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
Prerequisites 9
The Contents of This Book 10
Notes on formatting 11
Colophon 11

2. Why Program? 13
2.1. In this Chapter 13
2.2. The Long View 13
2.3. What's coding got to do with this? 15
2.4. The 10x Actuary 16
2.5. Risk Governance 17
2.6. Managing and Leading the Transformation . . . 18
2.7. Outlook 18

3. Why use Julia? 21
3.1. Expressiveness and Syntax 22
3.1.1. Example: Retention Analysis 22
3.1.2. Example: Random Sampling 24
3.2. The Speed 25
3.3. More of Julia's benefits 26
3.4. The Tradeoff 27
3.5. Package Ecosystem 28
3.6. Conclusion 29

II. Conceptual Foundations **31**

4. Elements of Financial Modeling **33**
4.1. In this Chapter . . . 33
4.2. What is a model? . . . 33
4.3. What is a *Financial* Model? . . . 35
4.4. The Process of Building a Financial Model . . . 36
4.5. Predictive versus Explanatory Models . . . 37
4.5.1. A Historical Example . . . 37
4.5.2. Examples in the Financial Context . . . 40
4.6. What makes a good model? . . . 41
4.6.1. Achieving original purpose . . . 41
4.6.2. Usability . . . 41
4.6.3. Performance . . . 42
4.6.4. Separation of Model Logic and Data . . . 43
4.7. What makes a good modeler? . . . 43
4.7.1. Domain Expertise . . . 43
4.7.2. Model Theory . . . 44
4.7.3. Curiosity . . . 46
4.7.4. Rigor . . . 47
4.7.5. Clarity . . . 47
4.7.6. Humble . . . 47
4.7.7. Architecture . . . 47
4.7.8. Planning . . . 47
4.7.9. Toolset . . . 48

5. Elements of Programming **49**
5.1. In this section . . . 49
5.2. Computer Science, Programming, and Coding . . . 49
5.3. Assignment and Variables . . . 51
5.4. Data Types . . . 52
5.4.1. Numbers . . . 52
5.4.2. Type Hierarchy . . . 56
5.4.3. Arrays . . . 56
5.4.4. Characters, Strings, and Symbols . . . 61
5.4.5. Tuples . . . 63
5.4.6. Parametric Types . . . 64
5.4.7. Types for things not there . . . 65
5.4.8. Union Types . . . 65
5.4.9. Creating User Defined Types . . . 67
5.4.10. Mutable structs . . . 70

5.4.11. Constructors 71
5.5. Expressions and Control Flow 72
5.5.1. Compound Expression 73
5.5.2. Conditional Expressions 74
5.5.3. Assignment and Variables 76
5.5.4. Loops 77
5.5.5. Performance of loops 79
5.6. Functions 79
5.6.1. Special Operators 79
5.6.2. General Functions 80
5.6.3. Keyword Arguments 82
5.6.4. Default Arguments 83
5.6.5. Anonymous Functions 83
5.6.6. Passing by Sharing 84
5.6.7. Broadcasting 85
5.6.8. First Class Nature 89
5.7. Scope 90
5.7.1. Modules and Namespaces 91

6. Patterns of Abstraction **93**
6.1. In this section 93
6.2. Introduction 93
6.3. Interfaces 94
6.3.1. Conceptual Strategies 95
6.4. Programming Interfaces and Patterns 96
6.4.1. (Multiple) Dispatch 99
6.4.2. Programming Paradigms 101
6.4.3. Composition over Inheritance 102
6.4.4. Method Dispatch 104
6.5. Macros & Homoiconicity 104
6.6. Misc Techniques 104
6.6.1. Recursion 104
6.6.2. Iterators 104

7. Elements of Computer Science **105**
7.1. In this section 105
7.2. Computer Science for Financial Professionals . . 105
7.3. Algorithms 106
7.3.1. Computational Complexity 107
7.3.2. Expected versus worst-case complexity . 112
7.3.3. Complexity: Takeaways 112

7.4. Data Structures 113
7.5. Formal Verification 113
7.6. The Discipline of Software Engineering 113
7.6.1. Patterns 113

8. Hardware and It's Implications 115
8.1. In this section 115

9. Applying Software Engineering Principles 117
9.1. In this section 117

10. Statistical Inference and Information Theory 119
10.1. In This Chapter 119
10.2. Information Theory 119
10.2.1. Example: Classificaiton 121
10.2.2. Maxium Entropy Distributions 125
10.3. Bayes' Rule 130
10.3.1. Example: Model Selection via Likelihoods 131
10.4. Modern Bayesian Statistics 138
10.4.1. Advantages of the Bayesian Approach . 138
10.4.2. Challenges with the Bayesian Approach 139
10.4.3. Why Now? 140
10.4.4. Subjectivity of the Priors? 141
10.4.5. Implications for Risk Management . . . 143
10.4.6. Paving the Way Forward for Actuaries . 144

III. Computational Thinking in an Actuarial and Financial Context 145

11. Modeling 147
11.1. In This Chapter 147
11.2. Parsimony 147

12. Automatic Differentiation 149
12.1. In This Chapter 149
12.2. Motivation for (Automatic) Derivatives 149
12.3. Finite Differentiation 149
12.4. Automatic Differentiation 153
12.4.1. Dual Numbers 154
12.5. Performance of Automatic Differentiation 157

12.6. Automatic Differentiation in Practice 159
12.6.1. Performance 161
12.7. Forward Mode and Reverse Mode 162
12.8. Practical tips for Automatic Differentiation . . . 163
12.8.1. Choosing between Reverse Mode and Forward Mode 163
12.8.2. Mutation 163
12.8.3. Custom Rules 163
12.8.4. Available Libraries 164
12.9. References 164

13. Optimization **165**
13.1. In This Chapter 165

14. Sensitivity Analysis **167**
14.1. In This Chapter 167
14.2. Setup 167
14.3. The Data 168
14.4. Common Sensitivity Analysis Methodologies . . 170
14.4.1. Finite Differences 170
14.4.2. Scenario Analyses 170
14.4.3. Regression Analyses 171
14.4.4. Sobol Indices 171
14.4.5. Morris Method 172
14.4.6. Fourier Amplitude Sensitivity Tests . . . 172
14.5. Benchmarking 172

