx1 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 10x10 matrix is actually stored in memory as 100 elements coming in order, one after another in single file.

This 3x4 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)
```

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

5.5.3.4. Ranges

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

5. Elements of Programming

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 1 through the number of atoms on the earth instantaneously:

This is possible due to two things:

1. not needing to actually store that many numbers in memory, and
 2. Julia being smart enough to apply the triangular number formula²¹ when `sum` is given a consecutive range.

²¹ The triangular numbers (sum of integers from 1 to n) are:

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

There are more general ways to construct ranges:
Step 3 by another number instead of the default 1:

1:2:7

1:2:7

²² Whether the last number is in the resulting range depends on if the step evenly divides the end of the range.

Specify the number of values within the range, inclusive of the first number²²:

```
range(0, 10, 21)
```

0.0:0.5:10.0

5.5.3.5. 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'

5.5. Data Types

'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 collection²³ 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:

```
"""
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

²³ Under 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.

5. Elements of Programming

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: 'a' 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.

5.5.3.6. Tuples

Tuples are a set of values that belong together and are denoted by a values inside parenthesis and separated by a comma. An example might be x-y coordinates in 2 dimensional space:

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

(3, 4)

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`).

5.5. Data Types

- 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 case for a tuple.
- Tuples are generally stack allocated instead of being heap allocated like arrays²⁴, meaning that a lot of times they can be faster than arrays.

²⁴ What this means will be explained in Chapter 9 .

5.5.3.6.1. 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
```

5.5.3.7. 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:

5. Elements of Programming

```
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.

5.5.4. Parametric Types

We just saw how tuples can contain heterogeneous types of data inside a common container. Let's look at this a little bit closer by looking at the full type:

```
typeof(p1)
```

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

location 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}
```

5.5. Data Types

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.

5.5.5. 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 something *should* be there but it’s 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:

 Tip

`Missing` and `Nothing` are the *types* while `missing` and `nothing` are the values here¹. This is analogous to `Float64` being a type and `2.0` being a value.

5.5.6. 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

¹Missing and Nothing are instances of **singleton type**, which means that there is only a single value that either type can take on.

5. *Elements of Programming*

Table 5.1.: Three value logic with `true`, `missing`, and `false`.

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

(b) And logic			
AND (&)	<code>true</code>	<code>missing</code>	<code>false</code>
<code>true</code>	<code>true</code>	<code>missing</code>	<code>false</code>
<code>missing</code>	<code>missing</code>	<code>missing</code>	<code>false</code>
<code>false</code>	<code>false</code>	<code>false</code>	<code>false</code>

(c) Or Logic			
OR ()	<code>true</code>	<code>missing</code>	<code>false</code>
<code>true</code>	<code>true</code>	<code>true</code>	<code>true</code>
<code>missing</code>	<code>true</code>	<code>missing</code>	<code>missing</code>
<code>false</code>	<code>true</code>	<code>missing</code>	<code>false</code>

5.5. Data Types

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`.

5.5.7. 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:

`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
```

5. Elements of Programming

Line 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)
```

But both are now subtypes of `PPoint2D`. 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 the `x` and `y` are both of the same type in each `PPoint2D` that we created. If instead we wanted to allow the coordinates to be of different types, then we could have defined `PPoint2D` 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!

```
struct Point
    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 anything more about the type until runtime (the time at which our program encounters the data when running). 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** 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
```

5. Elements of Programming

```
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 for,e.g. "theta LaTeX" to find the appropriate name. Furhter, 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>
```

5.5.8. 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
```

5.5. Data Types

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

Generally you should default to using immutable types and consider only moving to mutable types in specific circumstances. You'll see some examples in the applications later in the book.

5.5.9. 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 `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] MyDate(year::Int64, month::Int64, day::Int64)
    @ In[54]:2
[2] MyDate(year, month, day)
    @ In[54]:2
```

Implicit constructors are nice in that you don't have to define a default method and the language does it for you. Sometimes

5. Elements of Programming

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")
        else if ~(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
```

5.6. Functions

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

5.6.1. 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 parathesis 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

5. Elements of Programming

- $-(x)$ 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: $-(x::\text{Float64})$ is a different method than $-(x::\text{Int})$

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.

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

5.6.3. Defining Methods on Types

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

```
function distance(point)          ①
    return sqrt(point.x^2 + point.y^2)  ②
end
```

- 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 make explicit what value the function will output.

`distance` (generic function with 1 method)

i 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.

5. Elements of Programming

```
"""
    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 **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 defintion 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 27.4.1) 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 the-

ory... 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, would define the method with the signature of `myfunction(Tuple{<:Real})` or `myfunction{T}(Tuple{T})` where `{T<:Real}`.

5.6.4. Keyword Arguments

Keyword arguments are arguments that are passed to a function but do not use *position* to pass data to functions but instead used named arguments. In the following example, `filepath` is a **positional argument** while the two arguments after the semi-colon (`;`) are keyword arguments.

```
function read_data(filepath; normalize_names, has_header_row)
    # ... function would be defined here
end
```

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

```
read_data("results.csv"; normalize_names=true, has_header_row=false)
```

5.6.5. 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;
    normalize_names = true,
    has_header = false
)
```

5. Elements of Programming

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.

5.6.6. 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 something from another value, or defining a function within optimization or solving routines.

5.6.7. 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

5.6. Functions

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)           ①
)
    sum(
        loss_function.(modeled_obs, actual_obs)
    )
end

let
    a = loss([1, 5, 11], [1, 4, 9], square_error)  ②  ③
    b = loss([1, 5, 11], [1, 4, 9], abs_error)
    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)

5.6.8. Broadcasting

Looking at the prior definition of `dist`, what if we wanted to compute the squared distance from the origin for a set of

5. Elements of Programming

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  
 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 values 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 can happen at the same time 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.

5.6.8.1. 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 1xN, “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 MxN and a PxQ 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 1x1 element is being combined with a 4x1, 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 1x3 works with the 2x3 (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
```

5. Elements of Programming

This 3×1 isn't compatible with this 2×3 array (neither M and P nor N and Q match)

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

```
LoadError: DimensionMismatch: arrays could not be broadcast to a common size;
DimensionMismatch: arrays could not be broadcast to a common size; got
```

Stacktrace:

```
[1] _bcs1
    @ ./broadcast.jl:555 [inlined]
[2] _bcs
    @ ./broadcast.jl:549 [inlined]
[3] broadcast_shape
    @ ./broadcast.jl:543 [inlined]
[4] combine_axes
    @ ./broadcast.jl:524 [inlined]
[5] instantiate
    @ ./broadcast.jl:306 [inlined]
[6] materialize(bc::Base.Broadcast.Broadcasted{Base.Broadcast.Def...
    @ Base.Broadcast ./broadcast.jl:903
[7] top-level scope
    @ In[67]:1
```

This 2×4 isn't compatible with the 2×3 (M and P match, but N nor Q is 1):

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

```
LoadError: DimensionMismatch: arrays could not be broadcast to a common size;
DimensionMismatch: arrays could not be broadcast to a common size; got
```

Stacktrace:

```
[1] _bcs1
    @ ./broadcast.jl:555 [inlined]
[2] _bcs (repeats 2 times)
    @ ./broadcast.jl:549 [inlined]
[3] broadcast_shape
    @ ./broadcast.jl:543 [inlined]
[4] combine_axes
```

5.6. Functions

```
@ ./broadcast.jl:524 [inlined]
[5] instantiate
@ ./broadcast.jl:306 [inlined]
[6] materialize(bc::Base.Broadcast.Broadcasted{Base.Broadcast.DefaultArrayStyle{2}}, Nothing, type
    @ Base.Broadcast ./broadcast.jl:903
[7] top-level scope
@ In[68]:1
```

5.6.8.2. 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, the result above is the result of this broadcasted result:

```
in(1, [1,2,3]) # the first row of the above result
in(9, [4,5,6])
```

If we were expecting Julia to return `[1,0]` (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:

5. Elements of Programming

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

```
2-element BitVector:
```

```
1  
0
```

5.6.9. Passing by Sharing

We often want to share data between scopes, such as between modules or by passing something 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[1] = 2 * v[i]  
    end  
end  
  
double!(v)
```

```
v
```

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

```
6  
2  
3
```



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

5.7. Scope

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
```

5.7. 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

5. Elements of Programming

- 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 error:

j

```
LoadError: UndefVarError: 'j' not defined
UndefVarError: 'j' not defined
```

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("bonds.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).

5.7.1. 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:

5. Elements of Programming

```
module Shape  
    (1)  
  
    struct Triangle{T}  
        base::T  
        height::T  
    end  
  
    function area(t::Triangle)  
        (2)  
        return 1 / 2 * t.base * t.height  
    end  
end  
  
t = Shape.Triangle(4, 2)  
area = Shape.area(t)  
    (3)  
    (4)
```

- ① module defines an encapsulated block of code which is anchored to the namespace Shape
- ② Here, area 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

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

5.7. Scope

- **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`).

Part III.

Conceptual Foundations: Abstractions

6. Functional Abstractions

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. In this section

Demonstrate different approaches to a problem which gradually introduce more re-usable or general techniques. These techniques will allow for constructing sophisticated models while maintaining consistency and simplicity. Imperative programming, functional programming, and recursion.

6.2. Introduction

This chapter will center around a simple task: calculate the present value of a portfolio of a single fixed, risk-free, coupon-paying bonds under two different interest rate environments. The focus will be on describing different approaches to this problem, not be adding complexity to the problem (e.g. no getting into credit spreads, settlement timing, etc.).

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

- ① The rates given are the one year rate for time zero, time one, etc.

The other bond and set of rates is described later in the chapter.

6. Functional Abstractions

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

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

Where

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

We will repeatedly solve the same problem before extending it to more examples. It may feel repetitive but the focus here is not the problem, but rather the variations between the different approaches.

6.3. Imperative Style

One of the most familiar styles of programming is called **imperative** (or **procedural**), where we provide step-by-step commands are provided to the computer. The programmer defines the data involved and how that data 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 the program's variables) is defined and managed by the programmer explicitly.

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

```
let (1)
    pv = 0.0
    discount = 1.0

    for i in 1:length(cf_bond) (2)
        discount = discount / (1 + rate[i])
        pv = pv + discount * cf_bond[i]
    end (3)
```

6.3. Imperative Style

```
pv  
end
```

- ① Declare variables to keep track of the discount rate and running total for value
- ② Loop 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. We are making the *assumption* that the indices of the vectors start with one, when in reality Julia arrays *can* be defined to start at 0 or another arbitrary index.
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:

6. Functional Abstractions

```
eachindex(cf_bond)
```

```
Base.OneTo(5)
```

Lazy Programming

The result, Base.OneTo(5) is a **lazy** object which represents a collection that does not get fully instantiated until asked to (which may not actually be necessary). Many (most?) iterators are lazy but we can interact with them without fully instantiating the data that they represent. For example, we could find the largest index:

```
maximum(eachindex(cf_bond))
```

```
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))
```

```
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.

And when used in context:

```
let  
    pv = 0.0  
    discount = 1.0
```

6.3. Imperative Style

```
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 cash-flows, 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`:

```
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 value 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)
```

6. Functional Abstractions

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 this approach 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.

Note that it's when state is mutable that a program tends to be more complex.

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 (#sec-anonymous-functions) are used frequently to define intermediary steps.

6.4. Functional Techniques and Terminology

Table 6.1.: Important Functional Methods.

Function	Description	Example
<code>map(f,v)</code>	Apply function f to each element of the collection v.	<code>map(</code> <code>x→x^2,</code> <code>[1,3,5]</code> <code>) # [1,9,25]</code>
<code>reduce(op,v)</code>	Apply binary op to pairs of values, reducing the dimension of the collection v.	<code>reduce(</code> <code>*</code> , <code>[1,3,5]</code> <code>) # 15</code>
	Has a couple of important, optional keyword arguments to note (which also apply to other variants of <code>reduce</code> below):	<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 <code>0</code> and <code>1</code> respectively) • <code>dims</code> defines which dimension to reduce across (if the dimension of v is more than one).
<code>mapreduce(op,f,M)</code>	Maps f over collection v and returns a reduced result using op.	<code>mapreduce(</code> <code>*</code> , <code>x→x^2,</code> <code>[1,3,5]</code> <code>) # 35</code>

6. Functional Abstractions

Function	Description	Example
<code>foldl(op, v)</code>	Like <code>reduce</code> , but applies op from left to right (<code>foldl</code>) or right to left (<code>foldr</code>). Also has <code>mapfoldl</code> and <code>mapfoldr</code> versions.	<code>foldl(*, [1,3,5]) # 15</code>
<code>accumulate(op, v)</code>	Apply op along v , creating a vector with the cumulative result.	<code>accumulate(+, [1,3,5]) # [1, 4, 9]</code>
<code>filter(f, v)</code>	Apply f along v and return a copy of v with elements where f is true	<code>filter(>=(3), [1,3,5]) # [3, 5]</code>

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 times 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 drive the calculation in any order since the operation is associative.
4. The lack of mutable state

Let’s build a version of the present value calculation using the functional building blocks described above.

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.

6.4. Functional Techniques and Terminology

First, to show how we could calculate the discount factor 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. This operation is conveyed visually in Figure 6.1.

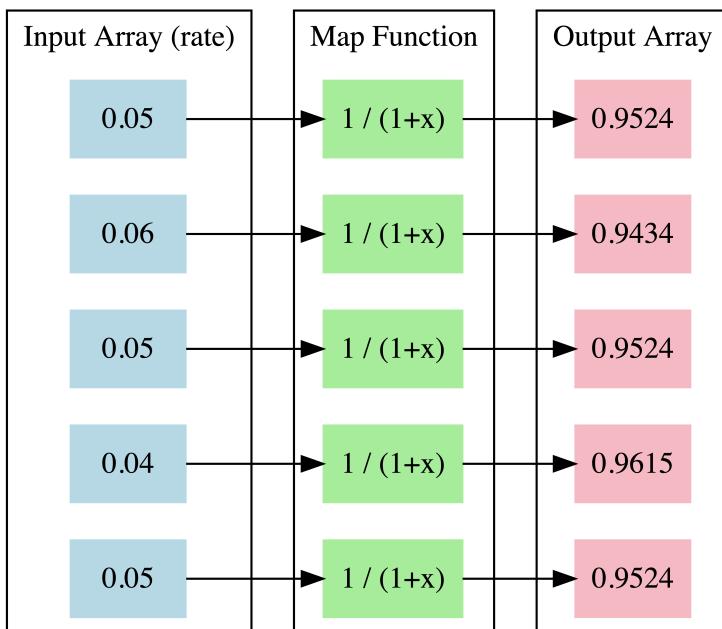


Figure 6.1.: A diagram showing that `map` creates a new collection mirroring the old one, after applying the given function to each element in the original collection.

6. Functional Abstractions

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).

An anti-pattern where `map` would likely be a better tool than a loop often looks like this:

```
output = []
for x in collection
    result = # ... do stuff ...
    push!(output,result)
end
output
```

Instead, `map` simplifies this to:

```
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 manage 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:

```
accumulate(*, map(x → 1 / (1 + x), rate))
```

```
5-element Vector{Float64}:
0.9523809523809523
```

6.4. Functional Techniques and Terminology

```
0.898472596585804
0.8556881872245752
0.822777103100553
0.7835972410481457
```

This results in a vector of the cumulative discount factors for each point in time corresponding to the given cashflows.

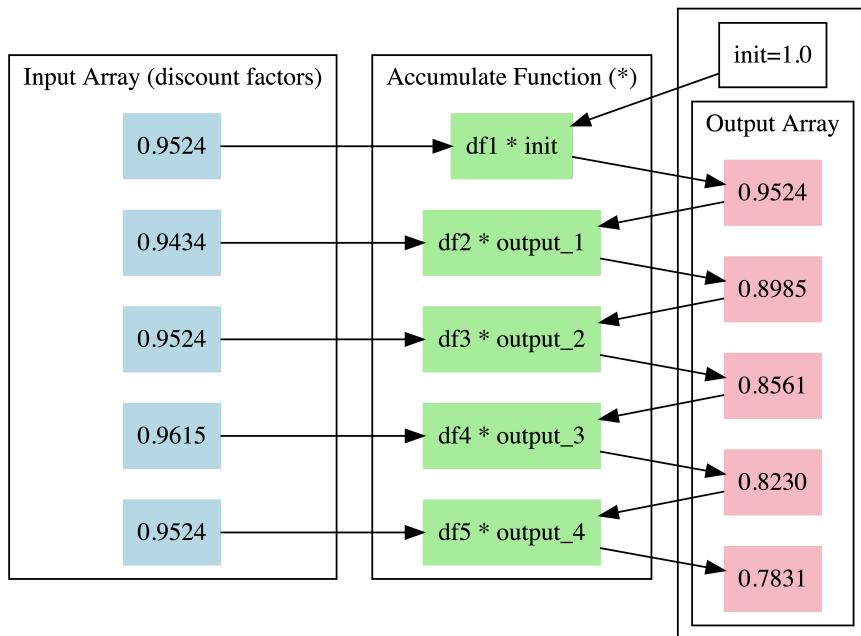


Figure 6.2.: A diagram showing that `accumulate` creates a new collection where each element is the cumulative result of applying the given operation to all previous elements.

i Note

For `accumulate` and `reduce`, an important value is the `init` (an optional keyword argument), which is the initial value to start the accumulation or reduction. By default, for common operations this **identity element** is predefined. For example, for `+` the identity is 0 while for `*` it is 1. The identity element `e` is the one where for a given

6. Functional Abstractions

binary operation \odot , that $x \odot e = x$.

Another example beyond addition and subtraction 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 things that work with the binary operator and identity elements as described above is a **monoid**. You may ignore this fact.*

6.4.3. `reduce`

`reduce` takes an operation and a collection and applies the operation repeatedly to pairs of elements 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`:

6.4. Functional Techniques and Terminology

```
reduce(+, discounted_cfs)
```

121.48888490821487

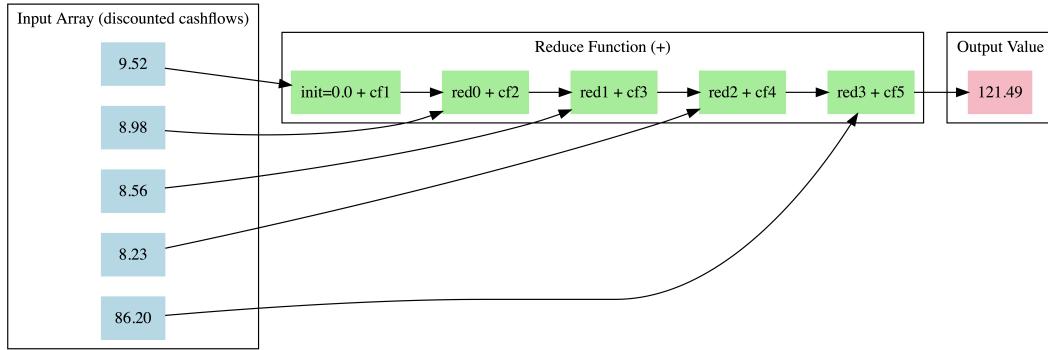


Figure 6.3.: A diagram showing how `reduce` applies the given operation to pairs of elements, ultimately reducing the collection to a single value.

6.4.4. mapreduce

We can combine `map`, `accumulate` and `reduce` to concisely calculate the present value in a functional style:

```
dfs = accumulate(*, map(x → 1 / (1 + x), rate))
mapreduce(*, +, cf_bond, dfs)
```

121.48888490821487

This calculates the discount factors, applies them to the cash-flows with `map`, and sums the result with a reduction.

Tip

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!

6. Functional Abstractions

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 (cf, r) in zip(rates, cashflows)      ①
        discount = discount / (1 + r)
        pv = pv + discount * cf
    end
    pv
end
```

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`.

6.4.5. filter

`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:

6.4. Functional Techniques and Terminology

```
filter(isodd, 1:6)
```

```
3-element Vector{Int64}:
1
3
5
```

Or filtering out things that don't match a criteria:

```
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. Functional Abstractions

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
        "prime"
    elseif x == 2
        "not prime"
    elseif x == 3
        "prime"
    elseif x > 4
        "probably not prime"
    end
end,
[1, 2, 3, 10]
)
```

This can be written more cleanly with the `do` syntax:

```
map([1, 2, 3, 10]) do x
    if x == 1
        "prime"
    elseif x == 2
        "not prime"
    elseif x == 3
        "prime"
    elseif x > 4
        "probably not prime"
    end
)
```

6.4. Functional Techniques and Terminology

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
8.100000000000001
7.3
```

Or an example with the `do` syntax:

```
map(discounts, cashflows) do d, c
    d * c
end
```

3-element Vector{Float64}:

```
9.0
8.100000000000001
7.3
```

6.4.6.3. Mixing Funcitonal And Imperative Styles

One of the best things about Julia is how natural it can be to mix the different styles. Sometimes the best is the mix of both styles and that's one of the benefits of Julia: use the style that's most natural to the problem.

Note

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

6. Functional Abstractions

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 because is that it’s 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 call 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

Array-oriented programming is one that is practiced in two main contexts:

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 define C++ (or similar) libraries for users to call out to.

6.5. Array-Oriented Styles

Array-oriented programming is not always natural for financial and actuarial applications. Differences in behavior or timing of underling 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.: Julia's broadcasting makes for 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] cf_bond = np.array([10, 10, 10, 10, 110]) discount_factors = cumprod(1 / np.array([0.05, 0.06, 0.05, 0.04, 0.05])) sum(cf_bond .* discount_factors) discount_factors = np.cumprod(1 / (1 + rate)) result = np.sum(cf_bond * discount_factors)</pre>	

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. Functional Abstractions

6.6. Recursion

The Fibonacci sequence is a classic example of a recursive algorithm:

$$F(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F(n - 1) + F(n - 2), & \text{if } n > 1 \end{cases}$$

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
```

One could imagine a possible pattern where the value of a stream of cashflows is defined as the sum of the value of the discounted next period plus the cashflows that occur this period.

```
function pv_recursive(rate,cashflows,accumulated_value=0.0)
    if isempty(cashflows)
        return accumulated_value
    else
        v = (accumulated_value + cashflows[end]) / (1 + rate)
        return pv_recursive(rate,cashflows[begin:end-1], v)
    end
end

pv_recursive (generic function with 2 methods)
```

And an example of its use:

6.6. Recursion

```
pv_recursive(0.05,[10,10,10])
```

```
27.232480293704782
```

Generally, the recursive pattern includes defining a ‘base case’ where you stop the recursive behavior and return the result that has been accumulated up to that point.

7. Data and Types

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. In this section

The powerful benefits that using assigning types to data has within the model's system, some examples and relating some aspects of object oriented design.

7.2. Using Types to Value a Portfolio

We will assemble a set of interfaces that let's us value a portfolio of assets. Using the constructs introduced in the prior chapter, we can describe this as additively reducing over the value-mapped collection of assets in the portfolio. Or more concisely:

```
mapreduce(value,+,portfolio)
```

In `portfolio`, the assets may be heterogeneous so we will need to define what the valuation semantics are for the different kinds of assets. To get to our end goal, we will need to:

1. Define the different kinds of assets within our portfolio
2. How the assets are to be valued.

7.3. Benefits of Using Types

Defining types allows us to do several things:

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.
3. **Dispatch on type.** 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 5.5.7 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 five types of assets in this simplified universe:

- Cash
- Risk Free Bonds (coupon and zero-coupon varieties)
- European Puts and Calls on Equities

To do the valuation of these, we need some economic parameters as well: risk free rates and option implied volatilities, which we will pass via named tuples. In a more robust model it would be wise to use types to differentiate between different kinds of economic assumption sets but we will limit the scope here such that

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

abstract type AbstractBond <: AbstractAsset end      ②

struct CouponBond <: AbstractBond
    par::Float64
    coupon::Float64
    tenor::Int
end

struct ZeroCouponBond <: AbstractBond
    par::Float64
    tenor::Int
end
```

7. Data and Types

- ① General convention is to name abstract types beginning with `Abstract`...

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:

```
... { .cell_execution_count=3 } { .julia .cell-code } econ_baseline=
EconomicAssumptions(0.05)

... { .cell-output .cell-output-display execution_count=4 }
EconomicAssumptions{Float64}(0.05) ... :.
```

Now on to defining the valuation for `Cash` and `AbstractBonds`. `Cash` is always equal to its balance:

```
value(asset::Cash, ea::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
    value = 0.0
    for t in 1:asset.tenor
        discount_factor /= (1 + r)                      ①
        value += discount_factor * cashflow(asset, t)
    end
    return value
end

function cashflow(bond::CouponBond, time)
```

7.4. Defining Types for Portfolio Valuation

```
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)

7.4.1. (Multiple) 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 the right 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"
```

7. Data and Types

```
# special code for Par bonds
# ...
elseif asset.type == "AmortizingBond"
    # special code for Amortizing Bonds
    # ...
else
    # here define the generic AbstractBond logic
end
end
```

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 the next section).

In our definition of `value` above, we used a simple scalar interest rate to determine the rate to discount the cash flows. What if instead of a scalar interest rate value we wanted to instead pass an object that represented a term structure of interest rates? Note how in the definition of `value` for `ZeroCouponBond`, we have defined a *more specific* signature: both the first and second arguments are specific, concrete types. When we call `value(ZeroCouponBond(100.0,3),0.05)`, we avoid the loop that's defined in the generic case and jump immediate to a more efficient definition of its `value`. This is **dispatching** on the combination of types and picking the most relevant (specific) version for what has been passed to it.

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`:

```
#| error: true
value(ZeroCouponBond(100.0,5),econ_baseline)
```

Let's fix that. Here we define 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.r)
```

7.4. Defining Types for Portfolio Valuation

```
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

i Note

This code:

```
mapreduce(asset-> value(asset,econ_baseline), +, portfolio)
```

is more verbose than what we set out do at the start (`mapreduce(value,+,portfolio)`) due to the two-argument `value` function requiring a second argument

7. Data and Types

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 which "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".

This is, Base.Fix1(f, x) behaves like $y \rightarrow f(x, y)$ and Base.Fix2(f, x) behaves like $y \rightarrow f(y, x)$.

In the context of our valuation model, this would look like:

```
val = Base.Fix2(value,econ_baseline)
mapreduce(val,+,portfolio)
```

```
228.3526166468459
```

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"#15#16"}(var"#15#16"())
```

Now value will not work, because we've only defined how value works on bonds if the given rate is a Float64 type:

```
#| error: true
value(ZeroCouponBond(100.0, 5), econ_sin)
```

We can extend our methods to account for this:

```
function value(bond::ZeroCouponBond, r::T) where {T<:Function} ①
    return bond.par / (1 + r(bond.tenor))^(bond.tenor) ②
end
```

7.5. Object Oriented Design

- ① The `r :: T ... where {T<:Function}` says use this method if `r` is any concrete subtype of the (abstract) `Function` type.
- ② `r` is a function, where we call 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
```

7.5. Object Oriented Design

There's enough general familiarity with object oriented ("OO") design that it's worth describing for understanding how it compares and contrasts to other design patterns. Object oriented systems attempt to form the analogy that various parts of the system are their own objects which encapsulate both data and behavior. Object oriented design is often one of the first computer programming abstractions introduced because it is very relatable²⁵, however this comparative discussion will point out a number of its flaws as well. That said, much of OO design can be emulated in Julia except for data inheritance.

We bring up object oriented design not because of the authors (admittedly subjective) opinion that the object-oriented paradigm can be less suitable for financial modeling, but because by having a (potentially more relatable) contrasting approach we can better illuminate certain ideas and concepts.

²⁵ And 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.