15. Stochastic Modeling **173**

16. Visiualizations **175**
16.1. In This Chapter 175

17. Matrices and Their Uses **177**
17.1. In This Chapter 177

18. Learning from Data **179**
18.1. In this chapter 179

IV. Applications in Practice **181**

19. Stochastic Mortality Projections **183**
19.1. In This Chapter . . . 183
19.2. Setup . . . 183
19.3. The Data . . . 184
19.4. Running the projection . . . 187
19.4.1. Stochastic Projection . . . 187
19.5. Benchmarking . . . 189
19.6. Further Optimization . . . 190

20. Scenario Generation **191**
20.1. In This Chapter . . . 191
20.2. Setup . . . 191
20.3. The Data . . . 191
20.4. Pseudo Random Number Generators . . . 191
20.4.1. Common PRNGs . . . 192
20.4.2. Consistent Interface . . . 193
20.5. Common Economic Scenario Generation Approaches . . . 194
20.5.1. Interest Rate Models . . . 194
20.5.2. Stock Models . . . 199
20.5.3. Copulas . . . 203
20.6. Benchmarking . . . 204

21. Similarity Analysis **205**
21.1. In This Chapter . . . 205
21.2. Setup . . . 205
21.3. The Data . . . 205
21.4. Common Similarity Measures . . . 207
21.4.1. Euclidean Distance (L2 norm) . . . 207
21.4.2. Manhattan Distance (L1 Norm) . . . 208
21.4.3. Cosine Similarity . . . 208
21.4.4. Jaccard Similarity . . . 209
21.4.5. Hamming Distance . . . 209
21.5. k-Nearest Neighbor (kNN) Clustering . . . 210
21.6. Benchmarking . . . 211

22. Portfolio Optimization **213**
22.1. In This Chapter . . . 213

23. Bayesian Mortality Modeling **215**

24. Other Useful Techniques **227**
24.1. In this chapter 227
24.2. Taking things to the Extreme 227
24.3. Range Bounding 227

V. Appendices **229**

25. Set up Julia and the Computing Environment **231**
25.1. Installation 231
25.2. Package Management 231
25.3. Editors 232
25.3.1. Visual Studio Code 232
25.3.2. Notebooks 233

26. Environment and Package Management **235**
26.1. In This Section 235
26.2. Projects, Manifests, and Dependencies 235
26.2.1. Project.toml 235
26.2.2. Manifest.toml 238
26.2.3. Reproducibility 238
26.3. Environemnts 239
26.4. Packages 240
26.4.1. Packages versus Projects 240
26.4.2. Basic Package Structure 240
26.4.3. Extension Packages 240
26.5. Regisries 240
26.5.1. Local Registries 240

27. The Julia Ecosystem Today **241**
27.0.1. Data 242
27.0.2. Plotting 242
27.0.3. Statistics 242
27.0.4. Machine Learning 243
27.0.5. Differentiable Programming 243
27.0.6. Utilities 244
27.0.7. Other packages 245
27.0.8. Actuarial packages 245

References **247**

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

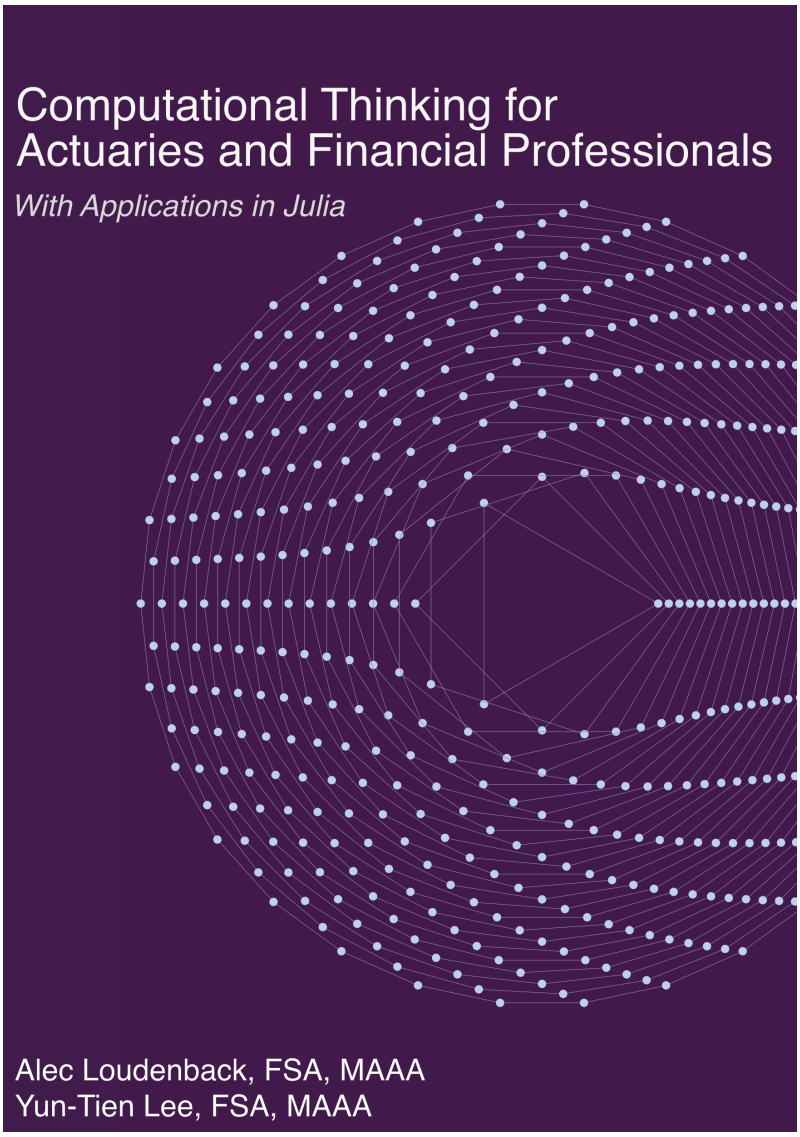


Figure 1.1.: Draft of Cover

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 practicitioner 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 finanical 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 specicific 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,

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 professsionals 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 usuable by others.

Prerequisites

Basic experience with financial modeling is not strictly required, but it will benefit the reader to be familar 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 25 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 □ 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.

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 25) 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 26.1), 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 indiate 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 cormullion on Github.

This book was rendered on March 16, 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)
Threads: 1 default, 0 interactive, 1 GC (on 4 virtual cores)
```

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 machine[1].

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 software[2].

[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

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 sharing[3]; Airlines are so dependent on their systems that the skies become quieter on the rare occasion that their computers give way.

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

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.**

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

- 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].

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

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

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

- 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.

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 new[5], 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.

[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.

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 modeling[6]. 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**.

[6] Aviva Case Study

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.

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`, `Policys`, 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
```

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
```

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

`using Distributions`
 	
`rand(100)`
`rand(Normal(), 100)`
`rand(Bernoulli(0.5), 100)`
`rand(["Preferred","Standard"], 100)`
	`runif(100)`
	`rnorm(100)`
	`rbern(100, 0.5)`
	`sample(c("Preferred","Standard"),`
	`100, replace=TRUE)`
		`import scipy.stats as sps`
		`import numpy as np`
		
		`sps.uniform.rvs(size=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

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!

3.3. 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! **Ju**lia-**Pyt**hon-**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.

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.

3.5. Package Ecosystem

Using packages as dependencies in your project is assisted by Julia' 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

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.

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 inpsected, 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 withing the "large world" that exists around us.

More on the "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 cashlows and the present value of those cashflows has increased by 5% I may ask why the result 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 "big 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* relfected 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 the 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)

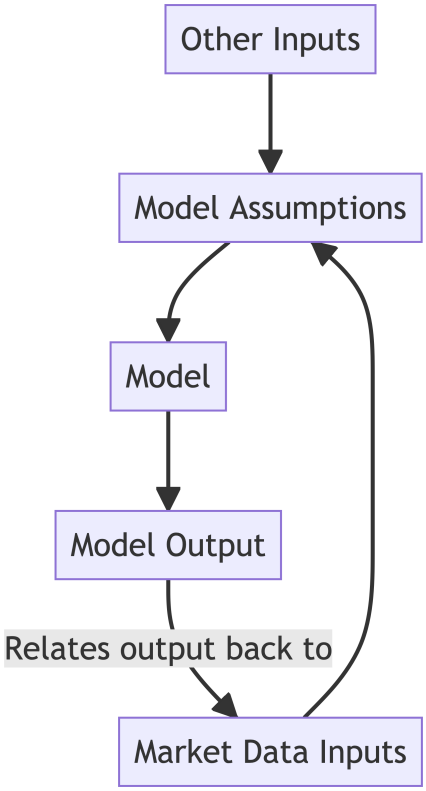
Models are sometimes taken a step further, such as transforming the underlying **economic view** into an accounting or regulatory view (such as representing associated debits and credits,

capital requirements, or associated intangible, capitalized balances).

We should also distinguish a financial model from a purely statistical model, where the 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

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 around[7].

[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

The existing Ptolemic 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 see 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 obersvational data that described the position of the planets in the sky.

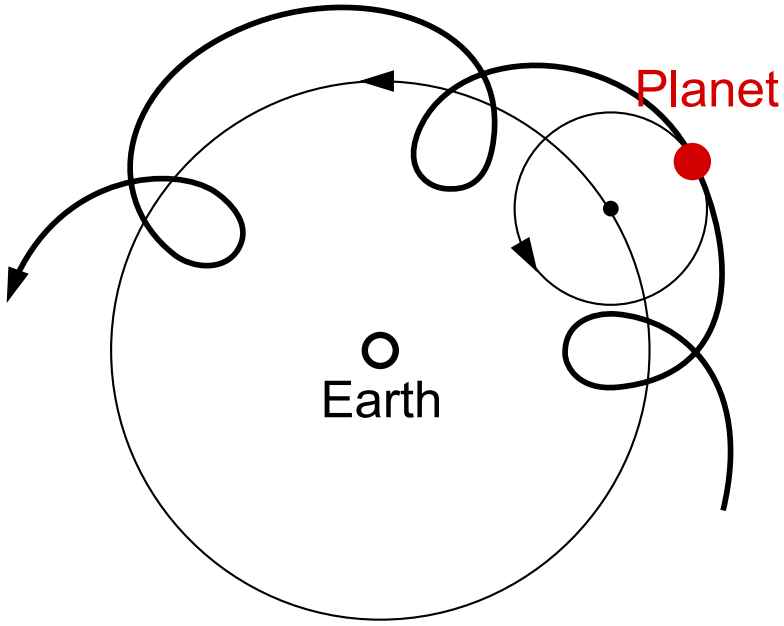


Figure 4.1.: In the Ptolemic solar model, the retrograde motion of the planets was explained by adding an epicycle to the circular orbit around the earth.

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

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:

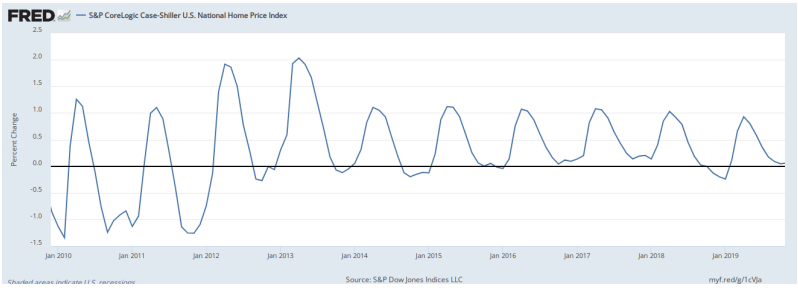
1. Predictive models need not have a scientific, causal structure to make accurate predictions.
2. It is difficult to capture the complete scientific interrelationships of a system and much care and thought needs to be given in what aspects are included in our model.
3. We should look at, or seek out, 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.

4.5.2. 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 constructed[8]. 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)

4.6. 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 inteded 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 acheive the same level of capability.

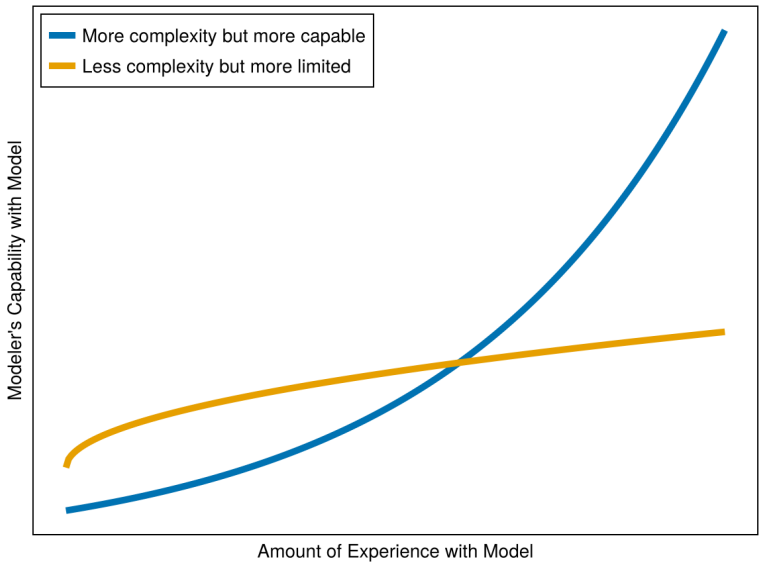


Figure 4.2.: Tradeoff between complexity and capability

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.

4.6.4. 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.

4.7. What makes a good modeler?

A model is nothing without it's 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 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's[9] "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 it's results[10].

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 it's 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
- 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.

[9] Ryle, G. The Concept of Mind. Harmondsworth, England, Penguin, 1963, first published 1949. Applying "Theory Building"

[10] The idea of "model theory" is adapted from Peter Naur's 1985 essay, "Programming as Theory Building". Indeed, this whole paragraph is only a slightly modified version of Naur's description of theory in the programming context.

- 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:

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* `Policy`*s 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. The latter 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 then answers provided. It is our experience that you modeler who 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 establish 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 speakers 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

4.7.8. 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.

5.2. Computer Science, Programming, and Coding

Computer Science is the study of computing and information. As a science, it is distinct from programming languages which are merely coarse implementations of specific computer science concepts[11]. 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.

[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 ($3^2 + 4^2 = 5^2$).

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 those to discuss algorithms (functions) which perform useful work and use control flow and recursion.
- The following chapter 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 8.

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 25 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...
```