7.6. Assigning Behavior

The `value` function is a good example of where the OO requirement to ascribe behavior to a single type (class) can lead to

7. Data and Types

confusing design. If we had to assign value to one of the objects involved, should it be the economic parameteres of the asset contracts? 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 possible define in advance the valuation semantics of all types of assets? What if one wanted to extend the valuation to a new asset class? These are problems presented in traditional OO designs and that resolve so elegantly with multiple dispatch.

7.7. Inheritance

We discussed the type hierarchy in Chapter 5 and 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.

Moreover, inheritance can sometimes lead to the duplication of code across the hierarchy, especially if the inherited behavior needs to be slightly modified in different child classes. This goes against the DRY (Don’t Repeat Yourself) principle, which is a fundamental concept in software engineering advocating for the reduction of repetition in code.

7.7.1. Composition over Inheritance

To mitigate some of the problems associated with inheritance, there's 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. There is a general preference for “composition over inheritance” among professional developers these days.

In composition, objects are constructed from other objects, and behaviors are delegated to these contained objects. This approach allows for greater flexibility, as it's easier to change the behavior of a system by replacing parts of it without affecting the entire hierarchy, as is often the case with inheritance.

Composition looks like this:

```
struct CUSIP
    code::string
end

struct FixedIncome
    coupon::Float64
    tenor::Float64
end

struct MunicipalBond
    cusip::CUSIP
    fi::FixedIncome
end

struct ListedOption
    cusip::CUSIP
    #... other data fields
end

struct UnlistedBond
    fi::FixedIncome
end
```

7. Data and Types

```
# define behavior which relies on defining
last_transaction(c::CUSIP) = # ... perform lookup of data
last_transaction(asset) = last_transaction(asset.cusip)

duration(f::FixedIncome) = # ... calculate duration
duration(asset) = duration(asset.fi)
```

In the above example, there are number of asset classes that have CUSIP related attributes (i.e. the 9 character code) and behavior (e.g. being able to look up transaction data). Other assets have fixed income attributes (e.g. calculating a duration). But not all of these assets have a CUSIP! Composition lets us bundle the data and behavior together without needing complex chains of inheritance.

Note

A CUSIP (Committee on Uniform Security Identification Procedures) number, is a unique nine-character alphanumeric code assigned to securities, such as stocks and bonds, in the United States and Canada. This code is used to facilitate the clearing and settlement process of securities and to uniquely identify them in transactions and records.

8. Higher Levels of Abstraction

“Simple things should be simple, complex things should be possible.” — Alan Kay (1970s)

8.1. In this section

Why we talk about abstraction as a technique in and of itself, discussion of abstraction at the level of code organization and interfaces.

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 or types. In this chapter, we zoom out and examine some principles that guide good model development and how that manifests itself 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, where we need trouble ourselves with the low level details.

8. Higher Levels of Abstraction

- 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 combine data, and or function, into a related group of concepts which can be shared in different parts of our model

8.3. Principles for Abstraction

Here is a list of 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.

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 high degree of dependence (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)	Don't let other parts of the program modify internal data and make the system easier to understand and maintain	Defining a type or module with well defined behavior and responsibility

8.3. Principles for Abstraction

Principle	What	Why	Example
Composability	Design simple components that can be combined to create more complex behaviors, as opposed to 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 the common behavior that arises 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

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.

8. Higher Levels of Abstraction

8.3.1. Pragmatic Considerations for Model Design

8.3.1.1. Behavior-Oriented

This strategies is to effectively group together components with a model that behaves 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 models design, So that different groups contributing to the model can focus on any more narrow aspect, Regardless of inherent similarity of components. For example, at a higher vertical level of obstruction, 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 attempt to bundle together more functionality instead of allowing users to compose a functionality that comes from different packages. For example, perhaps a certain visualization of a model result is particularly useful, It is not easy to create from scratch, And virtually everyone using the model, will desire 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.

8.4. Interfaces

In general, though it is preferred to try to loosely couple systems, you can pick and choose which components you use and that those components work well together.

8.4. Interfaces

Interfaces are the boundary between different encapsulated abstractions. The user-facing interface is the set of functionality and details that the user of the package or model must consider, which is separate from the intermediate variables, logic, and complexity that may be contained 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 that are used 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 this might mean that there is an interface for bonds, or there is an interface for interest-rate swaps. There may be a different interface for calculating risk metrics or visualizing the results.

Financial model this might mean that there is an interface for bonds, or there is an interface for interest-rate swaps. There may be a different interface for calculating risk metrics or visualizing the results. A better system design will separate the concern of visualizing output from the mechanics of a fixed income contract. This is what it means to put boundaries on different parts of a models logic. One of the easiest places to see this is with the available open source packages. There are packages available for visualizations, data frames, file, storage, statistical analysis, etc. for many of these it's easy to see where the natural boundary lies.

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 or features. This makes the interface easier to understand and facilitates building the necessary complexity through digestable, 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 of details 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'll 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 publically available as set of installable Julia packages.

8.4.2.1. Background

In actuarial work, it is common to need to work with interest rate and bond yield curves to determine current forward rates, estimates of the shape of future yield curves, or discount a series of cashflows to determine 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, quoted as a continuous rate or annual effective? 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 the type of complexity that we wish to save the user from needing to keep front of mind when the primary goal is, e.g., valuation of a stream of riskless life insurance payments, which might look like this:

```
risk_free_rates = [0.05, 0.06, ... 0.06]
tenors = [1/12, 3/12, ... 30]
yield_curve = Yields.Par(risk_free_rates, tenors)
```

8. Higher Levels of Abstraction

```
cashflow_vector = [1e6,3e6,...,1e3]  
present_value(yield_curve,cashflow_vector)
```

This is very clear from the variable and function names what the purpose and steps in the analysis are. Imagine starting with rates and cashflows in a spreadsheet, needing to perform the bootstrapping, interpolation, and discounting before getting to the simple present value sought in the analysis. What can be, with the right abstractions, distilled into five lines of code would take hundreds of cells in a spreadsheet. Providing abstractions like this at the hand of financial modelers is a productivity multiplier.

8.4.2.2. Initial Versions

There were two main abstractions to talk about from early versions of the packages.

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% rate 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

8.4. Interfaces

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.

8.4.2.3. The Birth of FinanceModels

There were a multiple of insights that led to a more flexible interface in more recent versions.

8. Higher Levels of Abstraction

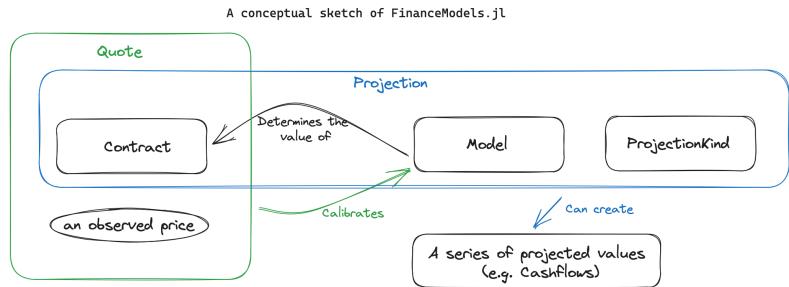


Figure 8.1.: A conceptual sketch of FinanceModels.jl components.

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 simple 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:

8.4. Interfaces

```
risk_free_rates = [0.05,0.06,...0.06]
tenors = [1/12,3/12,...30]
quotes = ParYield.(risk_free_rates,tenors)
model = fit(Spline.Cubic(),quotes,Fit.Bootstrap())

cashflow_vector = [1e6,3e6,...,1e3]
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 for 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 = [
    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.

8. Higher Levels of Abstraction

- ④ Fits the model and derives an approximate volatility of 0.15
- ⑤ Values the contract and in such a simple, noiseless model we recover the original price of 0.0541

With a consistent interface able to handle a wide variety of situations, the modeler is free to expand the model in new directions of analysis with the built in functionality allowing him or her to compose pieces together that was not 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.

²⁶ And 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.

Further, and critically, the new design allows modelers to create their own models or contracts²⁶ 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)`.

8.5. Macros & Homoiconicity

We've talked about transforming data and restructuring logic in order to make the model more effective. We can go still deeper!(Or is it higher level?) We can actually 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 find a need to implement this 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. In other words, 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 be generated or transformed during the compilation process.

Macros are a metaprogramming feature that leverage homoiconicity in Julia. They allow the programmer to write

8.5. Macros & Homoiconicity

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 code²⁷.

```
@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:

```
@time exp(rand())
```

0.000004 seconds

2.2849405502164637

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 a syntax closer to the domain language, which then gets expanded into the underlying implementation code.

²⁷ (`time?`) is a simple, built-in function. For true benchmarking purposes, see Section 30.1.

8. Higher Levels of Abstraction

²⁸ Accessor 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 structs field.

2. Automating boilerplate code: Macros can help reduce code duplication by generating common patterns or boilerplate code. This can include generating accessor functions²⁸, 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.

8.5.2. Commonly Encountered Macros

Table 8.2.: Useful macros for modeling work. There are others related to parallelism which will be covered in [?@sec-parallelization](#).

Macro	Description
<code>BenchmarkTools.@benchmark</code>	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.

8.5. Macros & Homoiconicity

Macro	Description
<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>@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,</code> <code>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>	Encloses a string literal, disabling 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.
<code>@inbounds</code>	Disables bounds checking for array accesses within a block, improving performance but removing safety checks.

8. Higher Levels of Abstraction

Macro	Description
<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.

Part IV.

Conceptual Foundations: Learning from Related Disciplines

9. Hardware and Its Implications

9.1. In this section

A discussion of why a cursory understanding of modern computing hardware and architecture is important for making the right design decisions within a modeling context. Stack vs heap allocations, pointers, and bit types. A discussion of parallelism and the different kinds of parallelism.

10. Elements of Computer Science

“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)

10.1. In this section

Adapting computer science concepts to work for financial professionals. Concepts like computability, computational complexity, the language of algorithms and problem solving.

10.2. 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:

- **Algorithms and Optimization.** How can a problem be solved efficiently? How can that problem be solved *at all*? Given constraints, how can one find an optimal solution?
- **Information Theory.** Given limited data, what *can* be known or inferred from it?
- **Theory of Computation.** What sorts of questions are even answerable? Is an answer easy to computer or will resolving it require more resources than the entire known universe? Will a computation ever stop calculating?
- **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?

For a reader in the twenty-first century we hope that's it's patently obvious how impactful the *applied* computer science has been as an end-user of the internet, artificial intelligence, computational photography, safety control systems, etc., etc. have been to our lives. It is a testament to the utility of being able to harness some of the ideas of this science is. Many of the most impactful advances occur at the boundary between two disciplines. It's here in this chapter that we desire to bring together the financial discipline together with computer science and to provide the financial practitioner with the language and concepts to leverage some of computer science's most relevant ideas.

In this section, we will refer back to a problem called the travelling salesperson problem (TSP).

10.3. Algorithms & Complexity

Algorithms is a general term for a process that transforms an input to an output. It's the dirty, down-to-earth implementation of a mathematical function or process. Further, we should indicate that a process needs to be specified in sufficient detail to be able to call itself an algorithm versus a heuristic which does not indicate with enough detail how the process would unfold.

10.3.1. 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 .

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 multiplication or division. Sometimes this is an important distinction and sometimes not, but for this book we will ignore this level of analysis.

10.3.1.1. 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

10. Elements of Computer Science

and the associated computational complexity for them.

10.3.1.2. 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.

10.3.1.3. 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.

10.3.1.4. Quadratic Time

What if we were less efficient, and instead said that the operation $n + 42$ was to be implemented not as the basic addition of two numbers, but that we should *add one to n* forty-two times? That is, we'll see that we add a second loop which increments our result by a unit instead of simply adding the current i to the running total result:

```
function nsum_quadratic(n)
    result = 0
    for i in 1:n
        for j in 1:i
            result += 1
        end
    end
    result
end
```

- ① The outer loop with iterator i .
- ② The inner loop with iterator j .

`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 $1 + \dots + (n-2) + (n-1) + n$ steps to complete. We actually know that this simplifies down to our constant time formula $n * (n + 1) \div 2$ or $n^2 + n \div 2$ steps to complete.

10.3.1.5. Comparison

10.3.1.5.1. Big-O Notation

²⁹ “Big-O”, so named because of the “O” in used in $O(1)$. $O(n)$, etc. Not one of the sciences’ more creative names.

We can categorize the above implementations using a convention called **Big-O Notation**²⁹ 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 10.1 shows for the examples constructed above what the description, order, and order of magnitude complexity is.

Table 10.1.: Complexity comparison for the three sample cases of summing integers from 1 to n .

Function	Computation Cost	Complexity Description	Big-O Order	Steps ($n = 10,000$)
<code>nsum_const</code>	fixed	Constant	$O(1)$	~1
<code>nsum_linear</code>	n	Linear	$O(n)$	~10,000
<code>nsum_quadratic</code>	n^2	Quadratic	$O(n^2)$	~100,000,000

Table 10.2 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 salesman problem 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 salesman 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 assets to optimize a portfolio).

10.3. Algorithms & Complexity

Table 10.2.: 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
$O(n)$	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	7	14
$O(n \times \log(n))$	Linearithmic Time	30	7,000	14,000,000
$O(2^n)$	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 million, then we see that it will result in:

$$n^2 + n \div 2(10^6)^2 + 10^6 \div 2(10^{12}) + 5^6$$

10^{12} is significantly more important than 5^6 (sixty-four million times as important, to be precise).

Conversely, if n is small then we don't really care about computational complexity in general. This is why Big-O notation reduces the problem down to only the most significant complexity cost term.

10.3.1.5.2. Empirical Results

10. Elements of Computer Science

```
using BenchmarkTools
@btime nsum_constant(10_000)

0.666 ns (0 allocations: 0 bytes)

50005000

@btime nsum_linear(10_000)

1.166 ns (0 allocations: 0 bytes)

50005000

@btime nsum_quadratic(10_000)

2.398 μs (0 allocations: 0 bytes)

50005000
```

The preceding examples of constant, linear, and exponential 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.

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 avoid 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 5.5.3.4).

10.3.2. Expected versus worst-case complexity

Another consideration is that there may be one approach which 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.

10.3.3. 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 and when searching for additional performance to look for the loops-within-loops or combinatorical explosions. It's often at this level, rather than micro-optimizations, that you can transform the performance of the overall model (unless the fundamental complexity of the problem at hand forbids it).

10.4. Data Structures

Data structures is the art and science of how to represent data in discrete objects. There are many common kinds and many specialized sub-kinds, and 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.

10.4.1. 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, and would look like this (with the different integers shown on new lines for clarity):

10. Elements of Computer Science

³⁰ In 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.

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 memory³⁰, 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.

10.4.2. 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 could be implemented as:

```
mutable struct Node
    value::Any
    next::Union{Node, Nothing}
end

z = Node(3,Nothing)
y = Node(2,z)
x = Node(1,y)
```

① Here, 'Nothing' would represent the end of the linked list.

Inserting a new node between existing nodes is efficient - if we wanted to inser a new node between the ones with value 2 and 3, we could do this:

```
a = Node(0,z)                                ①  
y.next = a                                    ②
```

10.4. Data Structures

However, accessing the n th element requires traversing the list from the beginning, making it $O(n)$ time complexity for random access. Also, if you have an intermediate node such as y , y itself does not know about x so there's no way to move 'up' the list to get to previous values.

10.4.3. Records/Structs

An aggregate of named fields, typically of fixed size and sequence. Records group related data together. We've encountered structs in Section 5.5.7, but here we'll add 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

10. Elements of Computer Science

In contrast, the `LessSimpleBond` uses a `String` to represent the ID of the bond. Strings are essentially arrays of containers, and the arrays themselves are mutable containers which is by definition not a constant set of bits. In memory, `b` would look like:

- 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 21 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.

10.4.4. Dictionaries (Hash Tables)

10.4.4.1. 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

10.4. Data Structures

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 a type of hash called an SHA hash on any data:

```
import SHA
let
    a = SHA.sha256("hello world") ▷ bytes2hex
    b = SHA.sha256(rand(UInt8, 10^6)) ▷ bytes2hex
    println(a)
    println(b)
end
```

b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9
08fbf19c5063d666e6c0885abda5fb47f2a3074b7eb414abbc9913a6b58deb9d

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.

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, maybe you want to see if two data files with different names contain the same data - comparing the hashes is a sure way to determine if they contain the same data.

10.4.4.2. 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, the more robust the collision handling means higher fixed cost for access).

10.4.5. 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 various real-world scenarios such as social networks, transportation systems, or computer networks.

In Julia, a simple graph could be implemented using a dictionary where keys are nodes and values are lists of connected nodes:

```
struct Graph
    nodes :: Dict{Any, Vector{Any}}
end

function add_edge!(graph::Graph, node1, node2)
    push!(get!(graph.nodes, node1, []), node2)
    push!(get!(graph.nodes, node2, []), node1)
end

g = Graph(Dict())
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.

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.

10.4.6. 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

10.4. Data Structures

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
    value::Any
    left::Union{TreeNode, Nothing}
    right::Union{TreeNode, 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.
- 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

10. Elements of Computer Science

trees. They are fundamental in many algorithms and data structures, including heaps, syntax trees in compilers, and decision trees in machine learning.

10.4.7. Data Structures Conclusion

Data structures have strengths and weakness 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.

10.5. 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.

10.5.1. Basic Concept

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.

10.5.2. 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. This is 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: Actually reasoning through how the system should behave in all scenarios requires actually 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.

10.5.3. Related Topics

10.5.3.1. Property Based Testing

Testing will be discussed in more detail in Chapter 11, 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:

10. Elements of Computer Science

```
using Test

myadd(a,b) = a + b

@test myadd(1,2) == myadd(2,1)
@test myadd(myadd(1,2),3) == myadd(1,myadd(2,3))
```

However, we really haven't proven the associative and commutative properties in general. There are techniques to do this, which is a more comprehensive alternative to testing specific examples above. Packages like Supposition.jl provide functionality for this. Note that like Formal Verification, property-based testing is a more advanced topic.

10.5.3.2. Fuzzing

Fuzzing is kind of like property based testing, but instead of testing general rules, we generalize the simple examples using randomness. For example, we could test the commutative property using random numbers instead, therefore 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.

11. Applying Software Engineering Principles

“Programs must be written for people to read, and only incidentally for machines to execute.” — Harold Abelson and Gerald Jay Sussman (1984)

11.1. In this section

We describe modern software engineering practices such as version control, testing, documentation, and pipelines which can be utilized by the financial professional to make their own work more robust and automated. Data practices and workflow advice.

11.2. Testing

Note some more advanced testing topics in Section 10.5.3.

12. Statistical Inference and Information Theory

"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)

12.1. In This Chapter

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.

12.2. 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 something.

12. Statistical Inference and Information Theory

Let's consider the following number that we encounter while reading a report which contains estimates of total amount of assets held. Unfortunately, for one reason or another one of the digits is not visible to you. Here's what you can read, with the $\underline{}$ indicating that the digit is not visible:

32,000, $\underline{0}$ 0

Now you probably already formed an opinion on what the missing number is, but let's look at how we can quantify the analysis.

Given that we know the number was an estimate and the tendency of humans to like 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 12.1. 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 as³¹:

$$h(x_i) = \log_2 \frac{1}{p(x_i)} \quad (12.1)$$

So if we were to find out that the missing digit were indeed 0, we have gained less information relative to our expectation than if the missing digit were anything other than 0 .

We can characterize the entire distribution X via the **entropy**, $H(X)$, of a probability set is the ensemble's average information content:

$$H(X) = \sum p(x_i) \log_2 \frac{1}{p(x_i)} \quad (12.2)$$

The entropy $H(X)$ of the presumed outcomes in Table 12.1 distribution of outcomes is 0.722bits.

12.2. Information Theory

Table 12.1.: Probability distribution of missing digit, knowing the human inclination to prefer round numbers when estimating.

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

Note that we have take a view on the probability distribution for the missing digit, and we'll refer to this as the **prior assumption** (or just **prior**). This is an opinionated assumption, so what if we had another colleague who believed humans are completely rational and without bias for certain numbers. They would then be arguing for a prior assumed distribution consistent with Table 12.2.

With the uniform prior assumption, $H(X) = 3.322$ bits and $h(x_i)$ is also uniform. We will not prove it here, but a uniform probability over a set of outcomes is the highest entropy distribution that can be assumed.

Table 12.2.: 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 an estimate), 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

12. Statistical Inference and Information Theory

Table 12.3.: 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

problem at hand. It is important to carefully consider and justify the prior assumptions used in information-theoretic and statistical analyses.

12.2.1. Example: Classification

³² A decision tree is a classification algorithm which attempts to optimally classify an output based on if/else type branches on the input variables.

In this example, we will determine the optimal splits for a decision tree³² 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, false, true]
default = [true, false, true, true, true, false, true]
default_data = DataFrame(; employed, good_credit, default)
```

[Info: Precompiling DataFrames [a93c6f00-e57d-5684-b7b6-d8193f3e460]

The entropy of the default rate data is, per Equation 12.2:

```
H0 = let
    p1 = sum(default_data.default) / nrow(default_data)
    p2 = 1 - p1
```

12.2. Information Theory

```
p1 * log2(1 / p1) + p2 * log(1 / p2)  
end
```

0.6578517147391054

Our goal is to determine which attribute (`employed` or `good_credit`) to use as the first split in the decision tree. We will decide this by calculating the information gain, which is the difference in entropy between the prior node and the candidate node. 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. **Information gain**, $IG(inputs, attributes)$, is:

...

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)
```

12. Statistical Inference and Information Theory

	employed	good_credit	default
	Bool	Bool	Bool
1	0	1	0
2	0	0	1
3	0	0	0

let's call it's entropy H_{employed} , which should be zero because there is no variability in the `default` outcome for this subset.

```
H_employed = let
    p1 = sum(df_employed.default) / nrow(df_employed)
    p2 = 1 - p1
    # p1 * log2(1 / p1) + p2 * log(1 / p2)
    p1 * log2(1 / p1) + 0
end
```

- 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
    p1 = sum(df_unemployed.default) / nrow(df_unemployed)
    p2 = 1 - p1
    p1 * log2(1 / p1) + p2 * log(1 / p2)
end
```

0.7986309056458281

The average of the two is weighted by the size of the data 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
```

12.2. Information Theory

0.29948658961718555

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`

0.35836512512191987

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` (0.358) is the majority of the overall entropy H_0 (0.658). Entropy is always additive and you cannot have negative entropy, therefore no other other attribute could have greater information gain. This also matches our intuition when looking at Table 12.3 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.

12.2.2. 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 distributions**. Some discussion of maximum entropy distributions in the context of risk assessment is available in an article by Duracz³³. probability distributions and risk asses

³³ https://www.researchgate.net/publication/239752412_Derivation_of_Probability_Distributions_for_Risk_Assessment

12. Statistical Inference and Information Theory

Table 12.4.: Maximum Entropy Distributions and the conditions under which they are applicable.

Constraint	Discrete Distribution	Continuous Distribution
Bounded range	Uniform (discrete)	Uniform (continuous)
Bounded range (0 to 1) with information about the mean or variance		Beta
Mean is finite, two possible values		Binomial
Mean is finite and positive	Geometric	Exponential
Mean is finite and range is > zero		Gamma
Mean and Variance is finite		Guassian (Normal)
Positive and equal mean and variance	Poisson	

The distributions in Table 12.4 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 (random) processes that maximize entropy pop up all over the place. As an example, let's look at processes that behave like the Gaussian (Normal) distribution.

12.2. Information Theory

12.2.2.1. 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 is 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.

Table 12.5.: Underlying processes create typical probability distributions. That there is significant overlap with the distributions in Section 12.2.2 is not a coincidence.

Process	Distribution of Data	Examples
Many <i>additive</i> pluses and minus 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 minus 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

12. Statistical Inference and Information Theory

Process	Distribution of Data	Examples
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 sim-

ply 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.
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 a wacky dataset you don't need to comb the depths of Wikipedia to find the perfect probability distributions.

12.2.2.2. Additive and Multiplicative Processes

Table 12.5 describes some examples, let us discuss further what it means to have a process that arises via an additive vs multiplicative effect³⁴.

An outcome is additive if it's the sum or difference of multiple independent processes. One of the simplest examples of this is rolling multiple dice and taking their sum. Or a random walk along the natural numbers wherein with equal probability you take a step left or right. The distribution of the position after n steps converges rapidly to a normal distribution. Another common one is when you are looking at the mean of a sample - since you are summing up the individual measurements you end up with a normal distribution (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.

³⁴ Multiplicative processes are often referred to as "geometric", as in "geometric Brownian motion" or "geometric mean". Additive processes are sometimes referred to as "arithmetic". This 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.

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.

Bringing this back to 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.

12.3. Bayes' Rule

The minister and statistician Thomas Bayes derived a relationship of conditional probabilities that we today know as **Bayes' Rule**, 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 12.4. First, let's look at combining information theory and Bayes' rule in an applied example.

12.3.1. Example: Model Selection via Likelihoods

Let's say that we have competing hypothesis about a data generating process, such as: "given a set of data representing risk outcomes, what distribution best fits the data"?

12. Statistical Inference and Information Theory

We can compare these models using Bayes' rules 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: $\$ P(H_1|D) = (P(D|H_1) * P(H_1)) / P(D)$

$$P(H_2|D) = (P(D|H_2) * P(H_2)) / P(D) \$$$

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 likelihoods of the two models, we can calculate the ratio of their posterior probabilities, known as the Bayes factor, BF :

$$BF = \frac{P(H_1|D)}{P(H_2|D)}$$

Substituting the expressions for the posterior probabilities from Bayes' rule, we get:

$$BF = \frac{P(D|H_1) \times P(H_1)}{P(D|H_2) \times P(H_2)}$$

The marginal likelihood $P(D)$ cancels out since it appears in both the numerator and denominator. If we assume equal prior probabilities for the models, i.e., $P(H_1) = P(H_2)$, then the Bayes factor simplifies to the likelihood ratio:

$$BF = \frac{P(D|H_1)}{P(D|H_2)}$$

The interpretation of the Bayes factor is as follows:

12.3. Bayes' Rule

- If $BF > 1$, the data favor H_1 over H_2 .
- If $BF < 1$, the data favor H_2 over H_1 .
- If $BF = 1$, the data do not provide evidence in favor of either model.

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 more simplistic evaluation of which model has the higher likelihood given the data: this is simply a matter of comparing the magnitude of the likelihoods.

⚠ 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 concept, the NHST choices of both null hypothesis and model contain significant amounts of subjectivity! 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:

12. Statistical Inference and Information Theory

- “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 (p-value 0.0001) or a study indicating a 1 in 3 chance (p-value 0.06)? Many journals would only publish the former study.
- Difficulty to determine *causal* relationships.

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.

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).

³⁵ <https://data.ca.gov/dataset/annual-precipitation-data-for-northern-california-1944-current>

³⁶ See @sec-predictive-vs-explanatory.

The example we'll look at relates to the annual rainfall totals for a specific location in California³⁵, 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 model³⁶ 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

12.3. Bayes' Rule

recommend a model that best fits the data and justify that recommendation quantitatively. Before even looking at the data, Table 12.6 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 12.6.: 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	Since rainfall totals would be the sum of exponentially-distributed independent rainfall events

i Note

In the literature, H_3 (the Gamma distribution) is known as the "Log-Pearson Type III distribution". It's actually recommended by the US Corps of Army Engineers as the recommended way to model rainfall totals.

```
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,
```

12. Statistical Inference and Information Theory

```
13.46, 9.34, 16.51, 48.24  
];
```

When we do plot it, we can 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)
```

```
Warning: Found 'resolution' in the theme when creating a 'Scene'. Th  
@ Makie ~/.julia/packages/Makie/iRM0c/src/scenes.jl:220
```

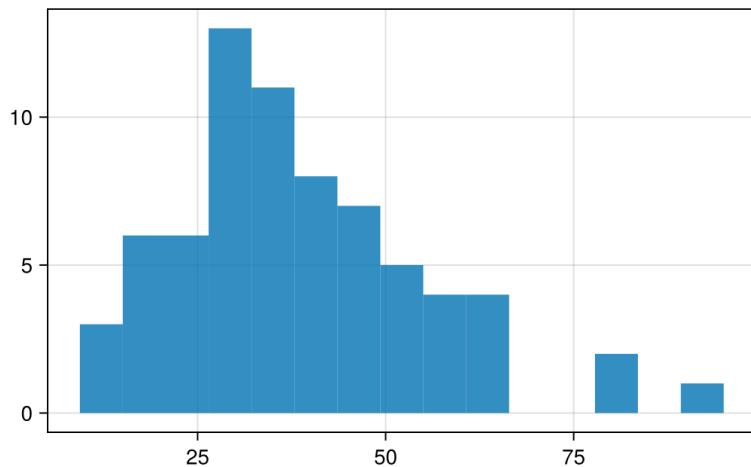


Figure 12.1.: Annual rainfall totals for a specific location in California.

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`:

12.3. Bayes' Rule

```
using StatsBase
using Distributions

n = fit_mle(Normal, rain)
ln = fit_mle(Normal, log.(rain))
lg = fit_mle(Gamma, log.(rain))
@show n
@show ln
@show lg;

n = Normal{Float64}(\mu=38.91442857142857, σ=16.643603630714306)
ln = Normal{Float64}(\mu=3.5690550009062663, σ=0.44148379736539156)
lg = Gamma{Float64}(α=61.58531301458412, θ=0.05795302201453571)

let x = rain

range = 1:0.1:100
fig, ax, _ = lines(range, cdf.(n, range), label="Normal", axis=(xgridvisible=false, ygridvisible=false))
lines!(ax, range, cdf.(ln, log.(range)), label="LogNormal")
lines!(range, cdf.(lg, log.(range)), label="LogGamma")
lines!(quantile.(Ref(x), 0.01:0.01:0.99), 0.01:0.01:0.99, label="Data", color(:black, 0.6),
fig[1, 2] = Legend(fig, ax, "Model", framevisible=false)
fig
end

Warning: Found 'resolution' in the theme when creating a 'Scene'. The 'resolution' keyword for 'Scene'
@ Makie ~/.julia/packages/Makie/iRM0c/src/scenes.jl:220
```

12. Statistical Inference and Information Theory

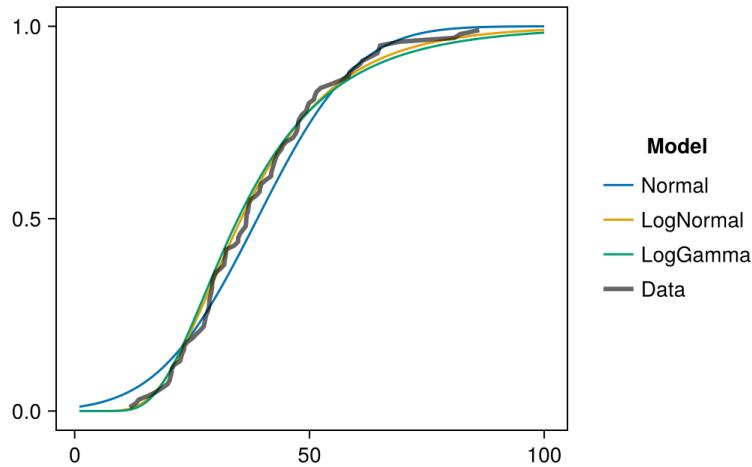


Figure 12.2.

Let's look at the likelihoods. 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.

```

let
    n_lik = sum(log.(pdf.(n, rain)))
    ln_lik = sum(log.(pdf.(ln, log.(rain))))
    lg_lik = sum(log.(pdf.(lg, log.(rain))))

    @show n_lik
    @show ln_lik
    @show lg_lik
end;

n_lik = -296.1675156647812
ln_lik = -42.09272021737914
lg_lik = -43.79151806348801

```

The results indicate that the LogNormal and the Gamma model for rainfall distribution are very superior to the Normal model, consistent with the visual inspection of the quantiles

12.4. Modern Bayesian Statistics

in Figure 12.2. We reach that conclusion by noting how much more likely the latter two are, as the likelihoods of -42 and -44 is much greater than -296^{37} .

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. Expanding the analysis to account for that point will be the focus of the remainder of this chapter.

³⁷ The 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.

12.4. Modern Bayesian Statistics

12.4.1. 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. Why is the applied practice of statistics then gravitating towards Bayesian approaches? There are both philosophical and practical reasons why.

Philosophically, one of the main reasons why Bayesian thinking is appealing is its ability to provide a straightforward interpretation 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.

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

12. Statistical Inference and Information Theory

randomness. This perspective pertains to whether you regard the data being random or the parameters being random.

Frequentist statistics treats parameters as fixed and unknown, and the data as random — this is reflective of the view 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.

Practically, recent advances in computational power, algorithm development, and open-source libraries have enabled practitioners to adapt the Bayesian workflow.

Deriving the posterior distribution is analytically intractable so computational methods must be used. Advances in raw computing power only in the 1990's 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 2010's.

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 insurance companies trying to manage risk!

The availability of open-source libraries, such as Turing.jl, PyMC3, and Stan provide access to the core routines in an accessible interface. The exercise remains undoubtedly one

12.4. Modern Bayesian Statistics

that benefits from the computational thinking described in this book - understanding model complexity, model transformations and structure, data types and program organization, etc.

12.4.1.1. 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 understate the risk distribution.
2. **Model flexibility.** A Bayesian model can be as simple as an ordinary linear regression, but as complex as modeling a 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.

12.4.1.2. 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.

12.4.1.3. 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.

12.4.1.4. Subjectivity as a Flaw

Subjectivity is inherent in all useful statistical methods. Subjectivity in traditional approaches include how the data was collected, which hypothesis to test, what significant 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 credibility¹ and therefore should be similarly reconsidered.

12.4.2. Implications for Financial Modeling

Like Bayes' Formula itself, another aspect of actuarial literature that is taught but often glossed over in practice is the difference between process risk (volatility), parameter risk, and model formulation risk. Often when performing analysis that relies on stochastic result, in practice only process/volatility risk is assessed.

Bayesian statistics provides the tools to help actuaries 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 formulation 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:

¹Note that the approach discussed here is much more encompassing than the Bühlmann-Straub Bayesian approach described in the actuarial literature.

- Causal Modeling: Identifying not just correlated relationships, but causal ones, in contexts where a traditional 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.

12.4.3. 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

Having defined the first two components and collected our data, 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.

12.4.4. Markov-Chain Monte Carlo

While computing the posterior distribution for most model parameters is analytically intractable, 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

12.4. Modern Bayesian Statistics

and 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.

Here is a simple example demonstrated with one of the oldest MCMC algorithm, 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.

Here is what a complete example looks like. We will try to find the posterior of an arbitrary 2D density function. Picking the product of an Exponential and Beta distribution looks like this:

```
target_dist = product_distribution(Exponential(1.),Beta(2.,15.))

# plot the target_distirubution
xs = LinRange(-0.5, 3, 400)
ys = LinRange(0, 3, 400)
zs = [pdf(target_dist,[x,y]) for x in xs, y in ys]
contour(xs, ys, zs)
```

```
[ Warning: Found 'resolution' in the theme when creating a 'Scene'. The 'resolution' keyword for 'Scene'
└ @ Makie ~/.julia/packages/Makie/iRM0c/src/scenes.jl:220
```

12. Statistical Inference and Information Theory

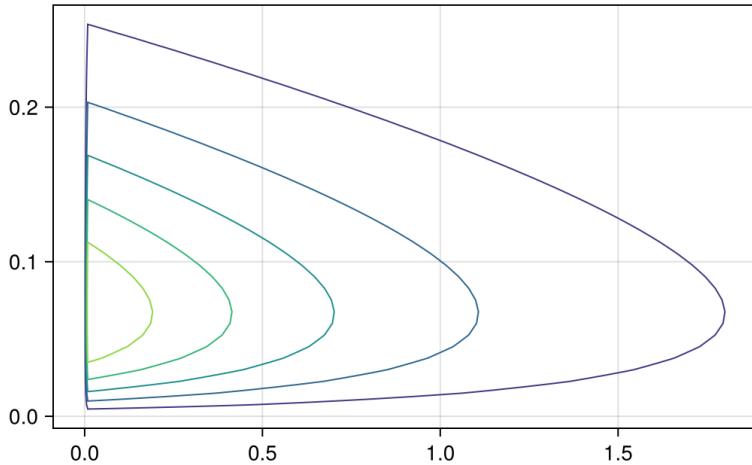


Figure 12.3.: The target probability density which we will attempt to infer via MCMC.

We will next define a probability distribution for the random step that we take from the `current_point`. We choose a 2D Gaussian for this. The `proposal_std` controls how big of a movement is taken at each step.

```
# Define the proposal distribution (2D Gaussian with diagonal covariance)
proposal_std = 0.025
proposal_dist = MvNormal(proposal_std * ones(2))

ZeroMeanDiagNormal(
  dim: 2
  μ: Zeros(2)
  Σ: [0.00062500000000000001 0.0; 0.0 0.00062500000000000001]
)
```

We next define how many steps we want the chain to sample for, and implement the algorithm's main loop containing the logic described above. 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 be effectively independent of both (1) the starting point for the sample, and (2) so that different chains are effectively independent.

12.4. Modern Bayesian Statistics

```
# MCMC parameters
num_samples = 10000
burn_in = 500

# Initialize the Markov chain
current_state = [0.0, 0.0]
chain = zeros(num_samples, 2)

# MCMC sampling loop
for i in 1:num_samples
    # Generate a new proposal
    proposal = rand(proposal_dist) + current_state

    # Calculate the acceptance ratio
    current_prob = pdf(target_dist, current_state)
    proposal_prob = pdf(target_dist, proposal)
    acceptance_ratio = proposal_prob / current_prob

    # Accept or reject the proposal
    if rand() < acceptance_ratio
        current_state = proposal
    end

    # Store the current state as a sample
    chain[i, :] = current_state
end

chain

10000×2 Matrix{Float64}:
 0.0      0.0
 0.0      0.0
 0.0318715  0.0203549
 0.000316927  0.0255503
 0.000316927  0.0255503
 0.000316927  0.0255503
 0.0267271   0.066587
 0.0267271   0.066587
 0.0193473   0.0485052
 0.0193473   0.0485052
```

12. Statistical Inference and Information Theory

```
0.00921787  0.0320062
0.00921787  0.0320062
0.00921787  0.0320062
:
0.758633    0.0839344
0.725751    0.0844488
0.712897    0.0519065
0.647975    0.0946723
0.655023    0.0892262
0.625028    0.0937472
0.651585    0.0583124
0.651043    0.0628776
0.620825    0.0323705
0.658583    0.0535995
0.675191    0.0377583
0.680616    0.0320376
```

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()
    ax = Axis(f[1, 1])

    contour!(ax,xs, ys, zs)
    lines!(ax,Point2f.(eachrow(chain[1:burn_in,:])),color=(:red, 0
```

```

for i in burn_in:num_samples
    ps = Point2f.(eachrow(chain[i-1:i,:]))
    lines!(ax,ps,color=(:blue,0.2))
end
f
end

```

- ① Iterate through pairs of points to plot as separate line segments so that the transparent blue line visually stack and the higher density near the center is apparent.

[Warning: Found 'resolution' in the theme when creating a 'Scene'. The 'resolution' keyword for 'Scene'

└ @ Makie ~/.julia/packages/Makie/iRM0c/src/scenes.jl:220

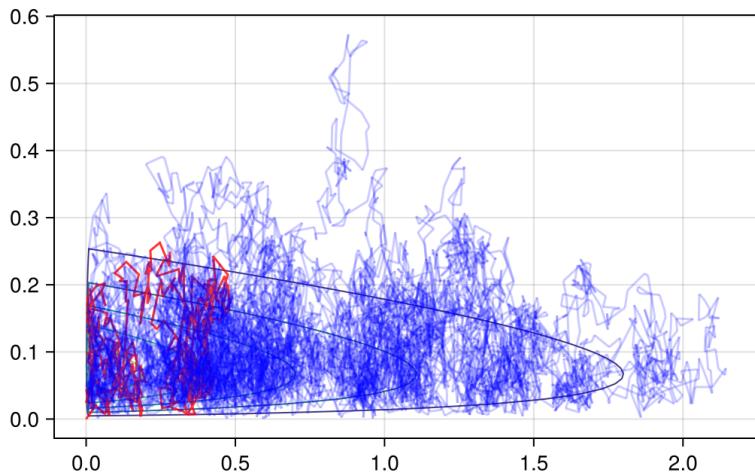


Figure 12.4.: The blue lines of the MCMC chain explore the posterior density of interest (after discarding the burn-in samples in red). Because of the sharp boundary at $x = 0$ right next to the highest density region, this will sample somewhat inefficiently since when the current state is near $x = 0$, it becomes increasingly likely that any proposal that moves further to the left will be rejected.

12.4.5. MCMC Algorithms

The Metropolis-Hastings algorithm is simple, but somewhat inefficient. Some challenges with MCMC sampling are both mathematical and computational:

1. Often times 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 regions.
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 problems 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: much falls back onto the 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.

i Note

What does it mean to transform the parameter space? An example will be shown in Chapter 25 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.

12.4.6. Rainfall Example (Continued)

We will construct a Bayesian model using the `Turing.jl` library, and fit the parameters of one of the competing models 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 either relate a parameter to a prior assumptions.
3. A loop (or broadcasted `.~`) that ties specific data observations to the random process.

Think of the `@model` block really as a model *constructor*. When you pass data to the model, then you get an instantiated `Model` type³⁸.

```
using Turing
```

```
@model function rainLogNormal(logdata) ①
    # Prior Assumptions for the (Log) Normal Parameters
    μ ~ Normal(4,1) ②
```

³⁸ Specifically: a `DynamicPPL.Model` type (PPL = Probabilistic Programming Language).

12. Statistical Inference and Information Theory

$\sigma \sim \text{Exponential}(0.5)$

(3)

```
# Link observations to the random process
for i in 1:length(logdata)
    logdata[i] ~ Normal(μ, σ)
end
end

m = rainLogNormal(log.(rain));
```

- ① Defining the model uses the @model macro from Turing.
- ② We know that there will be positive rainfall and 96% of mean annual rainfall will be 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.

```
[ Info: Precompiling Turing [fce5fe82-541a-59a6-adf8-730c64b5f9a0]
[ Info: Precompiling IntervalArithmeticDiffRulesExt [86439bda-9ede-5
[ Info: Precompiling IntervalArithmeticForwardDiffExt [ba47a815-ec9a-
[ Info: Precompiling BangBangDataFramesExt [d787bcad-b5c5-56bb-adaa-
[ Info: Precompiling TransducersDataFramesExt [cefb4096-3352-5e5f-85
[ Info: Precompiling IntervalArithmeticRecipesBaseExt [e3b91bd4-2888
Precompiling SciMLBaseMakieExt
  ✓ SciMLBase → SciMLBaseMakieExt
  1 dependency successfully precompiled in 4 seconds. 299 already prec
[ Info: Precompiling SciMLBaseMakieExt [565f26a4-c902-5eae-92ad-e10
[ Warning: Module SciMLBaseChainRulesCoreExt with build ID fafbfccfd-5
  This may mean SciMLBaseChainRulesCoreExt [4676cac9-c8e0-5d6e-a4e0-
  @ Base loading.jl:1948
Error: Error during loading of extension SciMLBaseChainRulesCoreExt
exception =
  1-element ExceptionStack:
  Declaring __precompile__(false) is not allowed in files that are b
  Stacktrace:
  [1] __require(pkg::Base.PkgId, env)::Nothing
      @ Base ./loading.jl:1952
  [2] __require_prelocked(uuidkey::Base.PkgId, env)::Nothing
      @ Base ./loading.jl:1812
  [3] #invoke_in_world#3
```

12.4. Modern Bayesian Statistics

```
@ ./essentials.jl:926 [inlined]
[4] invoke_in_world
    @ ./essentials.jl:923 [inlined]
[5] _require_prelocked
    @ ./loading.jl:1803 [inlined]
[6] _require_prelocked
    @ ./loading.jl:1802 [inlined]
[7] run_extension_callbacks(extid::Base.ExtensionId)
    @ Base ./loading.jl:1295
[8] run_extension_callbacks(pkgid::Base.PkgId)
    @ Base ./loading.jl:1330
[9] run_package_callbacks(modkey ::Base.PkgId)
    @ Base ./loading.jl:1164
[10] _tryrequire_from_serialized(modkey ::Base.PkgId, path::String, ocachepath::String, sourcepath::String)
    @ Base ./loading.jl:1487
[11] _require_search_from_serialized(pkg ::Base.PkgId, sourcepath::String, build_id::UInt128)
    @ Base ./loading.jl:1574
[12] _require(pkg ::Base.PkgId, env ::String)
    @ Base ./loading.jl:1938
[13] __require_prelocked(uuidkey ::Base.PkgId, env ::String)
    @ Base ./loading.jl:1812
[14] #invoke_in_world#3
    @ ./essentials.jl:926 [inlined]
[15] invoke_in_world
    @ ./essentials.jl:923 [inlined]
[16] _require_prelocked(uuidkey ::Base.PkgId, env ::String)
    @ Base ./loading.jl:1803
[17] macro expansion
    @ ./loading.jl:1790 [inlined]
[18] macro expansion
    @ ./lock.jl:267 [inlined]
[19] __require(into ::Module, mod ::Symbol)
    @ Base ./loading.jl:1753
[20] #invoke_in_world#3
    @ ./essentials.jl:926 [inlined]
[21] invoke_in_world
    @ ./essentials.jl:923 [inlined]
[22] require(into ::Module, mod ::Symbol)
    @ Base ./loading.jl:1746
[23] include
    @ ./Base.jl:495 [inlined]
```

12. Statistical Inference and Information Theory

```
[24] include_package_for_output(pkg::Base.PkgId, input::String,
    @ Base ./loading.jl:2222
[25] top-level scope
    @ stdin:3
[26] eval
    @ ./boot.jl:385 [inlined]
[27] include_string(mapexpr::typeof(identity), mod::Module, cod
    @ Base ./loading.jl:2076
[28] include_string
    @ ./loading.jl:2086 [inlined]
[29] exec_options(opts::Base.JLOptions)
    @ Base ./client.jl:316
[30] _start()
    @ Base ./client.jl:552
@ Base loading.jl:1301
```

12.4.6.1. 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 `Uniform(0, Inf)` distribution of mean log-rainfall total. We admit a level of subjectivity when we encode into the model some limitations on what we would believe to be true. 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!

“Strongly informative” priors would be something 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.

“Uninformative” priors use only maximum entropy or uniform priors to avoid encoding bias into the model.

12.4.6.2. 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 12.8 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.

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)
```

Sampling: 63% |██████████|

| ETA: 0:00:00 Sampling: 100% |██████████|

We now 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)
```

```
[ Info: Found initial step size
  ε = 0.025
```

12.4.6.3. 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:

12. Statistical Inference and Information Theory

```
Chains MCMC chain (1000×3×1 Array{Float64, 3}):
```

```
Iterations      = 1:1:1000
Number of chains = 1
Samples per chain = 1000
Wall duration    = 1.06 seconds
Compute duration = 1.06 seconds
parameters       = μ, σ
internals         = lp
```

```
Summary Statistics
```

parameters	mean	std	mcse	ess_bulk	ess_tail	rhat
Symbol	Float64	Float64	Float64	Float64	Float64	Float64
μ	3.9464	1.0167	0.0329	955.0502	1025.7469	1.0036
σ	0.4870	0.4786	0.0156	1000.5271	874.2977	0.9993

1 column omitted

```
Quantiles
```

parameters	2.5%	25.0%	50.0%	75.0%	97.5%
Symbol	Float64	Float64	Float64	Float64	Float64
μ	1.9407	3.2857	3.9566	4.6473	5.9199
σ	0.0152	0.1523	0.3433	0.6538	1.6975

Figure 12.5.: Model output for the sampled prior. This isn't running an MCMC algorithm, it's simply taking draws from the defined prior assumptions.

12.4. Modern Bayesian Statistics

Chains MCMC chain (1000×14×1 Array{Float64, 3}):

```

Iterations      = 501:1:1500
Number of chains = 1
Samples per chain = 1000
Wall duration    = 2.62 seconds
Compute duration = 2.62 seconds
parameters       =  $\mu$ ,  $\sigma$ 
internals        = lp, n_steps, is_accept, acceptance_rate, log_density, hamiltonian_energy, hamiltonian

Summary Statistics
parameters   mean     std     mcse    ess_bulk    ess_tail    rhat    ...
Symbol      Float64  Float64  Float64  Float64  Float64  Float64  ...
 $\mu$           3.5708  0.0519  0.0017  884.6204  626.0803  1.0013  ...
 $\sigma$          0.4510  0.0399  0.0013  1041.1795  751.8330  0.9993  ...
                                         1 column omitted

Quantiles
parameters   2.5%    25.0%    50.0%    75.0%    97.5%
Symbol      Float64  Float64  Float64  Float64  Float64
 $\mu$           3.4687  3.5350  3.5709  3.6058  3.6764
 $\sigma$          0.3777  0.4236  0.4510  0.4747  0.5343

```

Figure 12.6.: Model output for the sampled posterior.

12. Statistical Inference and Information Theory

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. Therefore 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     884.6204    338.1577
σ     1041.1795   398.0044
```

Second, the `rhat` (\hat{R}) is the Gelman-Rubin convergence diagnostic and it's 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     1.0013
σ     0.9993
```

Next, we can look at the “trace” plots for the parameters being sampled (Figure 12.7). 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

```

Warning: Found 'resolution' in the theme when creating a 'Scene'. The 'resolution' keyword for 'Scene'

@ Makie ~/.julia/packages/Makie/iRM0c/src/scenes.jl:220

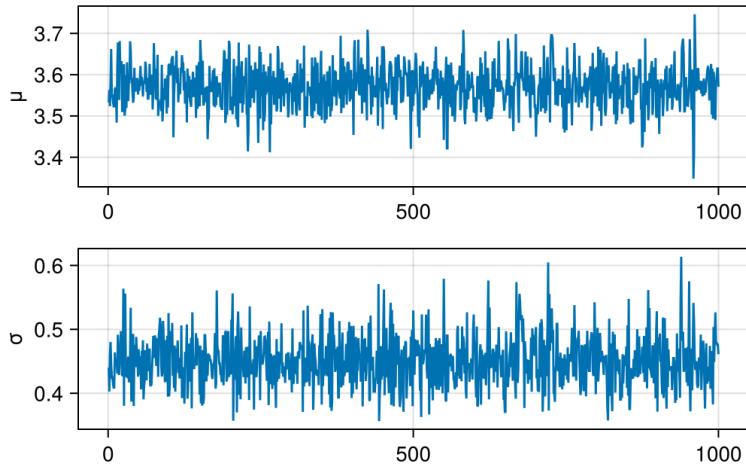


Figure 12.7.: The trace plots indicate low autocorrelation which is desirable for an MCMC sample.

The `ess`, `rhat`, and trace plots all look good for our sampled chain so we can next we will analyze the results proper.

12.4.6.4. Analysis

Let's see how it looks compared to the data first. Figure 12.8 shows 200 samples from the prior and posterior. The prior (top) shows how wide the range of possible rainfall outcomes

12. Statistical Inference and Information Theory

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(s, [:μ, :σ])
    ds = Normal.(vals.μ, vals.σ) # ,get(s,:σ)[i])
    rg = 1:200
    for (i, d) in enumerate(ds)
        lines!(axis, rg,cdf.(d,log.(rg)),color=(:gray,0.3))
    end

    # plot the actual data
    percentiles= 0.01:0.01:0.99
    lines!(axis,quantile.(Ref(rain),percentiles),percentiles,linew...
end

let
    f = Figure()
    ax1 = Axis(f[1,1],title="Prior", xgridvisible=false, ygridvisi...
    chn_cdf!(ax1,chain_prior,rain)

    ax2 = Axis(f[2,1],title="Posterior", xgridvisible=false, ygrid...
    chn_cdf!(ax2,chain_posterior,rain)

    linkxaxes!(ax1, ax2)

    f
end
```

```
└ Warning: Found 'resolution' in the theme when creating a 'Scene'. Th...
└ @ Makie ~/.julia/packages/Makie/iRM0c/src/scenes.jl:220
```

12.4. Modern Bayesian Statistics

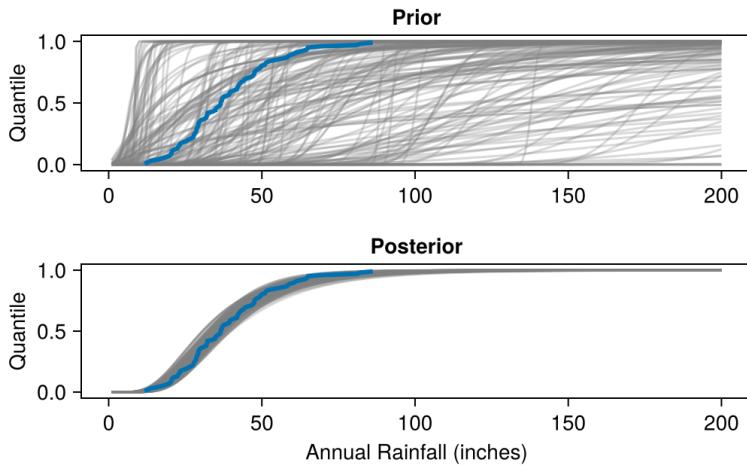


Figure 12.8.: The fitted posterior model (bottom) has good coverage of the observed data (shown in blue).

Comparing to the maximum likelihood analysis from before by plotting the MLE point estimate onto the marginal densities in Figure 12.9. The peak of the 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(ln)

    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

Warning: Found `resolution` in the theme when creating a `Scene`. Th
@ Makie ~/.julia/packages/Makie/iRM0c/src/scenes.jl:220

```

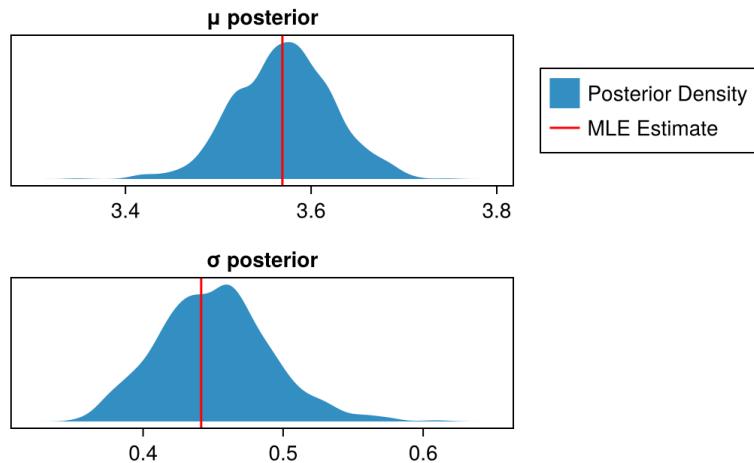


Figure 12.9.: The MLE point estimate does not necessarily align with the posterior densities.

12.4.6.5. Understanding 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

12.4. Modern Bayesian Statistics

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 idea apply in the Bayesian approach.

12.4.6.6. 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

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 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 12.4.8.