> **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 defintions, and a more complete tour of all language features, see the Manual at docs.julialang.org.

5.3. 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
```

```
0.5681818181818182
```

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 return to this subject in more detail in (**ref-assignment?**).

5.4. 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 `66`[12].

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 handle[13].

5.4.1. Numbers

Numbers are usually grouped into two categories: **integers** and **floating-point**[14] 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 8). Here are three integers that are input into the **REPL** (Read-Eval-Print-Loop)[15] and the result is **printed** below the input:

```
2
```

[12] This binary representations correspond to `B` and `66` with the *ASCII character set* and 8-bit integer encodings respectely, discussed later in this chapter.

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

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

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

```
2
```

```
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-bit[16], 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:

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

- `0.200000002980232238769531250` in 32-bit
- `0.200000000000000011102230246251565404236316680908203125` in 64-bit

This leads to special considerations that computers take when performing calculations on floating point maths, some of which will be covered in more detail in Chapter 8. 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 5.5.3.
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`.

```
0.2000000000000000011102230246251565404236316680908203125
```

> **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 usally 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 ouput.
> 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`

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

Financial Modeling Pro-tip

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

In practice, this means that it can be difficult to exactly replicate a calculation in Excel in a progamming 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.

5.4.2. Type Hierarchy

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

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

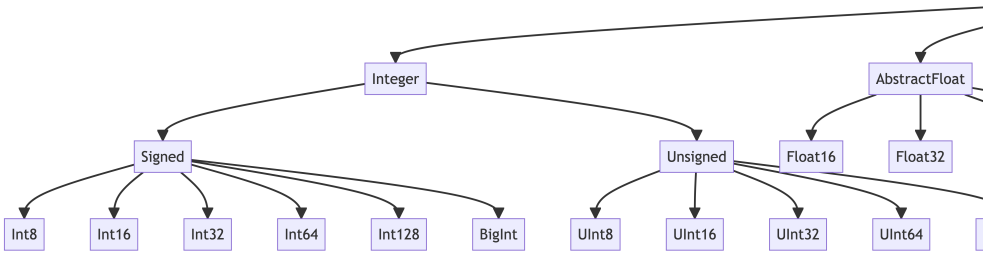


Figure 5.1.: Numeric Type Hierarchy in Julia. Leafs of the tree are concrete types.

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.

5.4.3. Arrays

□ Julia has very powerful and friendly array types.

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 represented[17] in floating point.

[17] Accurate only to a limited precision, as described in Section 5.4.1.

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

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

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 8.

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

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

```
3×3×3 Array{Float64, 3}:
[:, :, 1] =
 0.306579  0.433822  0.0279186
 0.11261   0.819463  0.57017
 0.20586   0.197198  0.914576

[:, :, 2] =
 0.504231  0.600536  0.711044
 0.805836  0.605322  0.367649
 0.843316  0.252639  0.769685

[:, :, 3] =
 0.818567   0.397939  0.282274
 0.0521626  0.50317   0.9586
 0.90322    0.386608  0.641944
```

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

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

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

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

The above example demonstrates **array comprehension** syntax which is a convienient 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
```

> **Note**
>
> In Julia, a `Vector{Float64}` is simply a one-dimensional array of floating pointsand a `Matrix{Float64}` is a two-dimentional array. More precisely, they are **type aliases** of the more generic `Array{Float64,1}` and `Array{Float64,2}` names.

5.4.3.1. Array indexing

Array elements are accessed with the integer position, starting at `1` for the first element[18] [19]:

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

```
20
```

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

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

```
3-element Vector{Int64}:
 20
 30
 40
```

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

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

[18] 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.

[19] Arrays in Julia can actually be indexed with an arbitrary starting point: see the package OffsetArrays.jl

```
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
```

5.4.3.2. 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 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 (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]
```

```
3×3 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
```

5.4.4. Characters, Strings, and Symbols

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

```
'a'
```

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

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

Strings are a collection[20] 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` 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.

[20] Under the
tially a vecto
are complexi
ing that don'
sion to indiv
form bit len
cal compatib
making mos
larger than it

5.4.5. 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 that one:

- Tuples have no problem having heterogeneous types in the different slots.
- Tuples are **immutable**, meaning that you cannot overwrite the value in memory (an error will be thrown if we try to do `p[1] = 5`).
- It's generally expected that within an array, you would be able to apply the same operation to all the elements (e.g. square each element) or do something like sum all of the elements together which isn't generally case for a tuple.
- Tuples are generally stack allocated instead of being heap allocated like arrays[21], meaning that a lot of times they can be faster than arrays.

[21] What this means will be explained in Chapter 8 .