12.4.7. Conclusion

This chapter has attempted to make accessible the foundations of statistical inference and the modern tools and approaches

12. Statistical Inference and Information Theory

available. Underlying this approach to thinking about statistical problems are informational theoretic and mathematical concepts that can be challenging to learn when traditional finance and actuarial curricula is not centered on the necessary computational foundations that are associated with modern statistical analysis. Further, the approach of treating estimation not as an exercise in determining a “best estimate” but instead as a range of outcomes will enhance financial analysis and quantification of risks.

12.4.8. 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/>.

Part V.

Computational Thinking in an Actuarial and Financial Context

13. Modeling

13.1. In This Chapter

We discuss how to approach a problem and identify the key attributes to include in the model, what are the inherent trade-offs with different approaches, and how to work with data that feeds your model.

13.2. Parsimony

14. Automatic Differentiation

14.1. In This Chapter

Harnessing the chain rule to compute derivatives not just of simple functions, but of complex programs.

14.2. 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 15). You can test properties and implications (monotonicity, maxima/minima).

14.3. 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's also flavors of finite differentiation to approximate derivatives to be aware of:

14. Automatic Differentiation

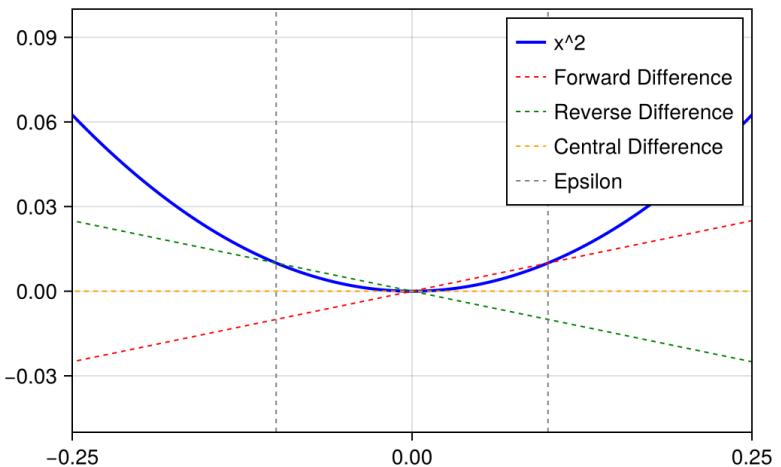
- forward difference is as defined in the above equation, where ϵ is added to x_0
- reverse difference is as defined in the above equation, where ϵ is subtracted from x_0
- central difference is where we evaluate at $x_0 \pm \epsilon$ and then divide by 2ϵ

The benefit of the central difference is that it limits issues around minima and maxima where the trough or peak respectively would seem much steeper if using forward or reverse. Here's a picture of this:

```

[Warning: Found 'resolution' in the theme when creating a 'Scene'.
 @ Makie ~/.julia/packages/Makie/iRM0c/src/scenes.jl:220
[Warning: Found 'resolution' in the theme when creating a 'Scene'.
 @ Makie ~/.julia/packages/Makie/iRM0c/src/scenes.jl:220

```



One benefit of the central difference method is that it is often more accurate than forward or reverse. However it comes at the cost of needing to evaluate the function an additional time in many circumstances. Take, for example, the process of optimizing a function to find a maxima or minima. The process usually involves 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

14.3. Finite Differentiation

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: for x , $x + \epsilon$, and $x - \epsilon$. With forward or reverse finite differences, you can reuse the prior function evaluation of the prior guess x . As one of the components in the estimation of the derivative, thereby saving an evaluation of the function for each iteration.

There are additional challenges with the finite differences 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 grow smaller generally, the accuracy of all three finite different methods increases. However, that's not always the case due to both the complexity of the function that you may be trying to differentiate or due to numerical inaccuracies of floating point math.

To demonstrate, here is a more complex example using an arbitrary function

for this example we'll show the results of the three methods calculated at different values of ϵ :

```
using DataFrames
```

```
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)                                ①

fig = Figure()
ax = Axis(fig[1, 1], xscale=log10, yscale=log10, xlabel="ε", ylabel="absolute error")
scatter!(ax, ε, abs.(estimate .- actual))
fig
```

14. Automatic Differentiation

- ① The derivative of $f(x) = \exp(x)$ is itself. That is $f'(x) = f(x)$ in this special case.

```
Warning: Found 'resolution' in the theme when creating a 'Scene'. The
@ Makie ~/.julia/packages/Makie/iRM0c/src/scenes.jl:220
```

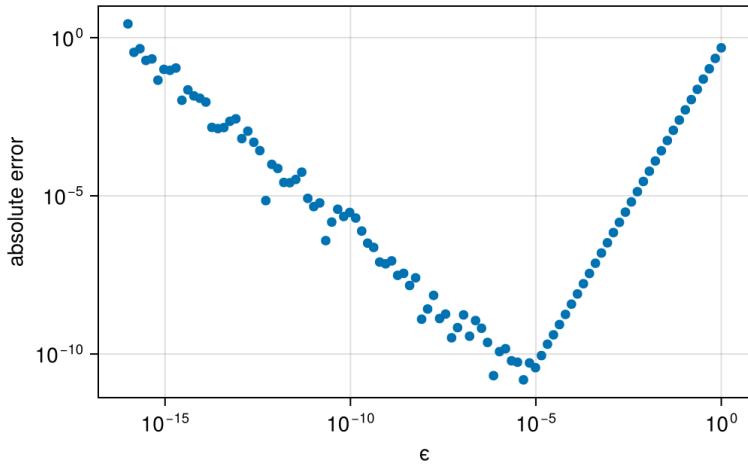


Figure 14.1.: A log-log plot showing the absolute error of the finite differences. Further to the left, roundoff error dominates while further to the right, truncation error dominates.

Note

The `@.` in the code example above is a macro that applies broadcasting each function 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: **truncation error** is inherent in that we are using a given value for ϵ and not determining the limiting analytic value as $\epsilon \rightarrow 0$. The

14.4. Automatic Differentiation

other component is **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 — plus the number of guesses required to find the right answer can depend on the ability to accurately determine the derivative at a point!

Admittedly, 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.

14.4. Automatic Differentiation

Automatic differentiation (“autodiff” or “AD” for short) is essentially the practice of defining algorithmically what the derivatives of functions 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 building up from a bunch of elementary operations, we can get a very long way in differentiating complex functions.

14.4.1. Dual Numbers

To understand where we are going, let's remind ourselves about complex numbers. Complex numbers are of the form which has an 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 is otherwise unsolvable with just the real numbers³⁹. 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 defintion. For our purposes here, one can think of b as the derivative of the function evaluated at the same point as a . An intial example should make this clearer. First let's define a DualNumber:

```
struct DualNumber{T,U} ①
    a::T
    b::U
    function DualNumber(a::T, b::U=zero(a)) where {T,U} ②
        return new{T,U}(a, b)
    end
end
```

14.4. Automatic Differentiation

- ① We define this type parametrically to handle all sorts of `<:Real` types and allow `a` and `b` to vary 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 `zero(a)` is a generic way to create a value equal to zero with the same type of the argument `a`. `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 compare 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)
```

That's not super interesting though - the derivative of `f1` 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:

$$(u \times v)' = u' \times v + u \times v'$$

14. Automatic Differentiation

```
Base.*(d::DualNumber, e::DualNumber) = DualNumber(d.a * g.a, d.b * g.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 $f2$ evaluated at 10.

i 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))
```

14.5. Performance of Automatic Differentiation

```
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:

```
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:

```
let
    f(x) = 5 * exp(sin(x)) + 3x
    derivative(f, 0)
end
```

```
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 the computer to an exact level, limited only by the same machine precision that applies to our primary function of interest as well.

14.5. 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?

14. Automatic Differentiation

```
using BenchmarkTools  
@btime f2(rand())
```

```
2.666 ns (0 allocations: 0 bytes)
```

```
7.2052766776729555
```

```
@btime f2(DualNumber(rand(), 1))
```

```
2.708 ns (0 allocations: 0 bytes)
```

```
DualNumber{Float64, Int64}(7.66642140510705, 3)
```

In performing this computation, the compiler has been able to optimize it such that we effectively are able to compute the function and its derivative at effetcitly the same speed as just the evaluating the function itself! As the function gets more complex, the overhead does increase but is still a *much* preferred option versus finite differentiation. This advantage becomes more pronounced as we contemplate derivatives with respect to many variables at once or for higher-order derivatives.

Note

In fact, it's largely due to the advances in applications of automatic differentiation that has 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 autodiffertiation in specialized C++ libraries underpin the libraries like PyTorch, Tensorfow, and Keras. These libraries specialize in allowing for autodiff on a limited subset of operations. Julia's available automatic differentiation is more general and can be applied to many more scenarios.

14.6. 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 trancendental (`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

- '`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) ①
    d1 = d1(S, K, τ, r, σ, q)
```

14. Automatic Differentiation

```
d2 = d2(S, K, τ, r, σ, q)
return (N(d1) * S * exp(τ * (r - q)) - N(d2) * K) * exp(-r * τ
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.

```
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)
```

```
0.02493376819403728
```

Tip

Some terminology in differentiation:

- **Derivative** is generally the scalar rate of change in output relative to a scalar input and can be used in the context of partial derivatives for a multi-variate function (e.g. $\frac{d}{dx} f(x, y, z)$).
- **Gradient** is the first derivative with respect to all dimensions of a function that outputs a scalar. For a function $f(x, y, z)$ the gradient would be a vector of partial derivatives such that you would get $[\frac{d}{dx}, \frac{d}{dy}, \frac{d}{dz}]$.
- **Jacobian** is the first derivative with respect to all dimensions of a function that outputs a vector.
- **Hessian** is the second derivative with respect to all dimensions of a function that outputs a scalar.

With the above code, now we can get the partial derivatives with respect to each parameter. The first, third, fourth, fifth, and

14.6. Automatic Differentiation in Practice

sixth correspond to the common “greeks” *delta*, *theta*, *rho*, *vega*, and *epsilon* respectively. The second term is the parital derivative with respect to the strike price:

```
ForwardDiff.gradient(eurocall, params)
```

6-element Vector{Float64}:

```
0.5399635456230838  
-0.5150297774290467  
0.16420676980838977  
0.042331214583209334  
0.11379886104405816  
-0.04438056539367815
```

We can also get the second order greeks with a simple call. In addtion to many uncommon second order parital derivatives. *Gamma* is in the [1,1] position for example:

```
ForwardDiff.hessian(eurocall, params)
```

6×6 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
```

14.6.1. Performance

Earlier we examined the impact on performance for the derivatives using the *DualNumber* developed in this chapter on a very basic function. What about if we take a more realistic example like *eurocall*? We can observe approximately a 9x slowdown when computing all of the first order derivatives which isn’t bad considering we are computing 6x of the outputs!

```
@btime eurocall($params)
```

```
37.298 ns (0 allocations: 0 bytes)
```

14. Automatic Differentiation

```
0.02493376819403728
```

```
let
    g = similar(params)                                ①
    @btime ForwardDiff.gradient!($g, eurocall, $params)
end
```

- ① To avoid benchmarking allocating a new array we are able to pre-allocate the memory to store the result and then call gradient! to fill in g for each result.

```
332.386 ns (2 allocations: 704 bytes)
```

```
6-element Vector{Float64}:
 0.5399635456230838
 -0.5150297774290467
 0.16420676980838977
 0.042331214583209334
 0.11379886104405816
 -0.04438056539367815
```

14.7. Forward Mode and Reverse Mode

The approach of autodiff outlined about is called **forward mode** auto-differentiation where the derivative is brought forward through the computation and accumulated through each step. The alternative to this is to first evaluate the function and then work backwards by accumulating the partial derivatives in what's called **reverse mode** automatic differentiation.

Reverse mode requires more book-keeping because unlike the forward mode the derivative needs to be carried backwards, unlike the DualNumber approach of forward mode.

14.8. Practical tips for Automatic Differentiation

Here are a few practical tips to keep in mind.

14.8.1. Choosing between Reverse Mode and Forward Mode

Forward mode is more efficient when the number of outputs is much larger than the number inputs. When the number of inputs is much larger than the number of outputs, then reverse mode will generally be more efficient. Examples of the number of inputs being larger than the outputs might be in a statistical analysis where many features are used to predict a limited number of outcome variables or a complex model with a lot of parameters.

14.8.2. 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 setup. As of 2024, Enzyme.jl has the best support for functions with mutation inside of them.

14.8.3. 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 infrastrucutre in Julia as well as provide a good set of documentation on how to extend the rules for your custom function.

14.8.4. Available Libraries

- **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.

14. Automatic Differentiation

- **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.

In the authors experience, they would probably recommend ForwardDiff.jl first and then Enzyme.jl if reaching for more advanced functionality or looking for reverse mode.

14.9. References

- https://book.sciml.ai/notes/08-Forward-Mode_Automatic_Differentiation.html
- <https://blog.esciencecenter.nl/automatic-differentiation-from-scratch-23d50c699555>

15. Optimization

15.1. In This Chapter

Optimization as root finding or minimization/maximization of defined objectives. Differentiable programming and the benefits to optimization problems. Model fitting as an optimization problem.

15.2. Setup

```
using Flux
using LsqFit
using CairoMakie

[ Info: Precompiling Flux [587475ba-b771-5e3f-ad9e-33799f191a9c]
[ Info: Precompiling IntervalArithmeticDiffRulesExt [86439bda-9ede-5c22-99a0-f82df582fb22]
Precompiling IntervalArithmeticForwardDiffExt
  ✓ IntervalArithmetic → IntervalArithmeticForwardDiffExt
  1 dependency successfully precompiled in 1 seconds. 25 already precompiled.
[ Info: Precompiling IntervalArithmeticForwardDiffExt [ba47a815-ec9a-57c1-b718-e4e972ac9261]
  Warning: Module IntervalArithmeticDiffRulesExt with build ID fabfcfd-30a8-0c86-0000-4b784097c38e
  This may mean IntervalArithmeticDiffRulesExt [86439bda-9ede-5c22-99a0-f82df582fb22] does not support
  @ Base loading.jl:1948
Error: Error during loading of extension IntervalArithmeticDiffRulesExt of IntervalArithmetic, use
  exception =
    1-element ExceptionStack:
  Declaring __precompile__(false) is not allowed in files that are being precompiled.
  Stacktrace:
    [1] _require(pkg::Base.PkgId, env)::Nothing
      @ Base ./loading.jl:1952
    [2] __require_prelocked(uuidkey::Base.PkgId, env)::Nothing
```

15. Optimization

```
    @ Base ./loading.jl:1812
[3] #invoke_in_world#3
    @ ./essentials.jl:926 [inlined]
[4] invoke_in_world
    @ ./essentials.jl:923 [inlined]
[5] _require_prelocked
    @ ./loading.jl:1803 [inlined]
[6] _require_prelocked
    @ ./loading.jl:1802 [inlined]
[7] run_extension_callbacks(extid::Base.ExtensionId)
    @ Base ./loading.jl:1295
[8] run_extension_callbacks(pkgid::Base.PkgId)
    @ Base ./loading.jl:1330
[9] run_package_callbacks(modkey ::Base.PkgId)
    @ Base ./loading.jl:1164
[10] _tryrequire_from_serialized(modkey ::Base.PkgId, path::String)
    @ Base ./loading.jl:1487
[11] _require_search_from_serialized(pkg ::Base.PkgId, sourcepath::String)
    @ Base ./loading.jl:1574
[12] _require(pkg ::Base.PkgId, env ::String)
    @ Base ./loading.jl:1938
[13] __require_prelocked(uuidkey ::Base.PkgId, env ::String)
    @ Base ./loading.jl:1812
[14] #invoke_in_world#3
    @ ./essentials.jl:926 [inlined]
[15] invoke_in_world
    @ ./essentials.jl:923 [inlined]
[16] _require_prelocked(uuidkey ::Base.PkgId, env ::String)
    @ Base ./loading.jl:1803
[17] macro expansion
    @ ./loading.jl:1790 [inlined]
[18] macro expansion
    @ ./lock.jl:267 [inlined]
[19] __require(into::Module, mod::Symbol)
    @ Base ./loading.jl:1753
[20] #invoke_in_world#3
    @ ./essentials.jl:926 [inlined]
[21] invoke_in_world
    @ ./essentials.jl:923 [inlined]
[22] require(into::Module, mod::Symbol)
    @ Base ./loading.jl:1746
```

15.3. Differentiable programming

```
[23] include
    @ ./Base.jl:495 [inlined]
[24] include_package_for_output(pkg::Base.PkgId, input::String, depot_path::Vector{String}, dl
    @ Base ./loading.jl:2222
[25] top-level scope
    @ stdin:3
[26] eval
    @ ./boot.jl:385 [inlined]
[27] include_string(mapexpr::typeof(identity), mod::Module, code::String, filename::String)
    @ Base ./loading.jl:2076
[28] include_string
    @ ./loading.jl:2086 [inlined]
[29] exec_options(opts::Base.JLOptions)
    @ Base ./client.jl:316
[30] _start()
    @ Base ./client.jl:552
@ Base loading.jl:1301
Precompiling ZygoteColorsExt
  ✓ Zygote → ZygoteColorsExt
  1 dependency successfully precompiled in 12 seconds. 74 already precompiled.
[ Info: Precompiling ZygoteColorsExt [e68c091a-8ea5-5ca7-be4f-380657d4ad79]
[ Warning: Module Zygote with build ID fafbfccfd-5ff8-72a2-0000-4b74a07df030 is missing from the cache
  This may mean Zygote [e88e6eb3-aa80-5325-afca-941959d7151f] does not support precompilation but is
  @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing ZygoteColorsExt [e68c091a-8ea5-5ca7-be4f-380657d4ad79]
[ Info: Precompiling LsqFit [2fda8390-95c7-5789-9bda-21331edee243]
[ Info: Precompiling FiniteDiffStaticArraysExt [75e56524-3a34-51de-85ea-03aa6eac4b64]
```

15.3. Differentiable programming

Differentiable programming is an approach to programming where functions are defined using differentiable operations, allowing automatic differentiation to be applied to them. Automatic differentiation is a technique used to efficiently compute derivatives of functions, and it is crucial in many machine learning algorithms, optimization techniques, and scientific computing applications.

Elements in differentiable programming

15. Optimization

- Differentiable Functions: Functions are defined using operations that are differentiable. These operations include basic arithmetic operations (addition, subtraction, multiplication, division), as well as more complex operations like exponentials, logarithms, trigonometric functions, etc.
- Automatic Differentiation (AD): Automatic differentiation is used to compute derivatives of functions with respect to their inputs or parameters. AD exploits the fact that every computer program, no matter how complex, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division), and elementary functions (exponentials, logarithms, trigonometric functions). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program.
- Optimization and Machine Learning: Differentiable programming is particularly useful in optimization problems, where gradients or higher-order derivatives are required to find the minimum or maximum of a function. It's also widely used in machine learning, where optimization algorithms like gradient descent are used to train models by adjusting their parameters to minimize a loss function.

```
using Flux
```

```
# Define a differentiable function
f(x) = 3x^2 + 2x + 1
# Define an input value
x = 2.0

@show "Value of f(x) at x=$x: ", f(x)
@show "Gradient of f(x) at x=$x: ", gradient(x → f(x), x)

("Value of f(x) at x=$(x): ", f(x)) = ("Value of f(x) at x=2.0: ", 17.0)
("Gradient of f(x) at x=$(x): ", gradient((x→begin
```

15.4. Model fitting

```
#= In[4]:9 =#
f(x)
end), x)) = ("Gradient of f(x) at x=2.0: ", (14.0,))

("Gradient of f(x) at x=2.0: ", (14.0,))
```

15.4. Model fitting

15.4.1. 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 of the function to approximate the root with increasing accuracy in each iteration.

```
using Flux

# Define a differentiable function
f(x) = 3x^2 + 2x + 1
# Define an initial value
x = 1.0
# tolerance of difference in value
tol = 1e-6
# maximum number of iteration of the algorithm
max_iter = 100
iter = 0
while abs(f(x)) > tol && iter < max_iter
    x -= f(x) / gradient(x → f(x), x)[1]
    iter += 1
end
if iter == max_iter
    @show "Warning: Maximum number of iterations reached."
```

15. Optimization

```
else
    @show "Root found after", iter, " iterations."
end
@show "Approximate root: ", x

"Warning: Maximum number of iterations reached." = "Warning: Maximum
("Approximate root: ", x) = ("Approximate root: ", -1.391591884376212

("Approximate root: ", -1.391591884376212)
```

15.4.2. Best fitting curve

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.

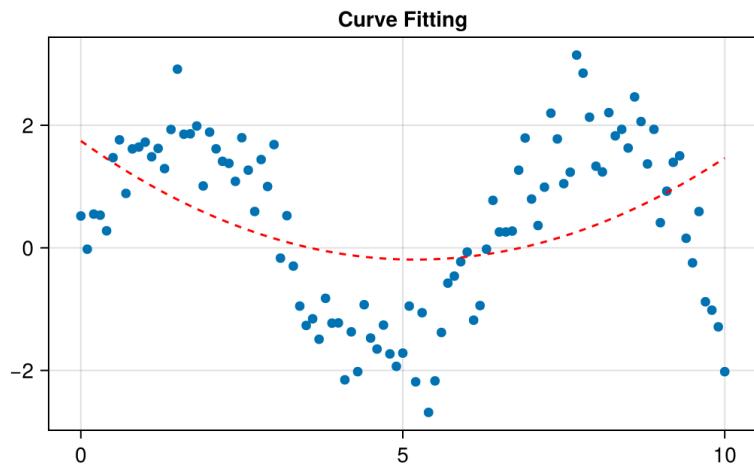
The process of finding the best fitting curve typically involves:

- Choosing a model: Based on the nature of the data and the underlying relationship between the variables, a suitable model or family of models are 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 you find a satisfactory fit to the data.

15.4. Model fitting

```
x_data = 0:0.1:10
y_data = 2 .* sin.(x_data) .+ 0.5 .* randn(length(x_data))
# Define the model function
model(x, p) = p[1] * x.^2 + p[2] * x .+ p[3]
# Initial parameter guess
p₀ = [1.0, 1.0, 1.0]
# Fit the model to the data
fit_result = curve_fit(model, x_data, y_data, p₀)
# Extract the fitted parameters
params = coef(fit_result)
# Evaluate the model with the fitted parameters
y_fit = model(x_data, params)
# Plot the data and the fitted curve
fig = Figure()
Axis(fig[1, 1], title = "Curve Fitting")
scatter!(x_data, y_data, label="Data")
lines!(x_data, y_fit, label="Fitted Curve", linestyle=:dash, color=:red)
fig
```

```
└ Warning: Found 'resolution' in the theme when creating a 'Scene'. The 'resolution' keyword for 'Scene'  
└ @ Makie ~/.julia/packages/Makie/iRM0c/src/scenes.jl:220
```



16. Sensitivity Analysis

[Drafting note: based on some examples. Needs to be revised with more exposition.]

16.1. In This Chapter

Different approaches to understanding the sensitivity of a model to changes in its inputs: derivatives, finite differences, global sensitivity analysis approaches, and statistical approaches.

16.2. Setup

```
using CSV, DataFrames
using MortalityTables, Dates
using GlobalSensitivity
using QuasiMonteCarlo
using CairoMakie

@enum Sex Female = 1 Male = 2
@enum Risk Standard = 1 Preferred = 2

mutable struct Policy
    id::Int
    sex::Sex
    benefit_base::Float64
    COLA::Float64
    mode::Int
    issue_date::Date
    issue_age::Int
```

16. Sensitivity Analysis

```
risk::Risk  
end
```

16.3. The Data

```
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"  
    )  
  
mort = Dict(  
    Male => MortalityTables.table(988).ultimate,  
    Female => MortalityTables.table(992).ultimate,  
)  
  
Dict{Sex, OffsetArrays.OffsetVector{Float64, Vector{Float64}}} with  
    Male => [0.022571, 0.022571, 0.022571, 0.022571, 0.022571, 0.022571]  
    Female => [0.00745, 0.00745, 0.00745, 0.00745, 0.00745, 0.00745]  
  
policies = let  
  
    # read CSV directly into a dataframe  
    # df = CSV.read("sample_inforce.csv", DataFrame) # use local storage  
    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.COLA,  
            row.mode,  
            row.issue_date,  
            row.issue_age,  
            row.risk == "Std" ? Standard : Preferred,  
        )  
    end
```

16.3. The Data

end

1-element Vector{Policy}:

```
Policy(1, Male, 100000.0, 0.03, 12, Date("1999-12-05"), 30, Standard)
```

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 at the end of the y^{th} policy year is defined by

$$res(y) = \sum_{t=age+y}^{120} (sur(t-age-y)*mort_t*B_y*\sqrt{(1+r)}) - (P_y*sur(t-age-y))$$

where

- $mort_t$ is the mortality at age t
- p_y is the survival probability adjusted with COLA, with values of $p_{y-1} = 1$ and $p_x = p_{x-1} * (1 - mort(\text{age} + y)) / (1 + \text{COLA})$ for $x \geq y$, and 0 for $x < y - 1$ or $\text{age} + x \geq 120$, 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
- r is the level interest rate throughout the policy

```
function sur(y::Int, pol::Policy)
    if y == 0
        1
    elseif y < 0 || 120 - y <= pol.issue_age
        0
    else
        sur(y - 1, pol) * (1 - mort[pol.sex][pol.issue_age+y]) / (1 + pol.COLA)
    end
end

function res(y::Int, pol::Policy, P::Float64)
    s = 0.0
```

16. Sensitivity Analysis

```
if y >= 1 && y <= 120 - pol.issue_age
    for t in (pol.issue_age+y):120
        prem = 0.0
        if y <= 9
            prem = P
        end
        s += sur(t - pol.issue_age - y, pol) * mort[pol.sex][t]
    end
end
s
```

res (generic function with 1 method)

16.4. Common Sensitivity Analysis Methodologies

16.4.1. Finite Differences

Define a customized finite difference function with respect to the COLA, rippled by a small difference.

```
function res_wrt_r_fd(y::Int, pol::Policy, P::Float64, r::Float64,
    p_+, p_- = deepcopy(pol), deepcopy(pol)
    p_+.COLA = r + h, p_-.COLA = r - h
    (res(y, p_+, P) - res(y, p_-, P)) / (2h)
end
```

res_wrt_r_fd (generic function with 2 methods)

16.4.2. Scenario Analyses

Scenarios can be generated following scenario generation methodologies to evaluate impacts. Refer to scenario generation chapter.

16.4. Common Sensitivity Analysis Methodologies

16.4.3. Regression Analyses

```
function r1_wrt_r(r)
    p = deepcopy(policies[1])
    p.COLA = r[2]
    res(Int(floor(r[1])), p, r[3])
end

gsa(r1_wrt_r, RegressionGSA(), [[1, 1.01], [0.025, 0.035], [10000.0, 10000.1]], samples=1000)

GlobalSensitivity.RegressionGSAResult{Matrix{Float64}, Nothing}([-0.002241307731791374 0.999704292
```

16.4.4. Sobol Indices

Sobol is a variance-based method, and it 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 affect and sensitivity from the interaction between the parameters.

The Sobol Indices are “order”ed, the first order indices given by the contribution to the output variance of the main effect of . Therefore, it measures the effect of varying 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 and so on can be formed by dividing other terms in the variance decomposition by $\text{Var}(Y)$.

```
L, U = QuasiMonteCarlo.generate_design_matrices(1000, [1, 0.025, 10000.0], [1, 0.035, 10000.1], S
gsa(r1_wrt_r, Sobol(), L, U)
```

```
Warning: The 'generate_design_matrices(n, d, sampler, R = NoRand(), num_mats)' method does not prod
      Prefer using randomization methods such as 'R = Shift()', 'R = MatousekScrambling()', etc., see [
@ QuasiMonteCarlo ~/.julia/packages/QuasiMonteCarlo/KvLfb/src/RandomizedQuasiMonteCarlo/iterator
```

```
GlobalSensitivity.SobolResult{Vector{Float64}, Nothing, Nothing, Nothing}([-0.0, 1.089514785129876
```

16. Sensitivity Analysis

16.4.5. 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:

$$EE_i = \frac{f(x_1, x_2, \dots x_i + \Delta, \dots x_k) - y}{\Delta}$$

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.

```
m = gsa(r1_wrt_r, Morris(), [[1, 1.01], [0.025, 0.035], [10000.0, :]
```

```
GlobalSensitivity.MorrisResult{Matrix{Float64}, Vector{Any}}{[0.0 1]
```

16.4.6. Fourier Amplitude Sensitivity Tests

```
gsa(r1_wrt_r, eFAST(), [[1, 1.01], [0.025, 0.035], [10000.0, 10000.0,
```

```
GlobalSensitivity.eFASTResult{Matrix{Float64}}{[4.0965144421127023]
```

16.5. Benchmarking

17. Stochastic Modeling

The Monte Carlo Method: (i) A last resort when doing numerical integration, and (ii) a way of wastefully using computer time. - Malvin H. Kalos⁴⁰ (c. 1960)

⁴⁰ Kalos was a pioneer in Monte Carlo techniques, quoted via https://doi.org/10.1007/978-3-540-74686-7_3

18. Visualizations

18.1. In This Chapter

The evolved brain and pattern recognition, recommended principles for looking at data, and avoiding common mistakes. Exploratory visualization versus visualizations intended for an audience.

19. Matrices and Their Uses

19.1. In This Chapter

Matrices and their myriad uses: reframing problems through the eyes of linear algebra, an intuitive refreshing on applicable maths, and recurring patterns of matrix operations in financial modeling.

19.2. Setup

```
using LinearAlgebra
using Recommendation
using SparseArrays
using MLDataUtils
using Statistics
using MultivariateStats
```

WARNING: using Recommendation.isdefined in module Main conflicts with an existing identifier.
[Info: Precompiling MLDataUtils [cc2ba9b6-d476-5e6d-8eaf-a92d5412d41d]
[Info: Precompiling MultivariateStats [6f286f6a-111f-5878-ab1e-185364afe411]

19.3. Matrix manipulation

19.3.1. Multiplication

```
# Define two matrices
A = [1 2 3;
      4 5 6;
      7 8 9]
B = [9 8 7;
```

19. Matrices and Their Uses

```
6 5 4;  
3 2 1]  
# Perform matrix multiplication  
C = A * B  
# Display the result  
println("Result of matrix multiplication:")  
println(C)
```

```
Result of matrix multiplication:  
[30 24 18; 84 69 54; 138 114 90]
```

19.3.2. Inversion

```
# Define a matrix  
A = [1 2; 3 4]  
# Compute the inverse of the matrix  
A_inv = inv(A)  
# Display the result  
println("Inverse of matrix A:")  
println(A_inv)
```

```
Inverse of matrix A:  
[-1.999999999999996 0.999999999999998; 1.499999999999998 -0.499999999999998]
```

19.4. Matrix decomposition

19.4.1. Eigenvalues

Eigenvalue decomposition, also known as eigendecomposition, is a matrix factorization that decomposes a matrix into its eigenvectors and eigenvalues.

```
# Create a square matrix  
A = [1 2 3;  
     4 5 6;  
     7 8 9]  
# Perform eigenvalue decomposition  
eigen_A = eigen(A)
```

19.4. Matrix decomposition

```
# Extract eigenvalues and eigenvectors
λ = eigen_A.values
V = eigen_A.vectors

# Display the results
println("Original Matrix:")
println(A)
println("\nEigenvalues:")
println(λ)
println("\nEigenvectors:")
println(V)
```

Original Matrix:

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

Eigenvalues:

```
[-1.1168439698070434, -8.582743335036247e-16, 16.11684396980703]
```

Eigenvectors:

```
[-0.7858302387420671 0.4082482904638635 -0.2319706872462857; -0.0867513392566285 -0.81649658092772
```

19.4.2. Singular values

Singular value decomposition breaks a matrix into three matrices U , Σ , and V , representing the left singular vectors, the singular values (diagonal matrix), and the right singular vectors, respectively.

```
# Create a random matrix
A = rand(4, 3)
# Perform Singular Value Decomposition (SVD)
U, Σ, V = svd(A)
# U: Left singular vectors
# Σ: Singular values (diagonal matrix)
# V: Right singular vectors (transpose)
# Reconstruct original matrix
A_reconstructed = U * Diagonal(Σ) * V'

# Display the results
println("Original Matrix:")
```

19. Matrices and Their Uses

```
println(A)
println("\nLeft Singular Vectors:")
println(U)
println("\nSingular Values:")
println(Σ)
println("\nRight Singular Vectors:")
println(V)
println("\nReconstructed Matrix:")
println(A_reconstructed)
```

Original Matrix:

```
[0.16720577565901018 0.9499198741405194 0.7782989327254124; 0.23490
```

Left Singular Vectors:

```
[0.6323059888162866 -0.11769697950211815 0.6002095350316541; 0.5680
```

Singular Values:

```
[1.9491068971910979, 0.6705216530687181, 0.17424819691180302]
```

Right Singular Vectors:

```
[0.2190399606832278 0.008929244420445112 -0.9756750300268887; 0.713
```

Reconstructed Matrix:

```
[0.16720577565901018 0.9499198741405191 0.7782989327254124; 0.23490
```

19.4.3. Matrix factorization and fatorization 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

19.4. Matrix decomposition

models extend traditional linear models by incorporating interactions between features, allowing them to capture complex relationships within the data.

```
# Generate synthetic user-item interaction data
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)
# Create a sparse user-item matrix
user_item_matrix = sparse(user_ids, item_ids, ratings)
# Split data into training and testing sets
train_data, test_data = splitobs(user_item_matrix, 0.8)
# Set parameters for matrix factorization
num_factors = 10
num_iterations = 10
# Train matrix factorization model
data = DataAccessor(user_item_matrix)
recommender = MF(data) # FactorizationMachines(data) alternatively
fit!(recommender)
# Predict ratings for the test set
rec = Dict()
for user in 1:num_users
    rec[user] = recommend(recommender, user, num_items, collect(1:num_items))
end
# Evaluate model performance
predictions = []
for (i, j, v) in zip(findnz(test_data.data)[1], findnz(test_data.data)[2], findnz(test_data.data)[3])
    for p in rec[i]
        if p[1] == j
            push!(predictions, p[2])
            break
        end
    end
end
rmse = measure(RMSE(), predictions, nonzeros(test_data.data))
println("Root Mean Squared Error (RMSE): ", rmse)
```

Root Mean Squared Error (RMSE): 1.3019156056001477

19.4.4. 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.

```
# Generate some synthetic data
data = randn(100, 5) # 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 = pca_model.prinvars
explained_variance_ratio = pca_model.prinvars / sum(pca_model.prinvars)

# Print results
println("Principal Components:")
println(principal_components)
println("Explained Variance Ratio:")
println(explained_variance_ratio)
```

Principal Components:
[29.742594233566354, 23.94085959875916]
Explained Variance Ratio:
[0.5540365254155246, 0.4459634745844755]

20. Learning from Data

20.1. In this chapter

Using data to inform a model: fitting parameters, forecasting, and fundamental limitations on prediction. Also covered are elements of practical review such as static and dynamic validations, and implied rate analysis.

20.2. Setup

```
using MLJ
using StatsBase
using Flux

[ Info: Precompiling MLJ [add582a8-e3ab-11e8-2d5e-e98b27df1bc7]
Precompiling BangBangStructArraysExt
  ✓ BangBang → BangBangStructArraysExt
  1 dependency successfully precompiled in 1 seconds. 23 already precompiled.
[ Info: Precompiling BangBangStructArraysExt [d139770a-8b79-56c4-91f8-7273c836fd96]
  Warning: Module BangBang with build ID fafbfcfd-d8ea-dd8b-0001-d67a5cc4e069 is missing from the cache.
  This may mean BangBang [198e06fe-97b7-11e9-32a5-e1d131e6ad66] does not support precompilation but is
  @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing BangBangStructArraysExt [d139770a-8b79-56c4-91f8-7273c836fd96]
Precompiling Flux
  ✓ OneHotArrays
  ✓ Flux
  2 dependencies successfully precompiled in 12 seconds. 111 already precompiled.
[ Info: Precompiling Flux [587475ba-b771-5e3f-ad9e-33799f191a9c]
  Warning: Module NNlib with build ID fafbfcfd-40b5-bddf-0001-d67b2ca386c5 is missing from the cache.
  This may mean NNlib [872c559c-99b0-510c-b3b7-b6c96a88d5cd] does not support precompilation but is
  @ Base loading.jl:1948
```

20. Learning from Data

```
[ Info: Skipping precompilation since __precompile__(false). Importing...
[ Info: Precompiling Optimisers [3bd65402-5787-11e9-1adc-39752487f4e...
[ Info: Precompiling Zygote [e88e6eb3-aa80-5325-afca-941959d7151f]
[ Info: Precompiling ZygoteColorsExt [e68c091a-8ea5-5ca7-be4f-380657...
[ Info: Precompiling OneHotArrays [0b1bfda6-eb8a-41d2-88d8-f5af5cad4...
[ Warning: Module NNlib with build ID fabfcfd-40b5-bddf-0001-d67b2ca...
[ This may mean NNlib [872c559c-99b0-510c-b3b7-b6c96a88d5cd] does not...
[ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing...
```

20.3. Applications

20.3.1. Parameter fitting

Refer to the chapter on Optimization for more details.

20.3.2. Forecasting

20.3.3. Static and dynamic validation

Static validation typically involves splitting the dataset into training and testing sets, where the testing set is held out and not used during model training. The model is trained on the training set and then evaluated on the held-out testing set to assess its performance. This approach helps to measure how well the model generalizes to unseen data.

Dynamic validation, on the other hand, involves using a rolling or expanding window to train and test the model iteratively over time. In each iteration, the model is trained on past data and tested on future data, simulating how the model would perform in a real-world scenario where new data becomes available over time. This approach helps to assess the model's ability to adapt to changing patterns and trends in the data.

```
# Generate synthetic time series data
num_samples = 100
data = rand(num_samples)
```

20.3. Applications

```
X = [ones(num_samples) data]
y = 2data .+ 1 .+ 0.1 * randn(num_samples, 1) # dependent variable with noise
# Train the model on the training set
θ = X \ y
# Predictions
y_pred = θ[2] .* data .+ θ[1]
# Compute evaluation metrics
mse = mean((y_pred .- y).^2)
mae = mean(abs.(y_pred .- y))

println("Static validation results:")
println("Mean Squared Error (MSE): ", mse)
println("Mean Absolute Error (MAE): ", mae)

# Dynamic validation to update model over time and evaluate
num_updates = 5
mse_dyn = Float64[]
mae_dyn = Float64[]
for i in 1:num_updates
    data = rand(num_samples)
    X = [ones(num_samples) data]
    y = 2data .+ 1 .+ 0.1 * randn(num_samples, 1) # dependent variable with noise
    # Train the model on the training set
    θ = X \ y
    # Predictions
    y_pred = θ[2] .* data .+ θ[1]
    # Compute evaluation metrics
    mse = mean((y_pred .- y).^2)
    mae = mean(abs.(y_pred .- y))
    push!(mse_dyn, mse)
    push!(mae_dyn, mae)
end

println("Dynamic validation results:")
println("Mean Squared Error (MSE): ", mean(mse_dyn))
println("Mean Absolute Error (MAE): ", mean(mae_dyn))

Static validation results:
Mean Squared Error (MSE): 0.008628877559463276
Mean Absolute Error (MAE): 0.07210160900201301
Dynamic validation results:
```

20. Learning from Data

```
Mean Squared Error (MSE): 0.009511619167681175
Mean Absolute Error (MAE): 0.07786799234508966
```

20.3.4. Implied rate analysis

Implied rates are rates that are derived from the prices of financial instruments, such as bonds or options. For example, in the context of bonds, the implied rate is the interest rate that equates the present value of future cash flows from the bond (coupons and principal) to its current market price.

```
# Define the bond cash flows and prices
cash_flows = [100, 100, 100, 100, 1000] # Coupons and principal
prices = [95, 96, 97, 98, 1050] # Market prices
# Define a function to calculate the present value of cash flows given a rate
function present_value(rate, cash_flows)
    pv = 0
    for (i, cf) in enumerate(cash_flows)
        pv += cf / (1 + rate)^i
    end
    return pv
end
# Define a function to calculate the implied rate using bisection method
function implied_rate(cash_flows, price)
    f(rate) = present_value(rate, cash_flows) - price
    return rootassign(f, 0.0, 1.0)
end
function rootassign(f, l, u)
    # Define an initial value
    x = 1.0
    # tolerance of difference in value
    tol = 1e-6
    # maximum number of iteration of the algorithm
    max_iter = 100
    iter = 0
    while abs(f(x)) > tol && iter < max_iter
        x -= f(x) / gradient(x → f(x), x)[1]
        iter += 1
    end
    if iter < max_iter && l < x < u
```

20.3. Applications

```
        return x
    else
        return -1.0
    end
end
# Calculate implied rates for each bond
implied_rates = [implied_rate(cash_flows, price) for price in prices]
# Print the results
for (i, rate) in enumerate(implied_rates)
    println("Implied rate for bond $i: $rate")
end

Implied rate for bond 1: -1.0
Implied rate for bond 2: -1.0
Implied rate for bond 3: -1.0
Implied rate for bond 4: -1.0
Implied rate for bond 5: -1.0
```


Part VI.

Applications in Practice

21. Stochastic Mortality Projections

[Drafting note: taken from a tutorial on JuliaActuary.org.
Needs to be revised with more exposition.]

21.1. In This Chapter

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.

21.2. Setup

```
using CSV, DataFrames
using MortalityTables, ActuaryUtilities
using Dates
using ThreadsX
using BenchmarkTools
using Random
using CairoMakie
```

Define a datatype. Not strictly necessary, but will make extending the program with more functions easier.

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. This makes the whole data be stored in the stack and is an example of data-oriented design. It's much slower without the type annotations (~0.5 million policies per second, ~50x slower).

21. Stochastic Mortality Projections

```
@enum Sex Female = 1 Male = 2
@enum Risk Standard = 1 Preferred = 2

struct Policy
    id::Int
    sex::Sex
    benefit_base::Float64
    COLA::Float64
    mode::Int
    issue_date::Date
    issue_age::Int
    risk::Risk
end
```

21.3. The Data

```
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"
    )

IOBuffer(data=UInt8[...], readable=true, writable=false, seekable=true)

policies = let

    # read CSV directly into a dataframe
    # df = CSV.read("sample_inforce.csv",DataFrame) # use local storage
    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.COLA,
            row.mode,
```

21.3. The Data

```
    row.issue_date,  
    row.issue_age,  
    row.risk == "Std" ? Standard : Preferred,  
)  
end  
  
end  
  
2-element Vector{Policy}:  
Policy(1, Male, 100000.0, 0.03, 12, Date("1999-12-05"), 30, Standard)  
Policy(2, Female, 200000.0, 0.03, 12, Date("1999-12-05"), 30, Preferred)
```

Define what mortality gets used:

```
mort = Dict(  
    Male => MortalityTables.table(988).ultimate,  
    Female => MortalityTables.table(992).ultimate,  
)  
  
function mortality(pol::Policy, params)  
    return params.mortality[pol.sex]  
end  
  
mortality (generic function with 1 method)
```

This defines the core logic of the policy projection and will write the results to the given out container (here, a named tuple of arrays).

This is using a threaded approach where it could be operating on any of the computer's available threads, thus achieving thread-based parallelism - as opposed to multi-processor (multi-machine) or GPU-based computation, which requires formulating the problem a bit differently (array/matrix based). For the scale of computation here, I think I'd apply this model of parallelism.

21. Stochastic Mortality Projections

```
function pol_project!(out, policy, params)
    # some starting values for the given policy
    dur = duration(policy.issue_date, params.val_date)
    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)

    # grab the current thread's id to write to results container w
    tid = Threads.threadid()

    ω = lastindex(qs)

    # inbounds turns off bounds-checking, which makes hot loops fa
    @inbounds for t in 1:min(params.proj_length, ω - start_age)

        q = qs[start_age+t] # get current mortality

        if (rand() < q)
            return # if dead then just return and don't increment t
        else
            # pay benefit, add a life to the output count, and increment t
            out.benefits[t, tid] += cur_benefit
            out.lives[t, tid] += 1
            cur_benefit *= COLA_factor
        end
    end
end

pol_project! (generic function with 1 method)
```

Parameters for our projection:

```
params = (
    val_date=Date(2021, 12, 31),
    proj_length=100,
    mortality=mort,
)
```

21.4. Running the projection

```
(val_date = Date("2021-12-31"), proj_length = 100, mortality = Dict{Sex, OffsetArrays.OffsetVector{Float64}}(
```

Check the number of threads we're using:

```
Threads.nthreads()
```

4

```
function project(policies, params)
    threads = Threads.nthreads()
    benefits = zeros(params.proj_length, threads)
    lives = zeros(Int, params.proj_length, threads)
    out = (; benefits, lives)
    ThreadsX.foreach(policies) do pol
        pol_project!(out, pol, params)
    end
    map(x → vec(reduce(+, x, dims=2)), out)
end
```

```
project (generic function with 1 method)
```

21.4. Running the projection

Example of a single projection:

```
project(repeat(policies, 100_000), params)
```

```
(benefits = [5.6309680538211426e10, 5.673725709783029e10, 5.7102406126984146e10, 5.740720327528844e10,
```

21.4.1. Stochastic Projection

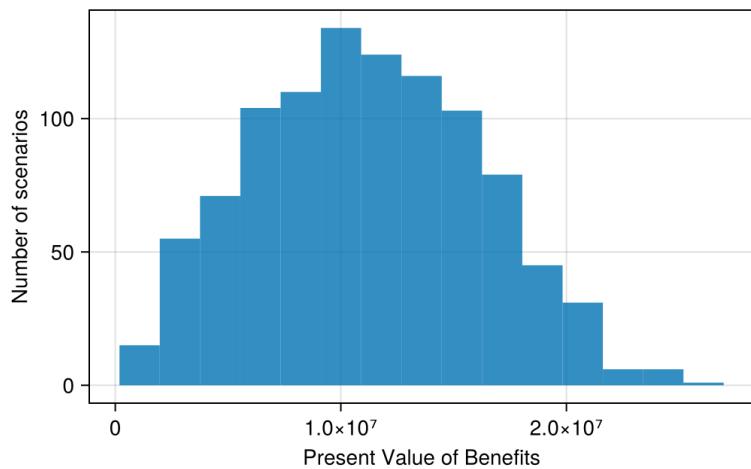
Loop through and calculate the results n times (this is only running the two policies in the sample data "n times").

21. Stochastic Mortality Projections

21.5. Benchmarking

```
hist(v,
    bins=15,
    axis=(  
    xlabel="Present Value of Benefits",
    ylabel="Number of scenarios"  
)  
)  
end
```

```
[Warning: Found 'resolution' in the theme when creating a 'Scene'. The 'resolution' keyword for 'Scene'  
@ Makie ~/.julia/packages/Makie/GtFuI/src/scenes.jl:227
```



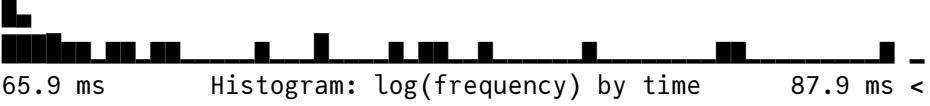
21.5. Benchmarking

Using a 2022 Macbook Air M2 laptop, about 30 million policies able to be stochastically projected per second:

```
policies_to_benchmark = 3_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)
```

21. Stochastic Mortality Projections

```
BenchmarkTools.Trial: 58 samples with 1 evaluation.
Range (min ... max): 65.865 ms ... 101.253 ms | GC (min ... max): 0.00% ... 0.00%
Time  (median):      66.275 ms                | GC (median):      0.00%
Time  (mean ± σ):   69.385 ms ±  6.711 ms | GC (mean ± σ):  0.00% ± 0.00%


Histogram: log(frequency) by time
65.9 ms          87.9 ms <
```

Memory estimate: 29.20 KiB, allocs estimate: 222.

21.6. Further Optimization

In no particular order:

- the RNG could be made faster: <https://bkamins.github.io/julialang/2020/01/21/RNG.html>
- Could make the stochastic set distributed, but at the current speed the overhead of distributed computing is probably more time than it would save. Same thing with GPU projections
- ...

22. Scenario Generation

[Drafting note: based on some examples. Needs to be revised with more exposition.]

22.1. In This Chapter

How to generate synthetic data for your model using sub-models, with applications to economic scenario generation and portfolio composition.

22.2. Setup

```
using CSV, DataFrames
using Random
using StatsBase, Distributions
using CairoMakie
```

22.3. The Data

22.4. Pseudo Random Number Generators

Modern computers utilize Pseudo random number generators (PRNGs) to generate random-like numbers. PRNGs are algorithms used to generate sequences of numbers that appear to be random but are actually determined by an initial value, known as the seed. These generators are called “pseudo-random” because the sequences they produce are deterministic; if you provide the same seed, you’ll get the same sequence of numbers. In addition, they have a finite

22. Scenario Generation

period, which means that after a certain number of generated values, the sequence will repeat. It's important to choose or design PRNGs with a long enough period for practical applications.

22.4.1. Common PRNGs

22.4.1.1. Mersenne Twister

One of the strengths of the Mersenne Twister is its exceptionally long period. The period is $2^{19937} - 1$, which means it can generate $2^{19937} - 1$ pseudo random numbers before repeating. This long period is crucial for applications requiring a large number of independent random numbers. It is also known for its good statistical properties. It passes many standard tests for randomness and provides a relatively uniform distribution of random numbers. Moreover, it is designed to allow multiple independent instances to be used concurrently without interfering with each other. This makes it suitable for parallel computing. Although there are faster generators for specific use cases, the Mersenne Twister is still often favored for its balance between speed and quality.

22.4.1.2. Xorshift

Xorshift is a family of PRNGs known for their simplicity and relatively 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. Xorshift PRNGs use bitwise XOR, left shifts, and right shifts to update the internal state and generate pseudo-random numbers. The basic idea is to repeatedly apply these operations to the state to produce a sequence of numbers. The period of a typical xorshift generator is relatively short compared to some other PRNGs like the Mersenne Twister. However, there are variations of xorshift algorithms that can have longer periods. One of the main advantages of xorshift is its simplicity and

22.4. Pseudo Random Number Generators

speed. The bitwise XOR and bit-shifting operations can be efficiently implemented in hardware, making xorshift generators suitable for applications where fast random number generation is crucial.

22.4.1.3. Xoshiro

Xoshiro is a family of PRNGs known for their high performance and good statistical properties. The name “Xoshiro” is derived from the Japanese word “xoroshiro,” meaning “random.” Xoshiro algorithms, including Xoshiro128 and others, use a combination of bitwise XOR, bit-shifting, and addition operations. They often have more complex update rules than basic Xorshift algorithms. In addition, they typically have longer periods, making them suitable for applications that require more pseudo-random numbers before repetition.

22.4.2. 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.

```
rng = MersenneTwister(1234)
rand(Int, (2, 3))
```

```
2×3 Matrix{Int64}:
-4974381784106313274 2634959075320138178 2010061907088968002
-9188312129532596093 3178300835531959323 7603888936399828126
```

```
rng = MersenneTwister(1234)
rand(Float64, (2, 3))
```

```
2×3 Matrix{Float64}:
0.376287  0.148729  0.208413
0.464757  0.857355  0.416067
```

```
rng = Xoshiro(1234)
rand(Bool, (2, 3))
```

22. Scenario Generation

2x3 Matrix{Bool}:

0	1	1
1	0	0

22.5. 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, including stochastic differential equations (SDEs) and Monte Carlo simulations.

22.5.1. Interest Rate Models

22.5.1.1. 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 addresses the issue of negative interest rates by ensuring that interest rates remain positive. 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.

22.5. Common Economic Scenario Generation Approaches

- $dW(t)$ 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 become a Vasicek model.

```
# Set seed for reproducibility
Random.seed!(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

# Function to simulate CIR process
```

22. Scenario Generation

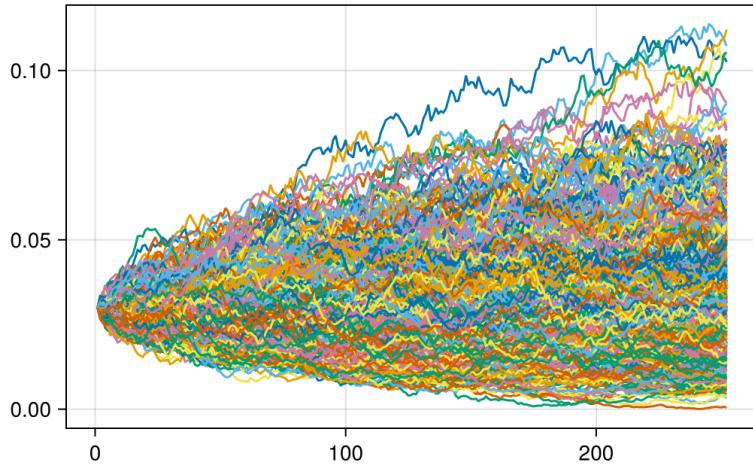
```
function cir_simulation(κ, θ, σ, r₀, Δt, num_steps, num_simulations)
    interest_rate_paths = zeros(num_steps, num_simulations)
    for j in 1:num_simulations
        interest_rate_paths[1, j] = r₀
        for i in 2:num_steps
            dW = randn() * sqrt(Δt)
            # for Vasicek
            # dr = κ * (θ - interest_rate_paths[i-1, j]) * Δt + σ *
            dr = κ * (θ - interest_rate_paths[i-1, j]) * Δt + σ * dW
            interest_rate_paths[i, j] = max(interest_rate_paths[i-1, j], dr)
        end
    end
    return interest_rate_paths
end

# Run CIR simulation
cir_paths = cir_simulation(κ, θ, σ, r₀, Δt, num_steps, num_simulations)

# Plot the simulated interest rate paths
f = Figure()
Axis(f[1, 1])
for i in 1:num_simulations
    lines!(1:num_steps, cir_paths[:, i])
end
f
```

```
[ Warning: Found 'resolution' in the theme when creating a 'Scene'. This
└ @ Makie ~/.julia/packages/Makie/iRM0c/src/scenes.jl:220
```

22.5. Common Economic Scenario Generation Approaches



22.5.1.2. Hull White

The Hull-White model is a one-factor model that extends the Vasicek model by allowing the mean reversion and volatility parameters to be time-dependent. It is commonly used for pricing interest rate derivatives. Brace-Gatarek-Musiela (BGM) Model extends the Hull-White model to incorporate more factors. It is one of the Libor Market Model (LMM) that describes the evolution of forward rates. It allows for the modeling of both the short-rate and the entire yield curve. It 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 .
- θ is the long-term mean or 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.

22. Scenario Generation

```
# Set seed for reproducibility
Random.seed!(1234)

# Hull-White model parameters
α = 0.1      # Mean reversion speed
σ = 0.02     # Volatility
r₀ = 0.03    # Initial short-term interest rate

# Number of time steps and simulations
num_steps = 252
num_simulations = 1_000

# Time increment
Δt = 1/252

# Function to simulate Hull-White process
function hull_white_simulation(α, σ, r₀, Δt, num_steps, num_simulations)
    interest_rate_paths = zeros(num_steps, num_simulations)
    for j in 1:num_simulations
        interest_rate_paths[1, j] = r₀
        for i in 2:num_steps
            dW = randn() * sqrt(Δt)
            dr = α * (σ - interest_rate_paths[i-1, j]) * Δt + σ * dW
            interest_rate_paths[i, j] = interest_rate_paths[i-1, j] + dr
        end
    end
    return interest_rate_paths
end

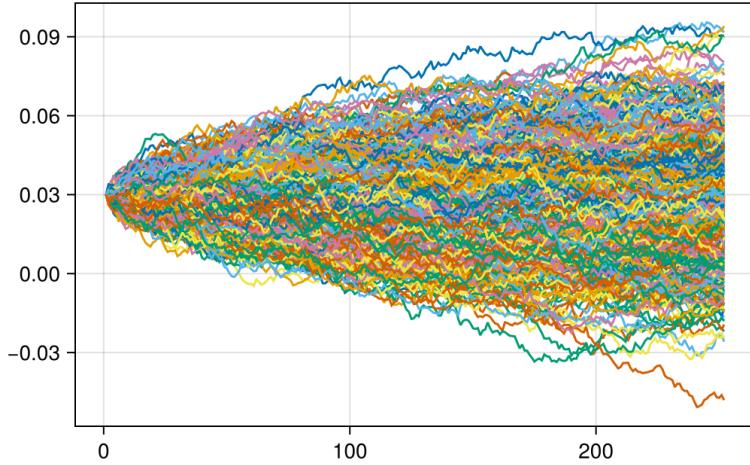
# Run Hull-White simulation
hull_white_paths = hull_white_simulation(α, σ, r₀, Δt, num_steps, num_simulations)

# Plot the simulated interest rate paths
f = Figure()
Axis(f[1, 1])
for i in 1:num_simulations
    lines!(1:num_steps, hull_white_paths[:, i])
end
f

Warning: Found 'resolution' in the theme when creating a 'Scene'. The
```

22.5. Common Economic Scenario Generation Approaches

```
L @ Makie ~/.julia/packages/Makie/iRM0c/src/scenes.jl:220
```



22.5.2. Stock Models

22.5.2.1. Geometric Brownian Motion (GBM)

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.