5.4.5.1. Named Tuples

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

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

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

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

```
p2.x
```

```
3
```

5.4.6. 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}
```

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.

5.4.7. 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 an 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 here. This is analagous to `Float64` being a type and `2.0` being a value.

5.4.8. Union Types

When two types may arise in a context, **union types** are a way to represent that. For example, if we have a data feed and we know that it will produce *either* a `Float64` *or* a `Missing` type then we can say that the value for this is

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

(a) Not logic

NOT (`!`)	Value
`true`	`false`
`missing`	`missing`
`false`	`true`

(b) And logic

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

(c) Or Logic

OR (`|`)	`true`	`missing`	`false`
`true`	`true`	`true`	`true`
`missing`	`true`	`missing`	`missing`
`false`	`true`	`missing`	`false`

`Union{Float64,Missing}`. This is much better for the compiler (and our performance!) than saying that the type of this is `Any`.

5.4.9. 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
```

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
```

> **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 any 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 sophicistited 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 hiearchy of types which we will later exploit (Chapter 6) to implement custom behavior depending on what type our data is.

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

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

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

struct Point3D{T} <: CartesianCoordinate
    x::T
```

```
    y::T
    z::T
end

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

> **Unicode Characters**
>
> Julia has wonderful Unicode support, meaning that it's not a problem to include characters like `θ`. The character can be typed in Julia editors by entering `\theta` and then pressing the TAB key on the keyboard.
> Unicode is helpful for following conventions that you may be used to in math. For example, the math formula 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>
> ```

5.4.10. Mutable structs

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

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

You may find that this more naturally represents what you are trying to do. However, recall that an advantage of an immutable datatype is that costly memory doesn't necessarily have to be allocated for it. So you may think that you're being more efficient by re-using the same object... but it may not actually be faster. Again, more will be revealed in Chapter 8.

> 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.

5.4.11. 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[38]:2
 [2] MyDate(year, month, day)
     @ In[38]:2
```

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

there's reasons to want to control how an object is created, either for convenience or to enforce certain restrictions.

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

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

    function MyDate(y,m,d)
        if ~(m in 1:12)
            error("month is not between 1 and 12")
        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
```

5.5. Expressions and Control Flow

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
```

5.5.1. Compound Expression

There's two kinds of blocks where we can ensure that subexpressions 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.

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

```
3
```

```
d
```

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

5.5.2. 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 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
```

> **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 egal to `1` (egal is defined later in this chapter).

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")
```

5.5.2.1. Equality

The "Ship of Theseus[22]" 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
- **Equal** equality is when the values of two objects are the same

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

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

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

```
(true, false)
```

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

[22] 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.

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

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

Now `t` is no longer equal to `s`:

```
s == t
```

```
false
```

Recall that arrays are able to be modified, but other types like tuples are immutable. Immutable types with the same value are egal because there is no way for us to make them different:

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

```
true
```

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

5.5.3. 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
```

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

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

5.5.4. 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

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

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

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

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

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

① `global` is used to incrment `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                                        ①
    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.

5.5.5. 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!*

5.6. 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
- `-(x)` is a unary function which negates its values, `-(x,y)` subtracts `y` from `x`
- Specific methods are then created for each combination of concrete types: `-(x::Float64)` is a different method than `-(x::Int)`

Methods are specific compiled versions of the function for specific types. This is important because at a hardware level, operations for different types (e.g. integers versus floating point) differ considerably. By optimizing for the speicifc 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 6.

5.6.2. General Functions

Funcitons more generally are defined like so:

```
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 argumemnt called `point`
② We compute the distance formla 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)
```

> **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 defintion if we think about the various `Coordinates` we defined earler: 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. We'll also use the opportunity to introduce the syntax for documenting functions in Julia, which is simply to put a string (`"..."`) or string literal (`"""..."""`) right above the defintion.

```
"""
    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
```

In the next chapter we'll develop some more tools, which would, for example let us define the function `dist(p::CartesianCoordinate)` and generically define the distance for all of `CartesianCoordinate`'s subtypes.

> **Defining Methods for Parametric Types**
>
> We learned that `Float64 <: Real` in the type hierarchy. However, note that `Tuple{Float64}` is not a subtype of `Tuple{Real}`. This is called being **invariant** in type theory... but for our purposes this just practically means that when we define a method we need to specify that we want it to apply to all subtypes.
> For example, `myfunction(x::Tuple{Real})` would *not* be called if x was a `Tuple{Float64}` because it's not a subtype of `Tuple{Real}`. To act the way we want, would define the method with the signature of `myfunction(Tuple{<:Real})` or `myfunction{Tuple{T}} where {T<:Real}`.

5.6.3. 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 semicolon (`;`) 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"; normalizenames=true, hasheaderrow=false)
```

5.6.4. Default Arguments

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

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

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

5.6.5. 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.

5.6.6. Passing by Sharing

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 it's arguments has a `!` in it's name and we follow this conven-

> tion in `double!` above. Another example would be the built-in function `sort!` which will sort an array in-place without allocating a new array to store the sorted values.

We won't discuss all potential ways that programming languages can behave in this regard, but an alternative that one may have seen before (e.g. in Matlab) is pass-by-value where a modification to an argument only modifies the value within the scope. Here's how to replicate that in Julia by `copy`ing 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
```

5.6.7. Broadcasting

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

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

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

5.6.7.1. 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]
```

```
2×3 Matrix{Int64}:
 2  4  6
 5  7  9
```

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

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

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

This 2x4 isn't compatible with the 2x3 (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; got a dimension with len
```

5.6.7.2. 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])
```

```
2×3 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`:

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

```
2-element BitVector:
 1
 0
```

5.6.8. First Class Nature

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

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

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

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

# obs meaning "observations"
function loss(modeled_obs,
    actual_obs,
    loss_function                                    ①
)
    sum(
        loss_function.(modeled_obs, actual_obs)
    )
end

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

5.7. 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                                                    ①
let                                                      ②
    j = 3                                                ③
    i + j
end
```

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
```

Here is an example with functions:

```
x = 2
base = 10
foo() = base^x                                    ①
foo(x) = base^x                                   ②
foo(x, base) = base^x                             ③