```
# Set seed for reproducibility
Random.seed!(1234)
```

22. Scenario Generation

```
# GBM parameters
μ = 0.05      # Drift (expected return)
σ = 0.2       # Volatility

# Initial stock price
S₀ = 100

# Number of time steps and simulations
num_steps = 252
num_simulations = 1_000

# Time increment
Δt = 1/252

# Function to simulate GBM
function gbm_simulation(μ, σ, S₀, Δt, num_steps, num_simulations)
    stock_price_paths = zeros(num_steps, num_simulations)
    for j in 1:num_simulations
        stock_price_paths[1, j] = S₀
        for i in 2:num_steps
            dW = randn() * sqrt(Δt)
            dS = μ * S₀ * Δt + σ * S₀ * dW
            stock_price_paths[i, j] = stock_price_paths[i-1, j] + dS
        end
    end
    return stock_price_paths
end

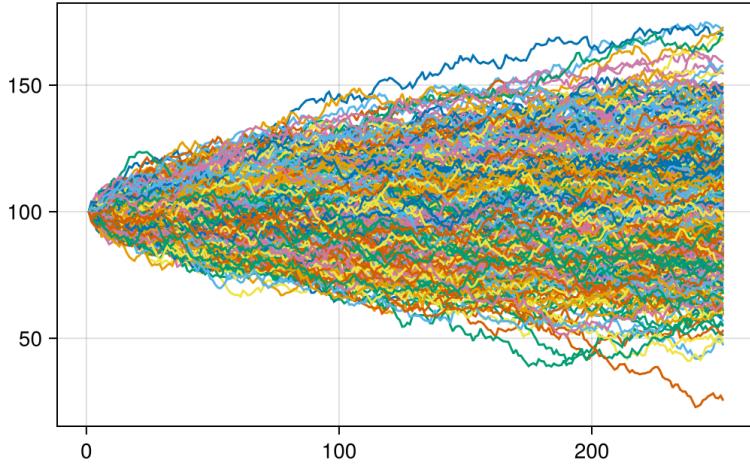
# Run GBM simulation
gbm_paths = gbm_simulation(μ, σ, S₀, Δt, num_steps, num_simulations)

# Plot the simulated stock price paths
f = Figure()
Axis(f[1, 1])
for i in 1:num_simulations
    lines!(1:num_steps, gbm_paths[:, i])
end
f
```

Γ Warning: Found 'resolution' in the theme when creating a 'Scene'. The

22.5. Common Economic Scenario Generation Approaches

```
L @ Makie ~/.julia/packages/Makie/iRM0c/src/scenes.jl:220
```



22.5.2.2. Generalized Autoregressive Conditional Heteroskedasticity (GARCH)

GARCH models capture time-varying volatility. They are often used in conjunction with other models to forecast volatility. It 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

```
# Set seed for reproducibility
Random.seed!(1234)
```

```
# GARCH(1,1) parameters
α₀ = 0.01      # Constant term
```

22. Scenario Generation

```
α₁ = 0.1      # Coefficient for lagged squared returns
β₁ = 0.8      # Coefficient for lagged conditional volatility

# Number of time steps and simulations
num_steps = 252
num_simulations = 1_000

# Time increment
Δt = 1/252

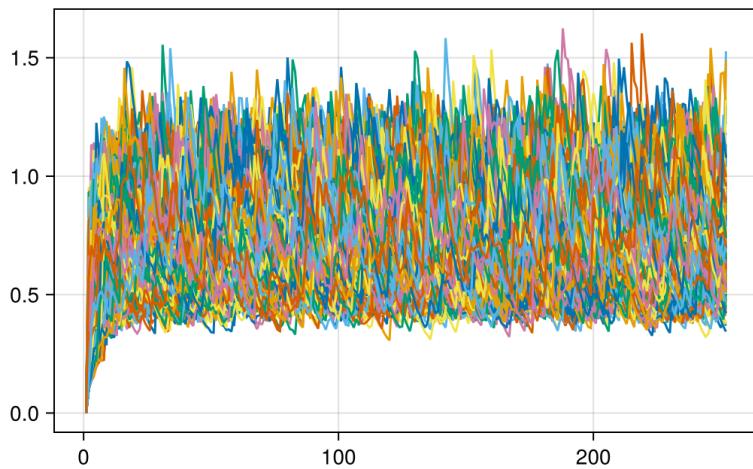
# Function to simulate GARCH(1,1) volatility
function garch_simulation(α₀, α₁, β₁, num_steps, num_simulations)
    volatility_paths = zeros(num_steps, num_simulations)
    for j in 1:num_simulations
        ε = randn(num_steps)
        squared_returns = zeros(num_steps)
        for i in 2:num_steps
            squared_returns[i] = α₀ + α₁ * ε[i-1]^2 + β₁ * squared_returns[i-1]
            volatility_paths[i, j] = sqrt(squared_returns[i])
        end
    end
    return volatility_paths
end

# Run GARCH simulation
garch_paths = garch_simulation(α₀, α₁, β₁, num_steps, num_simulations)

# Plot the simulated volatility paths
f = Figure()
Axis(f[1, 1])
for i in 1:num_simulations
    lines!(1:num_steps, garch_paths[:, i])
end
f
```

```
[ Warning: Found 'resolution' in the theme when creating a 'Scene'. This
  @ Makie ~/.julia/packages/Makie/iRM0c/src/scenes.jl:220
```

22.5. Common Economic Scenario Generation Approaches



22.5.3. Copulas

Simulating data using copulas involves generating multivariate samples with specified marginal distributions and a copula structure.

```
# Set seed for reproducibility
Random.seed!(1234)

# Marginal distributions (e.g., normal)
marginal1 = Normal(0, 1)
marginal2 = Normal(0, 1)

# Clayton copula parameters
theta = 0.5

# Number of data points
num_points = 1000

# Generate independent samples from marginals
u1 = rand(marginal1, num_points)
u2 = rand(marginal2, num_points)

# Clayton copula simulation
function clayton_copula_simulation(u1, u2, theta)
```

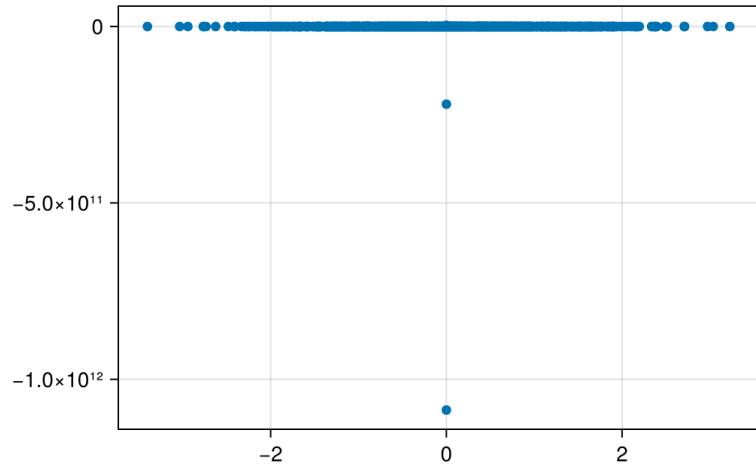
22. Scenario Generation

```
v1 = u1
v2 = u2 .* ((theta .* u1).^(-1/theta - 1))
return v1, v2
end

# Simulate Clayton copula
v1, v2 = clayton_copula_simulation(u1, u2, theta)

# Plot the simulated bivariate data
f = Figure()
Axis(f[1, 1])
scatter!(v1, v2)
f

Warning: Found 'resolution' in the theme when creating a 'Scene'. The
@ Makie ~/.julia/packages/Makie/iRM0c/src/scenes.jl:220
```



22.6. Benchmarking

23. Similarity Analysis

[Drafting note: based on some examples. Needs to be revised with more exposition.]

23.1. In This Chapter

Given a set of interest, understanding the relative similarity (or not) of features of interest is useful in classification and data compression techniques.

23.2. Setup

```
using CSV, DataFrames  
using LinearAlgebra  
using StatsBase, TableTransforms  
using CairoMakie  
using NearestNeighbors
```

23.3. The Data

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

23. Similarity Analysis

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. 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.

```
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"
    )

IOBuffer(data=UInt8[...], readable=true, writable=false, seekable=t
df = CSV.read(sample_csv_data, DataFrame)
df_num = apply(MinMax(), df[:, [:benefit_base, :issue_age]])[1]

| benefit_base | issue_age | |
|---|---|---|
| 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 |

arr_cat = hcat(indicatorformat(df.sex)', indicatorformat(df.education)',
```

23.4. Common Similarity Measures

```
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
```

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.

23.4. Common Similarity Measures

The following measures are commonly used to calculate similarities.

23.4.1. 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 rows in df_num.

```
#d12 = √(Σ((Array(df_num[1, :]) .- Array(df_num[2, :])) .* (Array(df_num[1, :]) .- Array(df_num[2, :])))
d12 = LinearAlgebra.norm(Array(df_num[1, :]) .- Array(df_num[2, :]))
```

```
0.5590169943749475
```

23. Similarity Analysis

23.4.2. Manhattan Distance (L1 Norm)

Manhattan distance, also known as the L1 norm, is defined as

$$d = \sum_{i=1}^n |w_i - v_i|$$

The distance is also usually meaningful when applied to numerical data. The following Julia code shows the Euclidean distance for the first two rows in df_num.

```
#d1 2 = Σ(abs.(Array(df_num[1, :]) .- Array(df_num[2, :])))
d1 2 = LinearAlgebra.norm1(Array(df_num[1, :]) .- Array(df_num[2, :]))
```

0.75

23.4.3. 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 distance would be meaningful when applied to both numerical and categorical data.

The following Julia code shows the cosine similarity for the first two rows in df_num.

```
d1 2 = (Array(df_num[1, :]) · Array(df_num[2, :])) / norm(df_num[1,
```

0.7071067811865475

The following Julia code shows the cosine similarity for the first and the third rows in arr_cat.

```
d1 3 = (arr_cat[1, :] · arr_cat[3, :]) / norm(arr_cat[1, :]) / norm
```

0.3333333333333337

23.4. Common Similarity Measures

Note how similar the syntax of processing for numerical or categorical data is. Multiple dispatch allows Julia to identify most efficient underlying procedure for different types of data. For categorical data, the *dot* operation on binary vectors is essentially count of 1's, while for numerical data it is the *dot* operation for most numerical processing libraries.

23.4.4. Jaccard Similarity

Jaccard similarity is defined as

$$d = \frac{|W \cap V|}{|W \cup V|}$$

The distance 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.

```
d13 = (arr_cat[1, :] .* arr_cat[3, :]) / sum(arr_cat[1, :] .| arr_cat[3, :])
```

0.2

23.4.5. Hamming Distance

Hamming distance is defined as $d = \text{Number of positions at which } w \text{ and } v \text{ differ}$. The distance 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.

```
d13 = sum(arr_cat[1, :] .\ arr_cat[3, :])
```

4

23.5. k-Nearest Neighbor (kNN) Clustering

kNN 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 kNN 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.

```
# Create a kNN model
k = 1
knn_model = KDTree(Array(df_num))

# Query point for prediction
query_point = rand(2)

# Find k nearest neighbors
indices, distances = knn(knn_model, query_point, k)

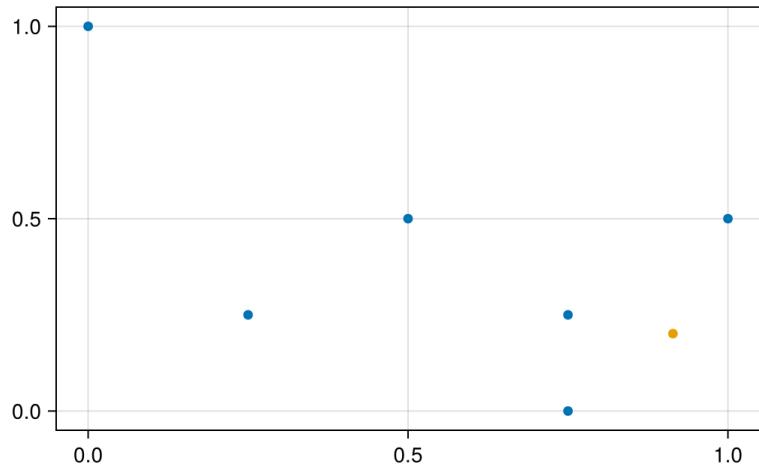
# Display results
println("Query Point: $query_point")
println("Nearest Neighbors Indices: $indices")
println("Distances to Neighbors: $distances")

f = Figure()
Axis(f[1, 1])
scatter!(df_num[:, 1], df_num[:, 2])
scatter!(query_point[1], query_point[2])
f

Query Point: [0.9140433698197334, 0.20115411931004723]
Nearest Neighbors Indices: [2]
Distances to Neighbors: [0.6938418960519378]

[ Warning: Found 'resolution' in the theme when creating a 'Scene'. The
  @ Makie ~/.julia/packages/Makie/iRM0c/src/scenes.jl:220
```

23.6. Benchmarking



23.6. Benchmarking

24. Portfolio Optimization

24.1. In This Chapter

Optimization in a portfolio context with examples of asset selection under different constraints and objectives.

24.2. Setup

```
using CSV, DataFrames
using JuMP, Ipopt, LinearAlgebra
using CairoMakie
```

24.3. The Data

```
μ = [0.1, 0.15, 0.12] # returns
ρ = [0.1 0.05 0.03;
      0.05 0.12 0.04;
      0.03 0.04 0.08] # covariances
n_a = length(μ) # number of assets
```

3

24.4. Theory

Harry Markowitz introduced the modern portfolio theory in 1952. The main idea is that investors are pursuing to maximize their expected return of a portfolio given a certain amount of risk. By definition any portfolio yielding a higher return must

24. Portfolio Optimization

have higher amount of risk, so there is a trade-off between desired expected returns and allowable risks. The risk versus maximized expected return relationship can be plotted out as a curve, a.k.a. the efficient frontier.

24.5. Mathematical tools

24.5.1. 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.

$$\begin{aligned} & \text{minimize} && w^T \Sigma w \\ & \text{subject to} && R^T \geq \mu_{\text{target}} \\ & && 1^T w = 1 \\ & && w \geq 0 \end{aligned}$$

```
# Create an optimization model
model = Model(optimizer_with_attributes(Ipopt.Optimizer, "print_level" = 0))
# Set up weights as variables to optimize
@variable(model, w[1:n_a] >= zero(0.0))
# Objective: minimize portfolio variance
@objective(model, Min, sum(w[i] * rho[i, j] * w[j] for i in 1:n_a, j in 1:n_a))
# Constraints: Sum of portfolio weights should equal to 1, and all weights should be non-negative
@constraint(model, sum(w) == 1)
# May also add additional constraints
# target_return = 0.1
# @constraint(model, dot(mu, w) >= target_return)
# Solve the optimization problem
optimize!(model)
# Print results
@show "Optimal Portfolio Weights:"
for i = 1:n_a
```

24.5. Mathematical tools

```
@show ("Asset ", i, ": ", value.(w)[i])
end

*****
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 Portfolio Weights:" = "Optimal Portfolio Weights:"
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 1, ": ", 0.3333333012309821)
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 2, ": ", 0.16666675086886984)
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 3, ": ", 0.4999999479001481)
```

24.5.2. 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.

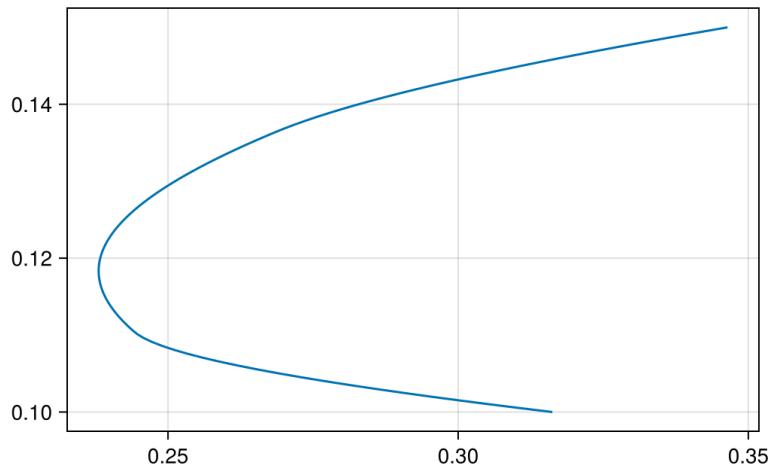
```
# Create an optimization model
model = Model(optimizer_with_attributes(Ipopt.Optimizer, "print_level" => 0))
# Set up weights as variables to optimize
@variable(model, w[1:n_a] >= zero(0.0))
# Define objective function: minimize portfolio variance
portfolio_variance = w'ρ * w
@objective(model, Min, portfolio_variance)
# Constraints: Sum of portfolio weights should equal to 1, and all weights should be zero or positive
@constraint(model, sum(w) == 1)
# Generate a range of target returns
points = 100
target_returns = range(minimum(μ), maximum(μ), length=points)

efficient_frontier = []
for target_return in target_returns
    # Add additional constraint for target return
    @constraint(model, c, dot(μ, w) == target_return)
```

24. Portfolio Optimization

```
# Solve the problem
optimize!(model)
# Show solution
if termination_status(model) == MOI.LOCALLY_SOLVED
    push!(efficient_frontier, (sqrt(objective_value(model)), t))
end
unregister(model, :c)
delete(model, c)
end
# Plot Efficient Frontier
fig = Figure()
Axis(fig[1, 1])
lines!(map(x → x[1], efficient_frontier), map(x → x[2], efficient_frontier))
fig
```

Warning: Found 'resolution' in the theme when creating a 'Scene'. This will be ignored.
@ Makie ~/.julia/packages/Makie/iRM0c/src/scenes.jl:220



24.5.3. 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

24.5. Mathematical tools

incorporates subjective opinions while maintaining diversification and risk management principles.

$$\text{maximize } \mu^T w - \lambda \cdot \frac{1}{2} w^T \Sigma w$$

$$\text{subject to } \sum_{i=1}^N w_i = 1$$

$$w_i \geq 0, \quad \forall i$$

```

λ = 2.5 # risk aversion
rfr = 0.02 # risk free rate
# Market equilibrium parameters (prior)
μ_market = [0.08, 0.08, 0.08] # Market equilibrium return
Σ_market = ρ # Market equilibrium covariance matrix
# Investor views
Q = μ # Expected returns on assets according to investor views
P = [1 0 0; 0 1 0; 0 0 1]      # Pick matrix specifying which assets views are on
Ω = [0.001^2 0.0 0.0; 0.0 0.002^2 0.0; 0.0 0.0 0.003^2] # Views uncertainty (covariance matrix)

# Create an optimization model
model = Model(optimizer_with_attributes(Ipopt.Optimizer, "print_level" => 0))
# Set up weights as variables to optimize
@variable(model, w[1:n_a] >= zero(0.0))
# Black-Litterman expected return adjustment
Σ_prior_inv = inv(Σ_market)
τ = 0.05 # Scaling factor
# Calculate the posterior expected returns
μ_posterior = Σ_prior_inv * (τ * Σ_market * (Σ_prior_inv + P' * inv(Ω) * P)) \
              (τ * Σ_market * (Σ_prior_inv * μ_market + P' * inv(Ω) * Q) + Σ_prior_inv * μ_market)
# Objective: maximize sharpe ratio
sr = (w' * μ_posterior - rfr) / (λ / 2 * w' * Σ_market * w)
@objective(model, Max, sr)
# Constraints: Sum of portfolio weights should equal to 1, and all weights should be zero or positive
@constraint(model, sum(w) == 1)
# Solve the optimization problem
optimize!(model)
# Print results
v = sqrt(value.(w)' * Σ_market * value.(w))
@show "Optimal Portfolio Weights, Expected Portfolio Return, Portfolio Volatility:", v

```

24. Portfolio Optimization

```
for i = 1:n_a
    @show ("Asset ", i, ": ", value.(w)[i], value.(w)[i] * μ_posterior[i])
end

("Optimal Portfolio Weights, Expected Portfolio Return, Portfolio Volatility")
("Asset ", i, ": ", value.(w)[i], value.(w)[i] * μ_posterior[i]) = ("Asset ", i, ": ", value.(w)[i], value.(w)[i] * μ_posterior[i])
("Asset ", i, ": ", value.(w)[i], value.(w)[i] * μ_posterior[i]) = ("Asset ", i, ": ", value.(w)[i], value.(w)[i] * μ_posterior[i])
```

24.5.4. 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.

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^N (w_i \cdot \sqrt{\sigma_i})^2 \\ \text{subject to} \quad & \sum_{i=1}^N w_i = 1 \\ & w_i \geq 0, \quad \forall i \end{aligned}$$

```
# Create an optimization model
model = Model(optimizer_with_attributes(Ipopt.Optimizer, "print_level" = 0))
# Set up weights as variables to optimize
@variable(model, w[1:n_a] >= zero(0.0))
# Objective: minimize portfolio variance
portfolio_variance = w'ρ * w
margin = (ρ * w ./ sqrt(portfolio_variance)) .* w
risk_contributions = margin ./ sum(margin)
target = repeat([1.0 / n_a], n_a)
@objective(model, Max, sum((risk_contributions .- target) .^ 2))
# Constraints: Sum of portfolio weights should equal to 1, and all weights must be non-negative
@constraint(model, sum(w) == 1)
```

24.5. Mathematical tools

```
# Solve the optimization problem
optimize!(model)
# Print results
@show "Optimal Portfolio Weights:"
for i = 1:n_a
    @show ("Asset ", i, ": ", value.(w)[i])
end

"Optimal Portfolio Weights:" = "Optimal Portfolio Weights:"
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 1, ": ", -6.957484531612737e-9)
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 2, ": ", 1.0000000131375544)
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 3, ": ", -6.180069741122123e-9)
```

24.5.5. 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.

$$\begin{aligned} \text{maximize} \quad & \frac{E[R_p] - R_f}{\sigma_p} \\ \text{subject to} \quad & \sum_{i=1}^N w_i = 1 \\ & w_i \geq 0, \quad \forall i \end{aligned}$$

```
# Create an optimization model
model = Model(optimizer_with_attributes(Ipopt.Optimizer, "print_level" => 0))
# Set up weights as variables to optimize
@variable(model, w[1:n_a] >= zero(0.0))
# Objective: minimize portfolio variance
rfr = 0.05 # risk free rate
@objective(model, Max, (dot(mu, w) - rfr) / sqrt(sum(w[i] * rho[i, j] * w[j] for i in 1:n_a, j in 1:n_a)))
# Constraints: Sum of portfolio weights should equal to 1, and all weights should be zero or positive
@constraint(model, sum(w) == 1)
```

24. Portfolio Optimization

```
# Solve the optimization problem
optimize!(model)
# Print results
@show "Optimal Portfolio Weights:"
for i = 1:n_a
    @show ("Asset ", i, ": ", value.(w)[i])
end

"Optimal Portfolio Weights:" = "Optimal Portfolio Weights:"
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 1, ": ", 0.010841995514)
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 2, ": ", 0.535229231810)
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 3, ": ", 0.453928772674)
```

24.5.6. 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. Robust optimization may involve incorporating stress tests, scenario analysis, or robust risk measures into the portfolio construction process.

$$\begin{aligned} & \text{minimize} && w^T \Sigma w + \gamma \|w - w_0\|_2^2 \\ & \text{subject to} && \sum_{i=1}^N w_i = 1 \\ & && w_i \geq 0, \quad \forall i \\ & && \|(\Sigma^{1/2}(w - w_0))\|_2 \leq \epsilon \end{aligned}$$

```
# Create an optimization model
model = Model(optimizer_with_attributes(Ipopt.Optimizer, "print_level" = 0))
# Set up weights as variables to optimize
@variable(model, w[1:n_a] >= zero(0.0))
# Objective: minimize portfolio variance
ε = 0.05 # Uncertainty level
γ = 0.1 # Robustness parameter
w₀ = [0.3, 0.4, 0.3]
```

24.5. Mathematical tools

```
@objective(model, Min, dot(w, ρ * w) + γ * sum((w[i] - w₀[i])^2 for i in 1:nₐ))
# 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((ρ[i, j] * (w[i] - w₀[i]) * (w[j] - w₀[j]))) for i in 1:nₐ, j in 1:nₐ) <= ε
# Solve the optimization problem
optimize!(model)
# Print results
@show "Optimal Portfolio Weights:"
for i = 1:nₐ
    @show ("Asset ", i, ": ", value.(w)[i])
end

"Optimal Portfolio Weights:" = "Optimal Portfolio Weights:"
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 1, ": ", 0.31250000098314346)
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 2, ": ", 0.31250000376951037)
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 3, ": ", 0.37499999524734623)
```


25. Bayesian Mortality Modeling

! Drafting Notes

Ideas: - First plot graph of outcomes and discuss some key features: - More variance when the subset are smaller

- Just plot the data first, with the group colors, to explain what we are looking at and for consistency in subsequent plots
- Generate sample data using parameters sampled from chain and show the bands on the associated outcomes.

25.1. 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 for 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

25. Bayesian Mortality Modeling

- 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 UUIDs
using DataFramesMeta
using MCMCChains
using LinearAlgebra
using CairoMakie
using StatsBase
using OffsetArrays

n = 10_000
inforce = map(1:n) do i
    (
        id=uuid1(),
        issue_age=rand(30:70),
        risk_level=rand(1:3),
    )
end

10000-element Vector{@NamedTuple{id::UUID, issue_age::Int64, risk_l
(id = UUID("2caad4fe-3da7-11ef-20aa-b9dd0f9791cf"), issue_age = 39,
(id = UUID("2caad51e-3da7-11ef-230b-bb3232827e59"), issue_age = 57,
(id = UUID("2caad528-3da7-11ef-3e86-6d5e10d2fb6a"), issue_age = 67,
(id = UUID("2caad528-3da7-11ef-312a-bfec924bdf31"), issue_age = 49,
(id = UUID("2caad528-3da7-11ef-0345-397078c003b5"), issue_age = 36,
(id = UUID("2caad532-3da7-11ef-0d20-6d71bbe9adfd"), issue_age = 38,
(id = UUID("2caad532-3da7-11ef-3b53-fd57f99488b2"), issue_age = 65,
(id = UUID("2caad532-3da7-11ef-37d2-4b61992e4bc5"), issue_age = 35,
(id = UUID("2caad53c-3da7-11ef-0d7d-2d361b993d64"), issue_age = 38,
(id = UUID("2caad53c-3da7-11ef-3a1d-67c1a114476b"), issue_age = 57,
:
(id = UUID("2cab500a-3da7-11ef-16eb-05bbdfcba9e0"), issue_age = 47,
(id = UUID("2cab5016-3da7-11ef-31ce-cd184f74fc71"), issue_age = 53,
```

25.1. Generating fake data

```
(id = UUID("2cab5016-3da7-11ef-1e25-19b07675798e"), issue_age = 38, risk_level = 1)
(id = UUID("2cab5016-3da7-11ef-090e-d91fa97a17bb"), issue_age = 58, risk_level = 2)
(id = UUID("2cab5016-3da7-11ef-0347-ad106f089234"), issue_age = 70, risk_level = 2)
(id = UUID("2cab5020-3da7-11ef-0b35-6fea118830d1"), issue_age = 63, risk_level = 3)
(id = UUID("2cab5020-3da7-11ef-30ac-ad5d4f6409cf"), issue_age = 52, risk_level = 3)
(id = UUID("2cab5020-3da7-11ef-1aeb-0f09d2f57ffc"), issue_age = 42, risk_level = 2)
(id = UUID("2cab502a-3da7-11ef-010e-054906753474"), issue_age = 58, risk_level = 1)

base_table = MortalityTables.table("2001 VBT Residual Standard Select and Ultimate - Male Nonsmoker")

function tabular_mortality(params, issue_age, att_age, risk_level)
    q = params.ultimate[att_age]
    if risk_level == 1
        q *= 0.7
    elseif risk_level == 2
        q = q
    else
        q *= 1.5
    end
end

tabular_mortality (generic function with 1 method)

function model_outcomes(inforce, assumption, assumption_params; n_years=5)

    outcomes = map(inforce) do pol
        alive = 1
        sim = map(1:n_years) do t
            att_age = pol.issue_age + t - 1
            q = assumption(
                assumption_params,
                pol.issue_age,
                att_age,
                pol.risk_level
            )
            if rand() < q
                out = (att_age=att_age, exposures=alive, death=1)
                alive = 0
                out
            else
                (att_age=att_age, exposures=alive, death=0)
            end
        end
    end
end
```

25. Bayesian Mortality Modeling

```
        end
    end
    filter!(x → x.exposures == 1, sim)

end

df = DataFrame(inforce)

df.outcomes = outcomes
df = flatten(df, :outcomes)

df.att_age = [x.att_age for x in df.outcomes]
df.death = [x.death for x in df.outcomes]
df.exposures = [x.exposures for x in df.outcomes]
select!(df, Not(:outcomes))

end

exposures = model_outcomes(inforce, tabular_mortality, base_table)
data = combine(groupby(exposures, [:issue_age, :att_age])) do subdf
    exposures=nrow(subdf),
    deaths=sum(subdf.death),
    fraction=sum(subdf.death) / nrow(subdf)
end

data2 = combine(groupby(exposures, [:issue_age, :att_age, :risk_level])) do subdf
    exposures=nrow(subdf),
    deaths=sum(subdf.death),
    fraction=sum(subdf.death) / nrow(subdf)
end
```

25.2. 1: A single binomial parameter model

	issue_age	att_age	risk_level	exposures	deaths	fraction
	Int64	Int64	Int64	Int64	Int64	Float64
1	30	30	1	68	0	0.0
2	30	30	2	93	0	0.0
3	30	30	3	83	0	0.0
4	30	31	1	68	0	0.0
5	30	31	2	93	0	0.0
6	30	31	3	83	0	0.0
7	30	32	1	68	0	0.0
8	30	32	2	93	0	0.0
9	30	32	3	83	0	0.0
10	30	33	1	68	0	0.0
11	30	33	2	93	0	0.0
12	30	33	3	83	0	0.0
13	30	34	1	68	0	0.0
14	30	34	2	93	0	0.0
15	30	34	3	83	0	0.0
16	31	31	1	81	0	0.0
17	31	31	2	77	0	0.0
18	31	31	3	83	1	0.0120482
19	31	32	1	81	0	0.0
20	31	32	2	77	0	0.0
21	31	32	3	82	1	0.0121951
22	31	33	1	81	0	0.0
23	31	33	2	77	0	0.0
24	31	33	3	81	0	0.0
...

25.2. 1: A single binomial parameter model

Estiamte q , the average mortality rate, not accounting for any variation within the population/sample. Our model is defines as:

$$q \sim \text{Beta}(1, 1)$$

```
@model function mortality(data, deaths)
    q ~ Beta(1, 1)
    for i = 1:nrow(data)
```

25. Bayesian Mortality Modeling

```

    deaths[i] ~ Binomial(data.exposures[i], q)
end
end

m1 = mortality(data, data.deaths)

DynamicPPL.Model{typeof(mortality), (:data, :deaths), (), (), Tuple{  

Row | issue_age  att_age  exposures  deaths  fraction  

     | Int64      Int64      Int64      Int64      Float64  

-----  

1   |       30       30       244       0       0.0  

2   |       30       31       244       0       0.0  

3   |       30       32       244       0       0.0  

4   |       30       33       244       0       0.0  

5   |       30       34       244       0       0.0  

6   |       31       31       241       1       0.00414938  

7   |       31       32       240       1       0.00416667  

8   |       31       33       239       0       0.0  

⋮   |       ⋮       ⋮       ⋮       ⋮       ⋮  

199  |      69       72       198       6       0.030303  

200  |      69       73       192       7       0.0364583  

201  |      70       70       283      16      0.0565371  

202  |      70       71       267       5       0.0187266  

203  |      70       72       262       4       0.0152672  

204  |      70       73       258       4       0.0155039  

205  |      70       74       254      11      0.0433071  


```

190 rows omitted, deaths = [0, 0, 0, 0, 0, 0,

25.2.1. 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 (400×13×4 Array{Float64, 3}):

Iterations = 201:1:600

25.2. 1: A single binomial parameter model

```
Number of chains = 4
Samples per chain = 400
Wall duration     = 1.88 seconds
Compute duration  = 7.43 seconds
parameters        = q
internals         = lp, n_steps, is_accept, acceptance_rate, log_density, hamiltonian_energy, hamilton

Summary Statistics
parameters   mean      std      mcse    ess_bulk  ess_tail    rhat  e...
Symbol       Float64  Float64  Float64  Float64  Float64  Float64  ...
q           0.0075  0.0004  0.0000  839.9381 977.5464  1.0031  ...
1 column omitted

Quantiles
parameters   2.5%    25.0%   50.0%   75.0%   97.5%
Symbol       Float64  Float64  Float64  Float64  Float64
q           0.0068  0.0073  0.0075  0.0078  0.0083
```

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:

```
sum(data.deaths) / sum(data.exposures)
```

```
0.0075447207236441815
```

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:

25. Bayesian Mortality Modeling

```
let
    data_weight = log.(data.exposures)
    #data_weight = .√(data_weight ./ maximum(data_weight) .* 20)
    f = Figure(title="Parametric Bayesian Mortality"
    )
    ax = Axis(f[1, 1],
        xlabel="age",
        ylabel="mortality rate",
        # ylims=(0.0, 0.25),
    )
    scatter!(ax,
        data.att_age,
        data.fraction,
        markersize=data_weight,
        color=(:blue, 0.5),
        label="Experience data point (size indicates relative expo
    )

    # show n samples from the posterior plotted on the graph
    n = 300
    ages = sort!(unique(data.att_age))

    q_posterior = sample(chain, n)[:q]

    for i in 1:n
        hlines!(ax, [q_posterior[i]], color(:grey, 0.1))
    end

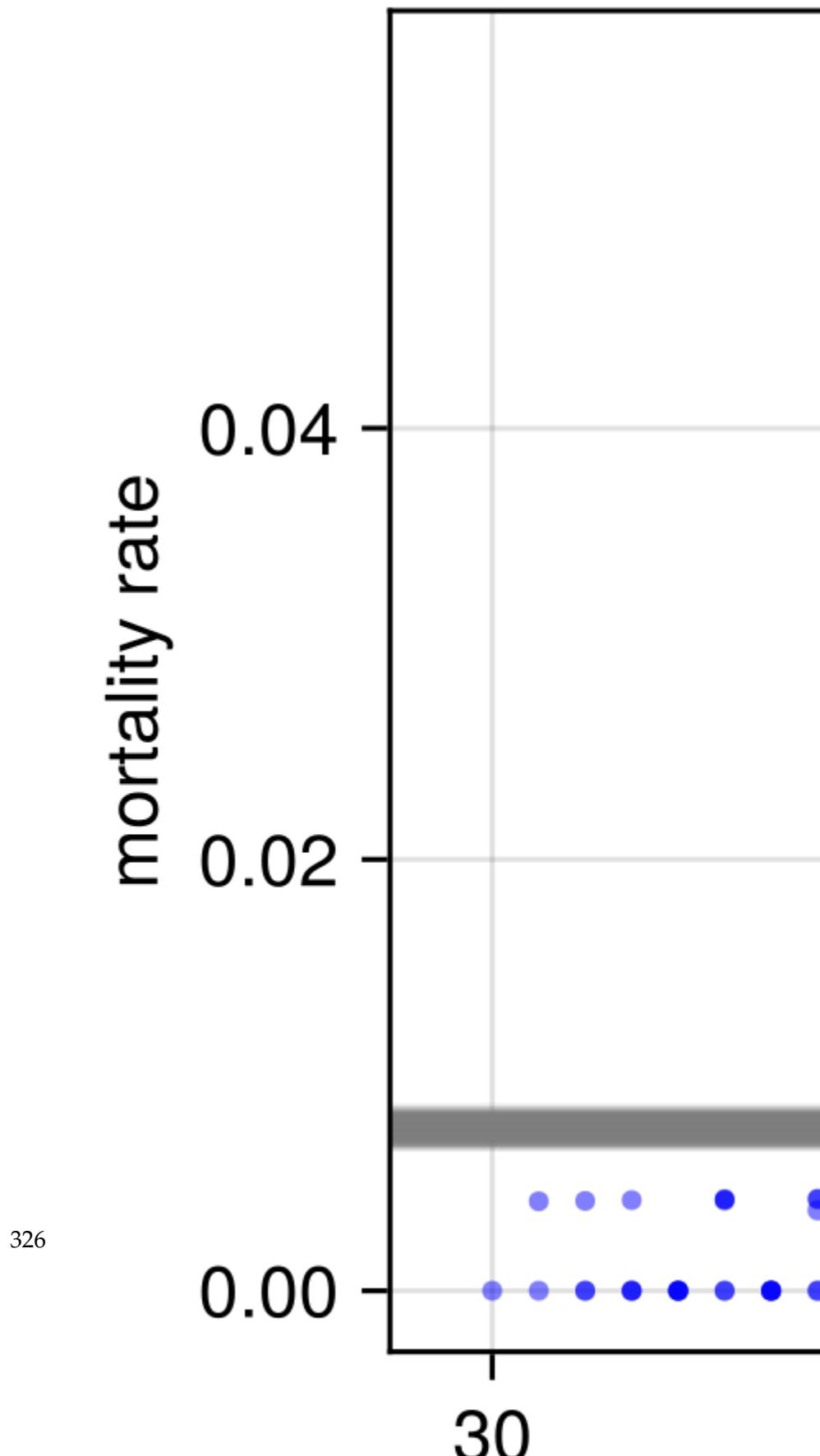
    # Need to simulate at individual level and then aggregate?

    sim05 = Float64[]
    sim95 = Float64[]
    for r in eachrow(data)
        outcomes = map(1:n) do i
            rand(Binomial(r.exposures, q_posterior[i]), 500)
        end
        push!(sim05, quantile(Iterators.flatten(outcomes), 0.05) /
        push!(sim95, quantile(Iterators.flatten(outcomes), 0.95) /
```

25.2. 1: A single binomial parameter model

end

f
end



25.3. 2. Parametric model

```
let
    n = 300
    q_posterior = sample(chain, n)[:q]

end

2-dimensional AxisArray{Float64,2,...} with axes:
  :iter, 1:300
  :chain, 1:1
And data, a 300×1 Matrix{Float64}:
0.0070217431534508
0.007849324866488181
0.00778886143908542
0.0075689663549648595
0.007786890734769192
0.00781320800096659
0.007605114932699948
0.006628112145377278
0.008233013910200999
0.007561848204925246
⋮
0.007783819684308208
0.007611236683016277
0.007463293775787834
0.0074743270678487034
0.007197137411204225
0.007222420179006868
0.007056157632167371
0.00811568004244103
0.007532710869657816
```

25.3. 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 (ie the probability of a death at a given age), so the value must be constrained to be in the interval between zero and one.

25. Bayesian Mortality Modeling

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(data, deaths)
    a ~ Exponential(0.1)
    b ~ Exponential(0.1)
    c = 0.0
    k ~ truncated(Exponential(1), 1, Inf)

    # use the variables to create a parametric mortality model
    m = MortalityTables.MakehamBeard(; a, b, c, k)

    # loop through the rows of the dataframe to let Turing observe
    # and how consistent the parameters are with the data
    for i = 1:nrow(data)
        age = data.att_age[i]
        q = MortalityTables.hazard(m, age)
        deaths[i] ~ Binomial(data.exposures[i], q)
    end
end

mortality2 (generic function with 2 methods)
```

We combine the model with the data and sample from the posterior using a similar call as before:

```
m2 = mortality2(data, data.deaths)

chain2 = sample(m2, 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      = 6.41 seconds
```

25.3. 2. Parametric model

```
Compute duration = 25.21 seconds
parameters      = a, b, k
internals       = lp, n_steps, is_accept, acceptance_rate, log_density, hamiltonian_energy, hamilton

Summary Statistics
parameters    mean      std      mcse   ess_bulk   ess_tail     rhat  e ...
Symbol      Float64  Float64  Float64  Float64  Float64  Float64  ...
a          0.0000  0.0000  0.0000  519.1440  524.9383  1.0094  ...
b          0.1027  0.0064  0.0003  526.0442  524.7989  1.0098  ...
k          1.9355  0.9155  0.0308  618.0445  459.0567  1.0062  ...
                                         1 column omitted

Quantiles
parameters    2.5%    25.0%    50.0%    75.0%    97.5%
Symbol      Float64  Float64  Float64  Float64  Float64
a          0.0000  0.0000  0.0000  0.0000  0.0000
b          0.0901  0.0985  0.1029  0.1068  0.1155
k          1.0192  1.2693  1.6739  2.2757  4.5308

summarize(chain2)
plot(chain2)
```

25.3.1. Plotting samples from the posterior

We can see that the sampling of possible posterior parameters fits the data well:

```
let
  data_weight = data.exposures ./ sum(data.exposures)
  data_weight = ./(data_weight ./ maximum(data_weight) .* 20)

  p = scatter(
    data.att_age,
    data.fraction,
    markersize=data_weight,
    alpha=0.5,
    label="Experience data point (size indicates relative exposure quantity)",
```

25. Bayesian Mortality Modeling

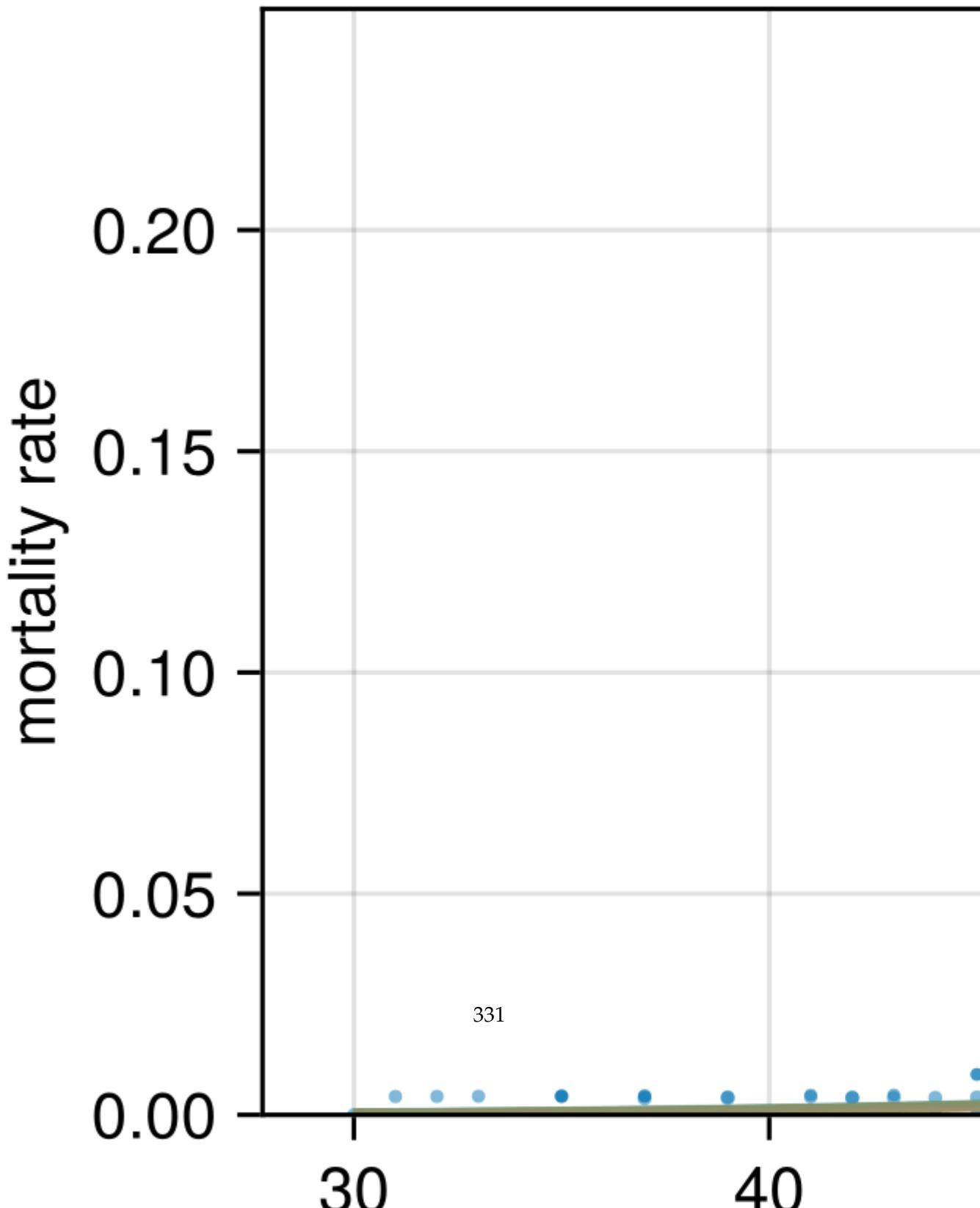
```
axis=(
    xlabel="age",
    limits=(nothing, nothing, 0.0, 0.25),
    ylabel="mortality rate",
    title="Parametric Bayesian Mortality"
)
)

# show n samples from the posterior plotted on the graph
n = 300
ages = sort!(unique(data.att_age))

for i in 1:n
    s = sample(chain2, 1)
    a = only(s[:a])
    b = only(s[:b])
    k = only(s[:k])
    c = 0
    m = MortalityTables.MakehamBeard(; a, b, c, k)
    lines!(ages, age → MortalityTables.hazard(m, age), alpha=
end
p
end
```

25.3. 2. Parametric model

Param



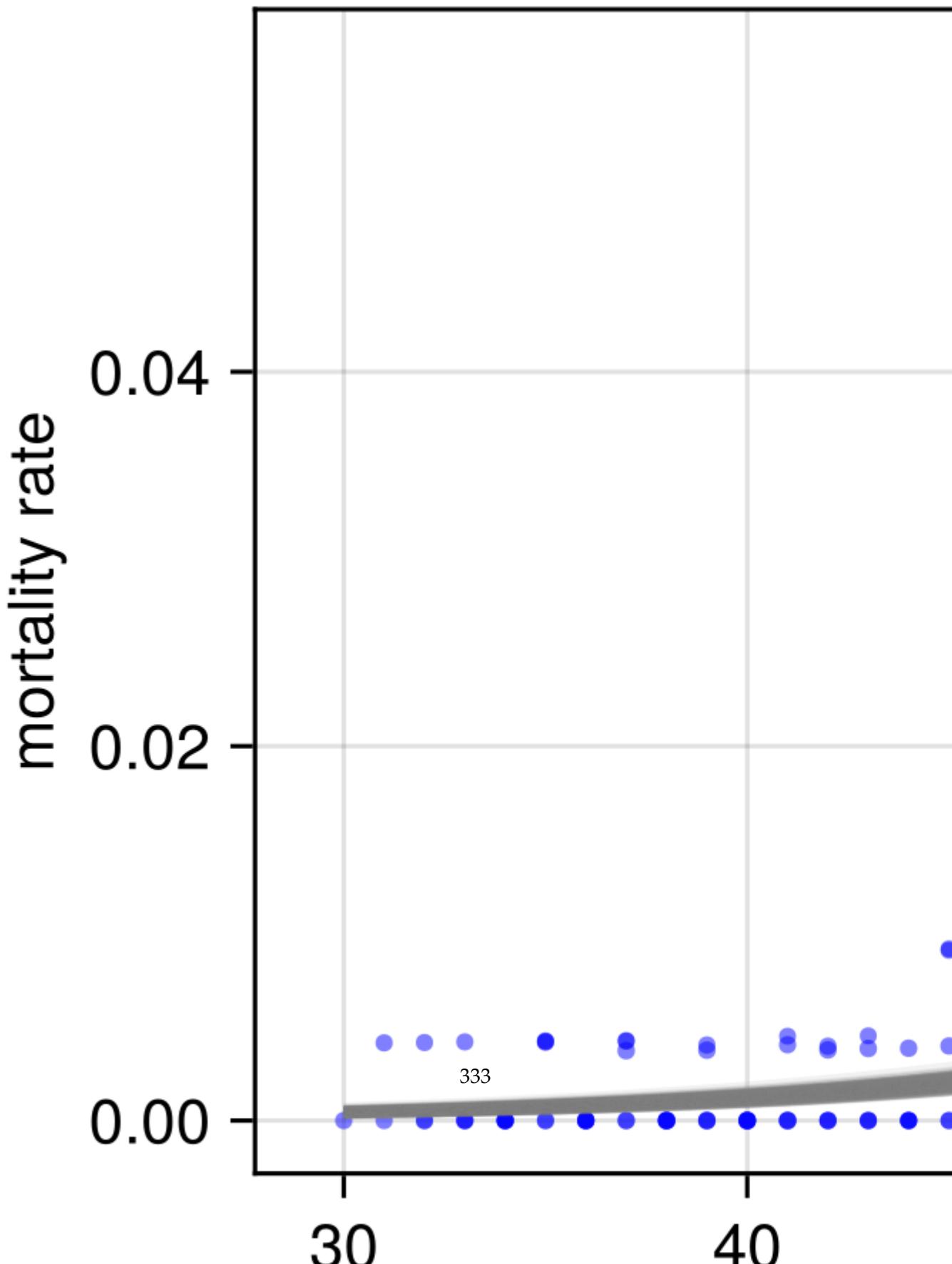
25. Bayesian Mortality Modeling

```
let
    data_weight = log.(data.exposures)
    #data_weight = .√(data_weight ./ maximum(data_weight) .* 20)
    f = Figure(title="Parametric Bayesian Mortality"
    )
    ax = Axis(f[1, 1],
        xlabel="age",
        ylabel="mortality rate",
        # ylims=(0.0, 0.25),
    )
    scatter!(ax,
        data.att_age,
        data.fraction,
        markersize=data_weight,
        color=(:blue, 0.5),
        label="Experience data point (size indicates relative expo
    )

    # show n samples from the posterior plotted on the graph
    n = 300
    ages = sort!(unique(data.att_age))

    for i in 1:n
        s = sample(chain2, 1)
        a = only(s[:a])
        b = only(s[:b])
        k = only(s[:k])
        c = 0
        m = MortalityTables.MakehamBeard(; a, b, c, k)
        qs = MortalityTables.hazard.(m, ages)
        lines!(ax, ages, qs, color(:grey, 0.1))
    end
    f
end
```

25.3. 2. Parametric model



25.4. 3. Parametric model

This model extends the prior to create a multi-level model. Each risk class (`risk_level`) gets its own a parameter in the MakhamBeard model. The prior for a_i is determined by the hyperparameter \bar{a} .