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:

```
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)                    ③
area = Shape.area(t)                        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`
>   - **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`).

6. Patterns of Abstraction

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

6.1. In this section

We extend the building blocks from the prior section and talk about how to combine them into more abstract patterns to simplify the design of our programs. Data driven design, object oriented design versus composition, multiple dispatch, and interfaces.

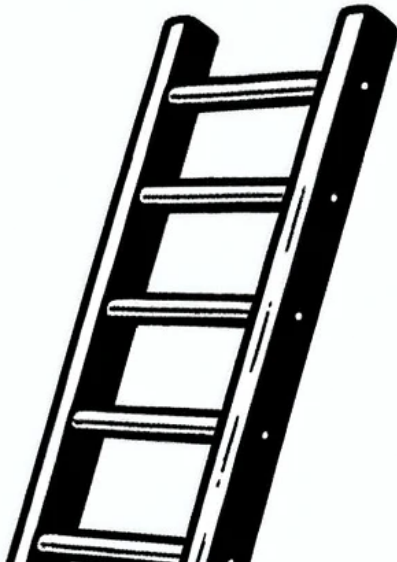
6.2. Introduction

Abstraction is a selective ignorance—focusing on the aspects of the problem that are relevant, and ignoring the others. 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 6.1.: An example of the different levels of abstraction when thinking about modeling an insurance company's assets and liabilites.

	Item
More Abstract	Sensitivity of an entire company's solvency position



	Item
	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 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 6.1, But it may make sense in our model to separate the data and behavior from each other. If we were to do that, that would be an example of creating horizontal abstraction in service of our overall modeling goals.

6.3. Interfaces

Interfaces are the boundary between different encapsulated abstractions. 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, 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.

6.3.1. Conceptual Strategies

Let's consider some stategies that you could think about when deciding where to draw different boundaries inside the model.

6.3.1.1. 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.

6.3.1.2. 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.

6.3.1.3. 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.

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.

6.4. Programming Interfaces and Patterns

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.
- 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

We will also discuss here some **patterns** which are ways of doing things that seem to appear repeatedly and specific design choices have proven to work well in the past and should be considered when similar conditions arise in the future.

Let's develop a simplified system to value simple fixed income assets in order to illustrate some patterns. Inside a module called `Asset`, we'll define a short hiearchy of types and then a function `value` with multiple methods for the relevant types.

```
module Asset

## 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


## Functions

"""
    value(asset,discount_rate)
```

```
The value of an asset with the given discount rate for it's cashfl
"""
value(asset::Cash, r) = asset.balance

function value(asset::AbstractBond, r)                    ④
    discount_factor = 1.0
    value = 0.0
    for t in 1:asset.tenor
        discount_factor /= (1 + r)                        ③
        value += discount_factor * cashflow(asset, t)
    end
    return value
end

function cashflow(bond::CouponBond, time)
    if time == bond.tenor
        (1 + bond.coupon) * bond.par
    else
        bond.coupon * bond.par
    end
end

function value(bond::ZeroCouponBond, r)                   ⑤
    return bond.par / (1 + r)^bond.tenor
end

end
```

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

② We define two simple bonds: a coupon paying and a zero-coupon instrument.

③ `x /= y`, `x += y`, etc. are shorthand ways to write `x = x / y` or `x = x + y`

④ `value` is defined for `AbstractBonds` in general...

⑤ ... and then more specifically for `ZeroCouponBonds`. This will be explained when discussing "dispatch" below.

Here's an example of how this would be used:

```
portfolio = [
    Asset.Cash(50.0),
```

```
    Asset.CouponBond(100.0, 0.05, 5),
    Asset.ZeroCouponBond(100.0, 5),
]

Asset.value.(portfolio, 0.05)
```

```
3-element Vector{Float64}:
 50.0
 99.99999999999999
 78.35261664684589
```

> **Note**
>
> In the example above, a docstring was included over `value(asset::Cash)` - but not over the others. That's okay. Julia will show docstrings for the *function* `value` not just individual *methods*.

There are quite a few things demonstrated here: dispatch, programming paradigms, and [!!what else?] . As each are addressed in turn, we will review how we could have designed the interface differently.

6.4.1. (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 used is the more specific `value(bond::ZeroCouponBond, r)`.

Already, this is a powerful tool to simplify our code. Imagine the alternative of a long chain of conditional statements trying to find the right logic to use:

```
# don't do this!
function value(asset,r)
    if asset.type == "ZeroCouponBond"
        # special code for Zero coupon bonds
        # ...
    elseif asset.type == "ParBond"
        # special code for Par bonds
        # ...
    elseif asset.type == "AmortizingBond"
        # special code for Amortizing Bonds
        # ...
    else
        # here define the generic AbstractBond logic
    end
end
```

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? All we have to do is define a new method where the first argument is our `AbstractBond`s as already written above, but the second argument is our new type, which might look like this:

```
struct MyCurve{F} where {F<:Function}
    discount_rate::F
end

function value(bond::ZeroCouponBond, c::MyCurve)    (5)
    return bond.par * c.discount_rate(bond.tenor)
end
```

In this way, multiple dispatch allows us to naturally define methods based on the combination of types.

what if we wanted to extend from a rate to a yield curve and fixed to floating. Where does logic lie? In yield curve or in bond?)

6.4.2. Programming Paradigms

6.4.2.1. Objected 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 the first computer programming abstractions introduced because it very relatable[23], 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.

[23] "Many people who have no idea how a computer works find the idea of object-oriented programming quite natural. In contrast, many people who have experience with computers initially think there is something strange about object oriented systems." - David Robson, "Object Oriented Software Systems" in Byte Magazine (1981).

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.

6.4.2.2. 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.

6.4.3. 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


# 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 secu-

> rities and to uniquely identify them in transactions and records.

6.4.4. Method Dispatch

An alternative and more limiting approach would be to be forced to assign the ownership of the method to one of the associated types, as is done in single-dispatch.

- Alternative designs:
  - ZeroCouponBond(par,tenor) = CouponBond(par,0.0,tenor)

6.5. Macros & Homoiconicity

6.6. Misc Techniques

6.6.1. Recursion

6.6.2. Iterators

7. 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)

7.1. In this section

Adapting computer science concepts to work for financial professionals. Concepts like computability, computational complexity, the language of algorithms and problem solving, looking for and using patterns, and adopting digital-first practices to automate the boring parts of the job.

7.2. 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 it's core, computer science concerns itself with areas of reserach 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 utlity 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).

7.3. Algorithms

Algorithms is a general term for a process that transforms an input to an output. It's the dirty, down-to-earn implementation of a mathemetical 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.

7.3.1. 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 exmaple, division instructions may require many CPU cycles when compared to multiplication or division. Sometimes this is an important distiction and sometimes not, but for this book we will ignore this level of analysis.

7.3.1.1. 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

and the associated computational complexity for them.

7.3.1.2. 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 mulitiplication, 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.

7.3.1.3. 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 alogrithm will take approximately n steps to complete.

7.3.1.4. 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                                    ①
        for j in 1:i                                ②
            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 $1 + ... + (n-2) + (n-1) + n$ 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.

7.3.1.5. Comparison

7.3.1.5.1. Big-O Notation

We can categorize the above implementations using a convention called **Big-O Notation**[24] 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 7.1 shows for the examples constructed above what the description, order, and order of magnitude complexity is.

[24] "Big-O", so named because of the "O" in used in $O(1)$. $O(n)$, etc. Not one of the sciences' more creative names.

Table 7.1.: Complexity comparison for the three sample cases of summing integers from 1 to n.

Function	Computational Cost	Complexity Description	Big-O Order	Steps ($n = 10,000$)
`nsum_constant`	fixed	Constant	$O(1)$	~1
`nsum_linear`	n	Linear	$O(n)$	~10,000
`nsum_quadratic`	$n^2 + n \div 2$	Quadratic	$O(n^2)$	~100,000,000

Table 7.2 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 nth Fibbonnaci 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).

Table 7.2.: Different Big-O Orders of Complexity

Big-O Order	$n = 10$	$n = 1,000$	$n = 1,000,000$
$O(1)$	1	1	1
$O(n)$	10	1,000	1,000,000
$O(n^2)$	100	1,000,000	10^12
$O(log(n))$	3	7	14
$O(n \times log(n))$	30	7,000	14,000,000
$O(2^n)$	1,024	~10^300	~10^301029
$O(n!)$	3,628,800	~10^2567	~10^5565708

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.

7.3.1.5.2. Empirical Results

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

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