```
@model function mortality3(data, deaths)
    risk_levels = length(levels(data.risk_level))
    b ~ Exponential(0.1)
    ā ~ Exponential(0.1)
    a ~ filldist(Exponential(ā), risk_levels)
    c = 0
    k ~ truncated(Exponential(1), 1, Inf)

    # use the variables to create a parametric mortality model

    # loop through the rows of the dataframe to let Turing observe
    # and how consistent the parameters are with the data
    for i = 1:nrow(data)
        risk = data.risk_level[i]

        m = MortalityTables.MakehamBeard(; a=a[risk], b, c, k)
        age = data.att_age[i]
        q = MortalityTables.hazard(m, age)
        deaths[i] ~ Binomial(data.exposures[i], q)
    end
    end

    m3 = mortality3(data2, data2.deaths)

    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      0.1038   0.0064   0.0004   221.5622  237.7339   1.0007
```

25.4. 3. Parametric model

	\bar{a}	0.0001	0.0003	0.0000	231.5781	207.9352	0.9999	...
a[1]	0.0000	0.0000	0.0000	229.4901	251.0185	1.0019	...	
a[2]	0.0000	0.0000	0.0000	232.9706	293.5854	1.0011	...	
a[3]	0.0000	0.0000	0.0000	224.7022	277.8057	1.0010	...	
k	1.9531	0.8981	0.0310	536.8096	479.1340	1.0037	...	
								1 column omitted

```

let data = data2

data_weight = data.exposures ./ sum(data.exposures)
data_weight = .sqrt(data_weight ./ maximum(data_weight) .* 20)
color_i = data.risk_level

p = scatter(
    data.att_age,
    data.fraction,
    markersize=data_weight,
    alpha=0.5,
    color=color_i,
    label="Experience data point (size indicates relative exposure quantity)",
    axis=(
        xlabel="age",
        limits=(nothing, nothing, 0.0, 0.25),
        ylabel="mortality rate",
        title="Parametric Bayesian Mortality"
    )
)

# show n samples from the posterior plotted on the graph
n = 100

ages = sort!(unique(data.att_age))
for r in 1:3
    for i in 1:n
        s = sample(chain3, 1)
        a = only(s[Symbol("a[$r]")])
        b = only(s[:b])
        k = only(s[:k])
        c = 0
        m = MortalityTables.MakehamBeard(; a, b, c, k)
    end
end

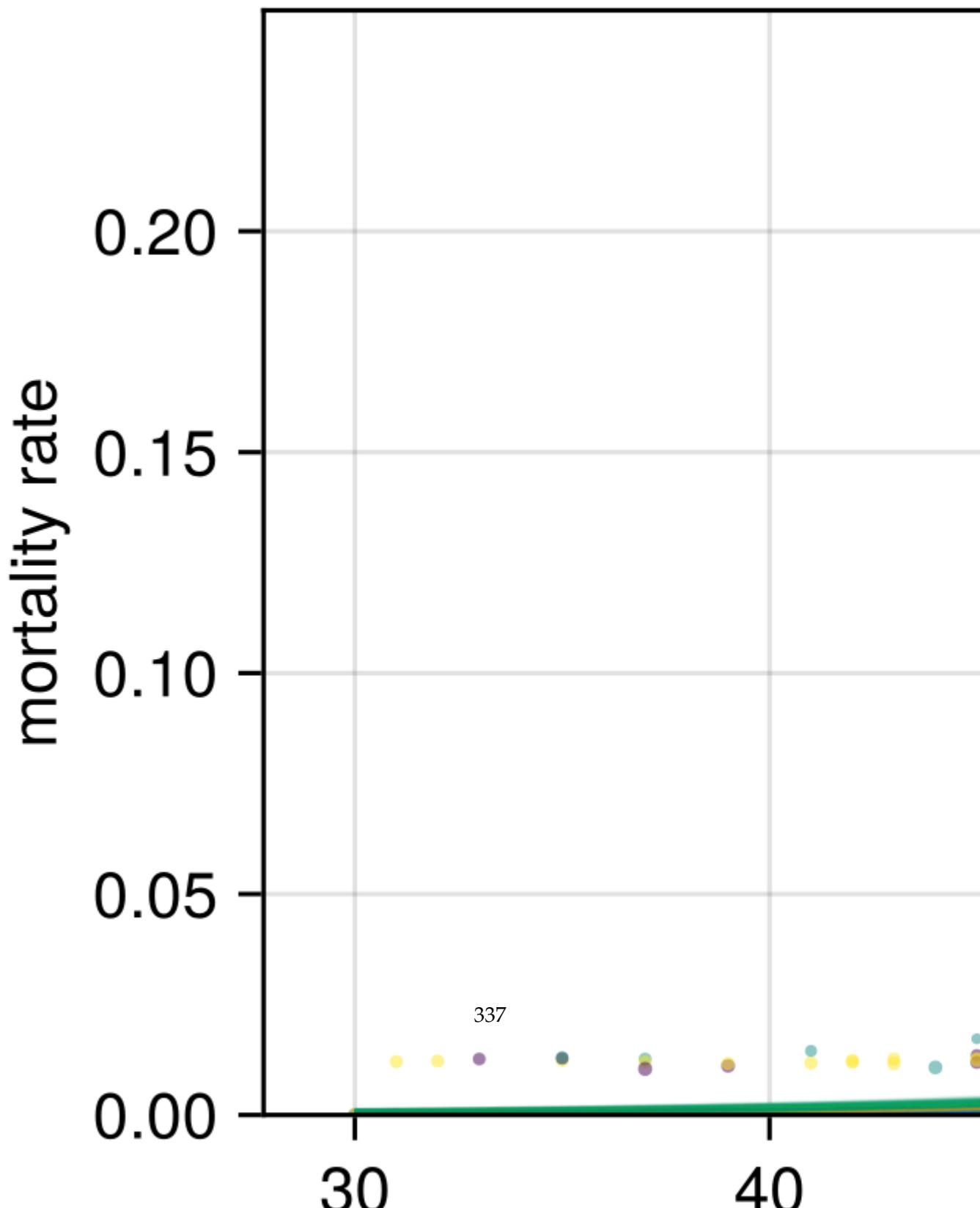
```

25. Bayesian Mortality Modeling

```
if i == 1
    lines!(ages, age → MortalityTables.hazard(m, age))
else
    lines!(ages, age → MortalityTables.hazard(m, age))
end
end
p
end
```

25.4. 3. Parametric model

Param



25.5. Handling non-unit exposures

The key is to use the Poisson distribution, which is a continuous approximation to the Binomial distribution:

```
@model function mortality4(data, deaths)
    risk_levels = length(levels(data.risk_level))
    b ~ Exponential(0.1)
    ā ~ Exponential(0.1)
    a ~ filldist(Exponential(ā), risk_levels)
    c ~ Beta(4, 18)
    k ~ truncated(Exponential(1), 1, Inf)

    # use the variables to create a parametric mortality model

    # loop through the rows of the dataframe to let Turing observe
    # and how consistent the parameters are with the data
    for i = 1:nrow(data)
        risk = data.risk_level[i]

        m = MortalityTables.MakehamBeard(; a=a[risk], b, c, k)
        age = data.att_age[i]
        q = MortalityTables.hazard(m, age)
        deaths[i] ~ Poisson(data.exposures[i] * q)
    end
end

m4 = mortality4(data2, data2.deaths)

chain4 = sample(m4, NUTS(), 1000)

Chains MCMC chain (1000×19×1 Array{Float64, 3}):
Iterations          = 501:1:1500
Number of chains   = 1
Samples per chain  = 1000
Wall duration      = 30.84 seconds
Compute duration   = 30.84 seconds
parameters         = b, ā, a[1], a[2], a[3], c, k
internals          = lp, n_steps, is_accept, acceptance_rate, log_densit
```

25.5. Handling non-unit exposures

```

Summary Statistics
parameters    mean      std      mcse    ess_bulk   ess_tail     rhat   e ...
  Symbol  Float64  Float64  Float64  Float64  Float64  Float64  ...
b      0.1242  0.0103  0.0008  184.5770  290.6476  1.0044  ...
ā      0.0000  0.0000  0.0000  269.5067  416.8641  0.9998  ...
a[1]   0.0000  0.0000  0.0000  192.6674  294.2482  1.0000  ...
a[2]   0.0000  0.0000  0.0000  181.5184  297.1364  1.0020  ...
a[3]   0.0000  0.0000  0.0000  181.1776  273.7515  1.0022  ...
c      0.0009  0.0003  0.0000  277.6631  450.0358  1.0022  ...
k      2.0818  1.1010  0.0386  539.6107  355.8021  1.0036  ...
                                         1 column omitted

Quantiles
parameters    2.5%    25.0%    50.0%    75.0%    97.5%
  Symbol  Float64  Float64  Float64  Float64  Float64
b      0.1052  0.1171  0.1234  0.1309  0.1457
ā      0.0000  0.0000  0.0000  0.0000  0.0001
a[1]   0.0000  0.0000  0.0000  0.0000  0.0000
a[2]   0.0000  0.0000  0.0000  0.0000  0.0000
a[3]   0.0000  0.0000  0.0000  0.0000  0.0000
c      0.0004  0.0007  0.0009  0.0011  0.0016
k      1.0275  1.3213  1.7264  2.4584  5.1355

risk_factors4 = [mean(chain4[Symbol("a[$f]")]) for f in 1:3]
risk_factors4 ./ risk_factors4[2]

let data = data2

data_weight = data.exposures ./ sum(data.exposures)
data_weight = ./maximum(data_weight) .* 20
color_i = data.risk_level

p = scatter(
  data.att_age,
  data.fraction,
  markersize=data_weight,
  alpha=0.5,

```

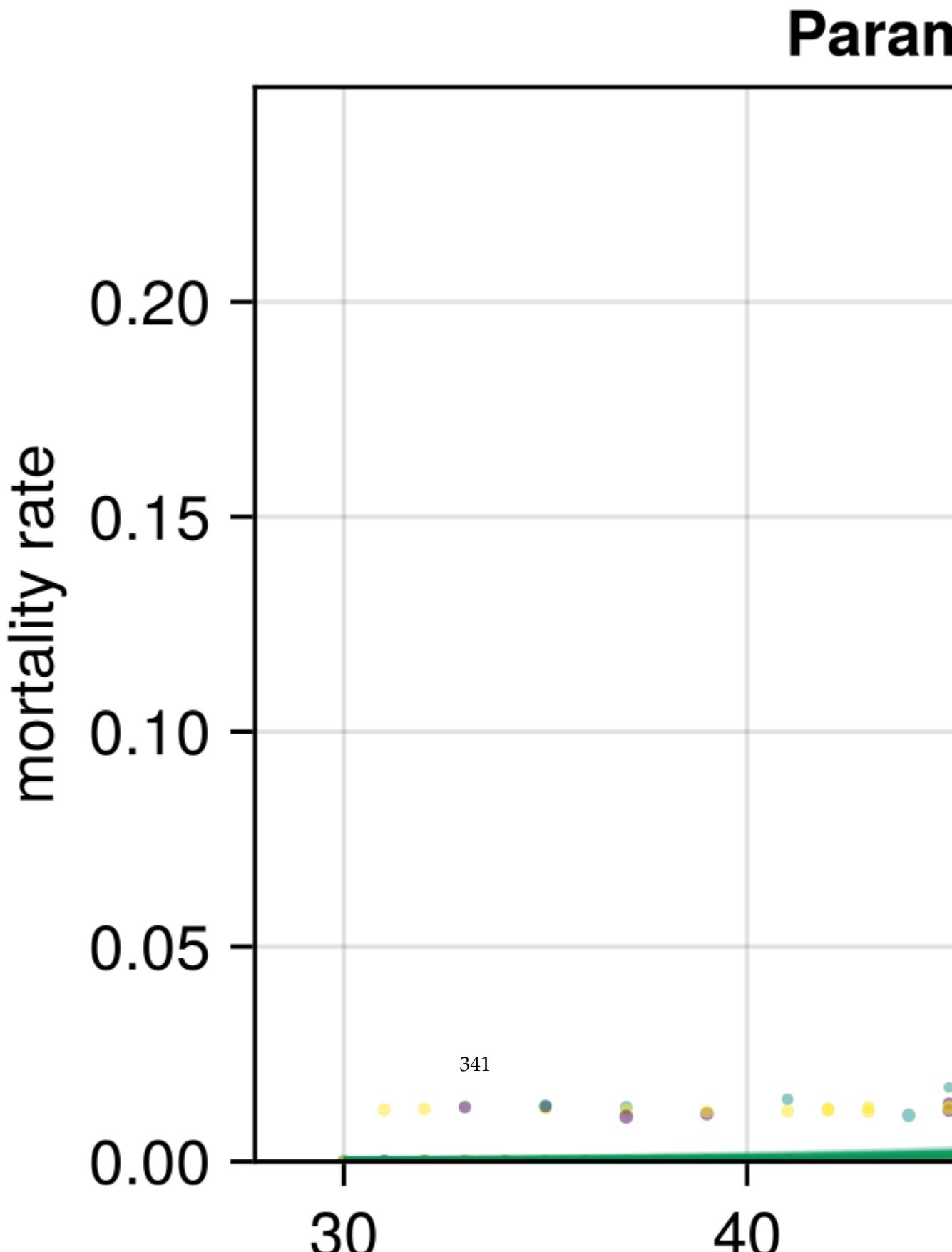
25. Bayesian Mortality Modeling

```
color=color_i,
label="Experience data point (size indicates relative expo
axis=(xlabel="age",
limits=(nothing, nothing, 0.0, 0.25),
ylabel="mortality rate",
title="Parametric Bayseian Mortality"
)
)

# show n samples from the posterior plotted on the graph
n = 100

ages = sort!(unique(data.att_age))
for r in 1:3
    for i in 1:n
        s = sample(chain4, 1)
        a = only(s[Symbol("a[$r]")])
        b = only(s[:b])
        k = only(s[:k])
        c = 0
        m = MortalityTables.MakehamBeard(; a, b, c, k)
        if i == 1
            lines!(ages, age → MortalityTables.hazard(m, age)
        else
            lines!(ages, age → MortalityTables.hazard(m, age)
        end
    end
end
p
end
```

25.5. Handling non-unit exposures



25.6. 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.

```
preds = predict(mortality4(data2, fill(missing, length(data2.deaths)))
```

Chains MCMC chain (1000×615×1 Array{Float64, 3}):

Iterations						
						= 1:1:1000
Number of chains						
						= 1
Samples per chain						
						= 1000
parameters						
						= deaths[1], deaths[2], deaths[3], deaths[4], deaths[5], deaths[6], deaths[7], deaths[8], deaths[9], deaths[10], deaths[11], deaths[12], deaths[13], deaths[14], deaths[15], deaths[16]
internals						
						=
Summary Statistics						
parameters	mean	std	mcse	ess_bulk	ess_tail	rhat
Symbol	Float64	Float64	Float64	Float64	Float64	Float64
deaths[1]	0.0620	0.2413	0.0076	998.3604	NaN	0.9993
deaths[2]	0.1240	0.3446	0.0114	917.2947	925.4944	1.0000
deaths[3]	0.1090	0.3335	0.0109	955.3812	996.8010	1.0000
deaths[4]	0.0580	0.2464	0.0077	1011.5880	1005.1910	0.9993
deaths[5]	0.1230	0.3688	0.0125	862.0197	864.1563	0.9993
deaths[6]	0.1000	0.3195	0.0114	813.2419	868.3968	1.0000
deaths[7]	0.0860	0.2910	0.0092	989.9642	975.7116	0.9993
deaths[8]	0.1060	0.3207	0.0112	814.7000	818.4557	0.9993
deaths[9]	0.1200	0.3431	0.0112	919.6003	900.8674	0.9993
deaths[10]	0.0730	0.2641	0.0085	972.9392	966.7060	1.0000
deaths[11]	0.1150	0.3315	0.0100	1105.5203	1010.1399	1.0000
deaths[12]	0.1100	0.3347	0.0107	967.8766	952.4651	1.0000
deaths[13]	0.0770	0.2778	0.0088	982.0951	975.5811	1.0002
deaths[14]	0.1170	0.3247	0.0108	910.8882	919.0384	0.9993
deaths[15]	0.1130	0.3322	0.0107	956.9732	958.1553	1.0001
deaths[16]	0.0620	0.2494	0.0079	992.5162	986.7350	1.0000

25.6. Predictions

```
deaths[17]  0.0900  0.2966  0.0093  1014.7008  1009.4549  1.0006  ...
:          :      :      :      :      :      :      :
1 column and 598 rows omitted
```

Quantiles

parameters	2.5%	25.0%	50.0%	75.0%	97.5%
Symbol	Float64	Float64	Float64	Float64	Float64
deaths[1]	0.0000	0.0000	0.0000	0.0000	1.0000
deaths[2]	0.0000	0.0000	0.0000	0.0000	1.0000
deaths[3]	0.0000	0.0000	0.0000	0.0000	1.0000
deaths[4]	0.0000	0.0000	0.0000	0.0000	1.0000
deaths[5]	0.0000	0.0000	0.0000	0.0000	1.0000
deaths[6]	0.0000	0.0000	0.0000	0.0000	1.0000
deaths[7]	0.0000	0.0000	0.0000	0.0000	1.0000
deaths[8]	0.0000	0.0000	0.0000	0.0000	1.0000
deaths[9]	0.0000	0.0000	0.0000	0.0000	1.0000
deaths[10]	0.0000	0.0000	0.0000	0.0000	1.0000
deaths[11]	0.0000	0.0000	0.0000	0.0000	1.0000
deaths[12]	0.0000	0.0000	0.0000	0.0000	1.0000
deaths[13]	0.0000	0.0000	0.0000	0.0000	1.0000
deaths[14]	0.0000	0.0000	0.0000	0.0000	1.0000
deaths[15]	0.0000	0.0000	0.0000	0.0000	1.0000
deaths[16]	0.0000	0.0000	0.0000	0.0000	1.0000
deaths[17]	0.0000	0.0000	0.0000	0.0000	1.0000
:	:	:	:	:	:

598 rows omitted

26. Other Useful Techniques

26.1. In this chapter

Other useful techniques are surveyed, such as: memoization to avoid repeated computations, psuedo-monte carlo, creating a model office, and tips on modeling a complete balance sheet.

26.2. Conceptual Techniques

26.2.1. Taking things to the Extreme

Consider what happens if something is taken to an extreme. For example, what happens in the model if we input negative rates? Where should negative rates be allowed and can the model handle them?

26.2.2. Range Bounding

Sometimes you just need to know that an outcome is within a certain range - if you can develop a "high" and "low" estimate by making assumptions that you know are outside of feasible ranges, then you can determine whether something is reasonable or within tolerances.

To take an example from the pages of interview questions: say you need to determine if a mortgaged property's value is greater than the amount of the outstanding loan (say \$100,000). You don't have an appraisal, but know that it's in reasonable condition and that (1) a comparable house with many more issues sold for \$100 per square foot. You also don't know the square footage of the house, but know from

26. Other Useful Techniques

the number of rooms and layout that it must be at least 1000 square feet. Therefore you know that the value should at least be greater than:

$$\frac{\$100}{\text{sq. ft}} \times 1000\text{sq. ft} = \$100,000$$

We'd then conclude that the value of the house very likely exceeds the outstanding balance of the loan and resolves our query without complex modeling or expensive appraisals.

26.3. Modeling Techniques

26.3.1. Serialization

Part VII.

Appendices

27. Set up Julia and the Computing Environment

27.1. Installation

Julia is open source and can be downloaded from [JuliaLang.org](https://julialang.org) and is available for all major operating systems. After you download and install, then you have Julia installed and can access the **REPL**, or Read-Eval-Print-Loop, which can run complete programs or function as powerful day-to-day calculator. However, many people find it more comfortable to work in a text editor or **IDE** (Integrated Development Environment).

If you are looking for managed installations with a curated set of packages for use within an organization, there are ways to self-host package repositories and otherwise administratively manage packages. Julia Computing offers managed support with enterprise solutions, including push-button cloud compute capabilities.

27.2. Package Management

Julia comes with **Pkg**, a built-in package manager. With it, you can install packages, pin certain versions, recreate environments with the same set of dependencies, and upgrade/remove/develop packages easily. It's one of the things that *just works* and makes Julia stand out versus alternative languages that don't have a de-facto way of managing or installing packages.

Package installation is accomplished interactively in the REPL or executing commands.

27. Set up Julia and the Computing Environment

- In the REPL, you can change to the Package Management Mode by hitting] and, e.g., add `DataFrames CSV` to install the two packages. Hit [backspace] to exit that mode in the REPL.
- The same operation without changing REPL modes would be: `using Pkg; Pkg.add(["DataFrames", "CSV"])`

Related to packages, are **environments** which are a self-contained workspaces for your code. This lets you install only packages that are relevant to the current work. It also lets you ‘remember’ the exact set of packages and versions that you used. In fact, you can share the environment with others, and it will be able to recreate the same environment as when you ran the code. This is accomplished via a `Project.toml` file, which tracks the direct dependencies you’ve added, along with details about your project like its version number. The `Manifest.toml` tracks the entire dependency tree.

Reproducibility via the environment tools above is a really key aspect that will ensure Julia code is consistent across time and users, which is important for financial controls.

27.3. Editors

Because Julia is very extensible and amenable to analysis of its own code, you can typically find plugins for whatever tool you prefer to write code in. A few examples:

27.3.1. Visual Studio Code

Visual Studio Code is a free editor from Microsoft. There’s a full-featured Julia plugin available, which will help with auto-completion, warnings, and other code hints that you might find in a dedicated editor (e.g. PyCharm or RStudio). Like those tools, you can view plots, search documentation, show datasets, debug, and manage version control.

27.3.2. Notebooks

Notebooks are typically more interactive environments than text editors - you can write code in cells and see the results side-by-side.

The most popular notebook tool is Jupyter (“Julia, Python, R”). It is widely used and fits in well with exploratory data analysis or other interactive workflows. It can be installed by adding the `IJulia.jl` package.

`Pluto.jl` is a newer tool, which adds 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 contains all the code needed to reproduce results (as long as Julia and Pluto are installed).

27.4. REPL

27.4.1. Help Mode

28. Environment and Package Management

28.1. In This Section

How to effectively utilize environments to ensure consistent and reproducible results. How to use and manage packages. How to create a package and share with others. How to use local registries.

28.2. Projects, Manifests, 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. It uses a couple of files to record this to your project: `Project.toml` and `Manifest.toml`.

28.2.1. 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.

i 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.

28. Environment and Package Management

This is a bit abstract, so here is a quick, annotated tour of an example Project.toml file:

```
name = "FinanceCore"                                     ①
uuid = "b9b1ffdd-6612-4b69-8227-7663be06e089"          ②
authors = ["alecloudenback <alecloudenback@users.noreply.github.com>"]
version = "2.1.0"                                         ③

[deps]
Dates = "ade2ca70-3891-5945-98fb-dc099432e06a"          ④
LoopVectorization = "bdcacae8-1622-11e9-2a5c-532679323890"
Roots = "f2b01f46-fcfa-551c-844a-d8ac1e96c665"

[compat]
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 commit⁴¹.
- 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 The `compat` section defines compatibility with packages 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 28.3 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 rule are note broken.

When adding the dependencies, those packages themselves likely specify their own set of dependencies and Julia must

28.2. Projects, Manifests, and Dependencies

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 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).

28.2.2. 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 in 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.

28.2.3. 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.

28.3. Environments

The combination of `Project.toml` and `Manifest.toml` go 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 time a set of code can be run accurately without requiring the exact same set of dependencies.

Since dependencies can have variation 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 used to download and verify the contents of a file match the 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 11.

28.3. Environments

Environment is meant to mean, in general, the computer you use and software installed in it. When we speak about **environments** in the Julia context, this means the Julia version and packages available to the current Julia code. For example, from the current code is a given package installed and usable?

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>
```

28. Environment and Package Management

The (@1.10) 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 (the global environment, like @1.10 still applies.)

28.4. Packages

28.4.1. Packages versus Projects

28.4.2. Basic Package Structure

28.4.3. Extension Packages

28.5. Regisries

28.5.1. Local Registries

29. The Julia Ecosystem Today

A tour of relevant available packages as of 2023.

The Julia ecosystem favors composability and interoperability, enabled by multiple dispatch. In other words, 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.

As you'll see, Julia packages tend to be less vertically integrated because it's easier to pass data around. Counterexamples of this in Python and R:

- Numpy-compatible packages that are designed to work with a subset of numerically fast libraries in Python
- special functions in Pandas to read CSV, JSON, database connections, etc.
- The Tidyverse in R has a tightly coupled set of packages that works well together but has limitations with some other R packages

Julia is not perfect in this regard, but it's neat to see how frequently things *just work*. It's not magic, but because of Julia features outside the scope of this article it's easy for package developers (and you!) to do this.

Julia also has language-level support for documentation, so packages can follow a consistent style of help-text and have the docs be auto-generated into web pages available locally or online.

The following highlighted packages were chosen for their relevance to typical actuarial work, with a bias towards those used regularly by the authors. This is a small sampling of the over 6000 registered Julia Packages⁴²

⁴² (`time?`) is a simple, built-in function. For true benchmarking purposes, see Section 30.1.

29.0.1. Data

Julia offers a rich data ecosystem with a multitude of available packages. Perhaps at the center of the data ecosystem are `CSV.jl` and `DataFrames.jl`. `CSV.jl` is for reading and writing files text files (namely CSVs) and offers top-class read and write performance. `DataFrames.jl` is a mature package for working with dataframes, comparable to Pandas or dplyr.

Other notable packages include `ODBC.jl`, which lets you connect to any database (given you have the right drivers installed), and `Arrow.jl` which implements the Apache Arrow standard in Julia.

Worth mentioning also is `Dates`, a built-in package making date manipulation straightforward and robust.

Check out [JuliaData.org](https://juliadata.org) for more packages and information.

29.0.2. Plotting

`Plots.jl` is a meta-package providing an interface to consistently work with several plotting backends, depending if you are trying to emphasize interactivity on the web or print-quality output. You can very easily add animations or change almost any feature of a plot.

`StatsPlots.jl` extends `Plots.jl` with a focus on data visualization and compatibility with dataframes.

`Makie.jl` supports GPU-accelerated plotting and can create very rich, beautiful visualizations, but its main downside is that it has not yet been optimized to minimize the time-to-first-plot.

29.0.3. Statistics

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.

`StatsBase.jl` and `Distributions.jl` are essentials for a range of statistics functions and probability distributions respectively.

Others include:

- `Turing.jl`, a probabilistic programming (Bayesian statistics) library, which is outstanding in its combination of clear model syntax with performance.
- `GLM.jl` for any type of linear modeling (mimicking R's `glm` functionality).
- `LsqFit.jl` for fitting data to non-linear models.
- `MultivariateStats.jl` for multivariate statistics, such as PCA.

You can find more packages and learn about them [here](#).

29.0.4. Machine Learning

`Flux`, `Gen`, `Knet`, and `MLJ` are all very popular machine learning libraries. There are also packages for PyTorch, Tensorflow, and SciKitML available. One advantage for users is that the Julia packages are written in Julia, so it can be easier to adapt or see what's going on in the entire stack. In contrast to this design, PyTorch and Tensorflow are built primarily with C++.

Another advantage is that the Julia libraries can use automatic differentiation to optimize on a wider range of data and functions than those built into libraries in other languages.

29.0.5. Differentiable Programming

Sensitivity testing is very common in actuarial workflows: essentially, it's 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. For example, the following is real Julia code to automatically calculate the sensitivity of the ending account value with respect to the inputs:

29. The Julia Ecosystem Today

```
julia> using Zygote

julia> function policy_av(pol)
    COIs = [0.00319, 0.00345, 0.0038, 0.00419, 0.0047, 0.00532]
    av = 0.0
    for (i,coi) in enumerate(COIs)
        av += av * pol.credit_rate
        av += pol.annual_premium
        av -= pol.face * coi
    end
    return av           # return the final account value
end

julia> pol = (annual_premium = 1000, face = 100_000, credit_rate =
4048.08

julia> policy_av(pol)      # the ending account value
4048.08

julia> policy_av'(pol)     # the derivative of the account value
(annual_premium = 6.802, face = -0.0275, credit_rate = 10972.52)
```

When executing the code above, Julia isn't just adding a small amount and calculating the finite difference. Differentiation is applied to entire programs through extensive use of basic derivatives and the chain rule. **Automatic differentiation**, has uses in optimization, machine learning, sensitivity testing, and risk analysis. You can read more about Julia's autodiff ecosystem [here](#).

29.0.6. Utilities

There are also a lot of quality-of-life packages, like `Revise.jl` which lets you edit code on the fly without needing to re-run entire scripts.

`BenchmarkTools.jl` makes it incredibly easy to benchmark your code - simply add `@benchmark` in front of what you want to test, and you will be presented with detailed statistics. For example:

```
julia> using ActuaryUtilities, BenchmarkTools

julia> @benchmark present_value(0.05,[10,10,10])

BenchmarkTools.Trial: 10000 samples with 994 evaluations.
Range (min ... max): 33.492 ns ... 829.015 ns | GC (min ... max): 0.00% ... 95.40%
Time (median): 34.708 ns | GC (median): 0.00%
Time (mean ± σ): 36.599 ns ± 33.686 ns | GC (mean ± σ): 4.40% ± 4.55%

Memory estimate: 112 bytes, allocs estimate: 1.
```

Test is a built-in package for performing testsets, while Documenter.jl will build high-quality documentation based on your inline documentation.

ClipData.jl lets you copy and paste from spreadsheets to Julia sessions.

29.0.7. Other packages

Julia is a general-purpose language, so you will find packages for web development, graphics, game development, audio production, and much more. You can explore packages (and their dependencies) at <https://juliahub.com/>.

29.0.8. Actuarial packages

Saving the best for last, the next article in the series will dive deeper into actuarial packages, such as those published by JuliaActuary for easy mortality table manipulation, common actuarial functions, financial math, and experience analysis.

30. Debugging and Performance Measurement

30.1. Benchmarking

References

- 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.