```
50005000
```

```
@btime nsum_linear(10_000)
```

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

```
50005000
```

```
@btime nsum_quadratic(10_000)
```

```
  2.393 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.

7.3.2. 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.

7.3.3. 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 thoughful 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).

7.4. Data Structures

Data structures is the art and science of how to represent data in discreate objects. There are

7.5. Formal Verification

7.6. The Discipline of Software Engineering

7.6.1. Patterns

8. Hardware and It's Implications

8.1. 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.

9. 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)

9.1. In this section

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

10. Statistical Inference and Information Theory

"My greatest concern was what to call [the amount of unpredictabilty 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)

10.1. In This Chapter

A brief introduction to infromation 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.

10.2. 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.

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 _ indicating that the digit is not visible:

```
32,000,_00
```

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 10.1. We shall call the individual outcomes x_i and the overall set of probabilities $\{x_0, x_1, ...x_9\}$ is called X.

The information content of an outcome, $h(x)$ is measured in bits and defined as[25]:

$$h(x_i) = \log_2 \frac{1}{p(x_i)} \tag{10.1}$$

[25] Log base two turns out to be the most natural representation of information content as it mimics the fundamental 0 or 1 value bit. A more complete introduction is available in "Information Theory, Inference, and Learning Algorithms" by David MacKay.

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:

$$H(X) = \sum p(x_i) \log_2 \frac{1}{p(x_i)} \tag{10.2}$$

The entropy $H(X)$ of the presumed outcomes in Table 10.1 distirubtion of outcomes is 0.722bits.

Table 10.1.: Probability distribution of missing digit knowing human inclinations 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 beleived humans are completely rational and without bias for certaim numbers. They would then be arguing for a prior assumed distribution consistent with Table 10.2.

With the uniform prior assumption, $H(X) = 3.322$bits and $h(x_i)$ is also unifrom. 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 10.2.: 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

10.2.1. Example: Classificaiton

In this example, we will determine the optimal splits for a decision tree[26] based on the information gained at each node in the tree.

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

```
using DataFrames
```

```
employed = [true, false, true, true, true, false, false, true]
good_credit = [true, true, false, true, false, false, false, true]
```

Table 10.3.: Fictional data regarding loan attributes and whether or not a loan defaulted before it's 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

```
default = [true, false, true, true, true, true, false, true]
default_data = DataFrame(; employed, good_credit, default)
```

The entropy of the default rate data is, per Equation 10.2:

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

	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                                    ①
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 $\lim_{p \to 0^+} p \log(p) = 0$.

```
0.0
```

And the corresponding candidate leaf is `H_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
```

```
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 `H0` (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 10.3 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.

10.2.2. Maxium 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 Duracz[27]. probability distributions and risk asses

[27] https://www.researchgate.net/publication/239752412_Derivation_of_Probability_Distributions_for_Risk_Assessment

Table 10.4.: 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)

Constraint	Discrete Distribution	Continuous Distribution
Positive and equal mean and variance	Poisson	

The distributions in Table 10.4 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.

10.2.2.1. 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 its 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 Guassian.

Table 10.5.: Underlying processes create typical probability distributions. That there is significant overlap with the distributions in Section 10.2.2 is not a coincidence.

Process	Distribution of Data	Examples
Many *additive* 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 *multiplicative* 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
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

Process	Distribution of Data	Examples
Continuous trials each with the same probability of success, measuring the time until the first success	Exponential	Time until a component fails, time until a sales call results in a sale
Waiting time until the r-th event occurs in a Poisson process	Gamma	Time until the 3rd customer arrives, time until the 5th defect occurs

Probability Distributions

There are a *lot* of specialized distributions. There are lists of distributions you can find online or in references such as Leemis and McQueston (2008) which has a full-page network diagram of the relationships.
The information-theoretic and Bayesian perspective on it is to eschew memorization of a bunch of special cases and statistical tests. If you pull up the aforementioned diagram in Leemis and McQueston (2008), you can see just a handful of distributions that have the most central roles in the universe of distributions. Many distributions are simply transformations, limiting instances, or otherwise special cases of a more fundamental distribution. Instead of trying to memorize a bunch of probability distributions, it's better to think critically about:

1. The fundamental processes that give rise to the randomness.
2. Tranformations 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.

10.2.2.2. Additive and Multiplicative Processes

Table 10.5 describes some examples, let us discuss further what it means to have a process that arises via an additive vs multiplicative effect[28].

[28] Mulitiplicative process 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.

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.

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:

$$Y = x_1 \times x_2 \times ... \times x_i$$

Then,

$$log(Y) = log(x_1) + log(x_2) + ... + \log(x_i)$$

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.

10.3. 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:

- $P(H \mid D)$ is the conditional probability of event H occurring given that D is true.
- $P(D \mid H)$ 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 10.4. First, let's look at combining information theory and Bayes' rule in an applied example.

10.3.1. 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"?

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:

$$BF = P(H_1|D)/P(H_2|D)$$

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

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

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:

- 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.

Null Hypthothesis 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 competiting theories.
Over 100 statistical tests have been developed in service of NHST Lewis (2013), but it's widely viewed now that a focus on NHST has led to *worse* science due to a multitude of factors, such as:

- "P-hacking" or trying to find subsets of data which can (often only by chance) support rejecting some null
- Cognitive anchoring to the importance of a p-value of 0.05 or less - why choose that number versus 0.01 or 0.001 or 0.49?
- Bias in research processes where one may stop data collection or experimentation after achieving a favorable test result
- Inappropriate application of the myriad of statistical tests
- Focus on p-values rather than effects that simply matter more or have greater effect
    - For example, which is of more interest to doctors? A study indicating a 1 in a billion chance of serious side effect (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 colleciton 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.

The example we'll look at relates to the annual rainfall totals for a specific location in California[29], which could be useful for insuring flood risk or determining the value of a catastrophe bond. Acknowloging that we are attempting to create a geocentric model[30] instead of a scientifically accurate weather model, we narrow the problem to finding a probability distribution that matches the historical rainfall totals. Our goal is to recommend a model that best fits the data and justify that recommendation quantitatively. Before even looking at the data, Table 10.6 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?

[29] https://data.ca.gov/dataset/annual-precipitation-data-for-northern-california-1944-current

[30] See @sec-predictive-vs-explanatory.

Table 10.6.: 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

Hypothesis	Process	Possible Rationale
H_3	A Gamma Distribution	Since rainfall totals would be the sum of exponentially-distributed independent rainfall events

```
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,
    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)
```

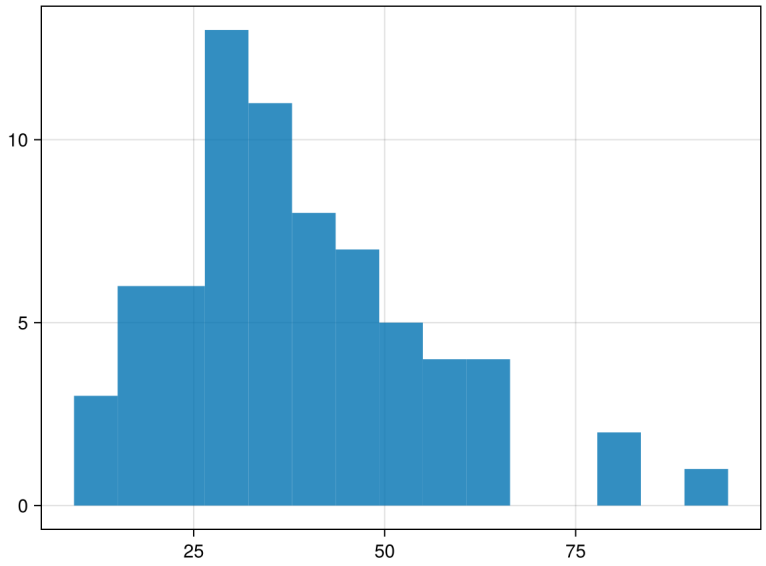


Figure 10.1.: Annual rainfall totals for a specific location in California.

We will show the likelihood of the three models after deriving the **maximum likelihood** (**MLE**), which is simply finding the parameters that maximize the calculated likelihood. In general, this can be accomplished by an optimization routine, but here we will just use the functions built into Distributions.jl:

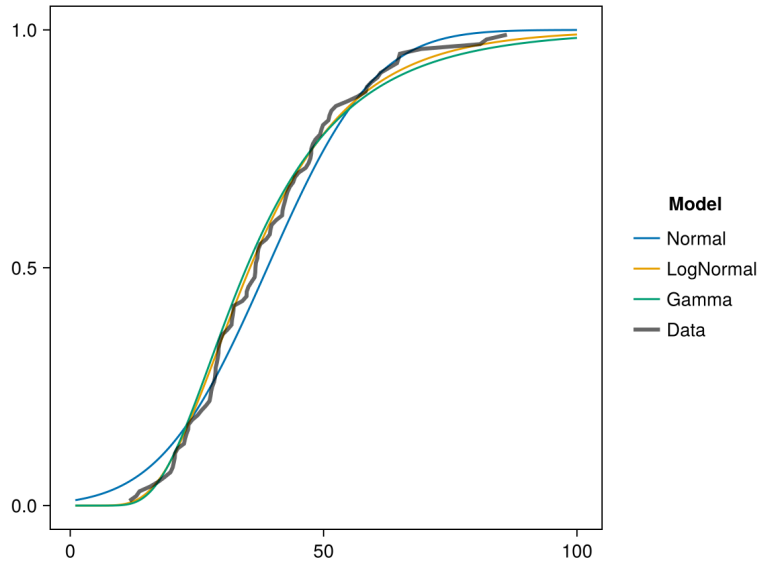
```
using StatsBase
using Distributions

n = fit_mle(Normal, rain)
c = 0.12                                                    ①
ln = fit_mle(Normal, log.(rain .- c))
g = fit_mle(Gamma, log.(rain .- c))
@show n
@show ln
@show g;
```

① The `0.12` is a constant translation factor. We can losslessly translate the data to back out some typical level of rainfall and allow the distributions to fit the variation in the residual rainfall amounts. It's not needed in the Normal distri-

bution case because the μ parameter allows the Normal distribution to translate itself to the center of the probability region.

```
n = Normal{Float64}(μ=38.91442857142857, σ=16.643603630714306)
ln = Normal{Float64}(μ=3.5652942753789882, σ=0.44331595919602507)
g = Gamma{Float64}(α=60.87983167649546, θ=0.05856281427196324)
```

```
let x = rain

    range = 1:0.1:100
    fig, ax, _ = lines(range, cdf.(n, range), label="Normal", axis=(xgridvisible=false, ygridvisi
    lines!(ax, range, cdf.(ln, log.(range .- c)), label="LogNormal")
    lines!(range, cdf.(g, log.(range .- c)), label="Gamma")
    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
# pdf(Normal(0,1),0.5)

# sum((pdf.(ln,x)))/sum((pdf.(n,x)))
# sum((pdf.(g,log.(x.-c))))/sum((pdf.(n,x)))
```



[TBD: commentary and comparison of the likelihoods.]

> **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.

10.4. Modern Bayesian Statistics

[DRAFTING NOTE: The content below was pulled from a blog post on bayesian statistics. The goal for the book is to adapt the concepts and follow the rainfall example by showing an MCMC example of fitting the distribution including the c parameter]

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.

10.4.1. 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.

10.4.2. 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 makes 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.

10.4.3. Why Now?

There are both philosophical and practical reasons why Bayesian analysis is rapidly changing the statistical landscape.

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 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!

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

10.4.4. 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.

10.4.4.1. Subjectivity as a Feature

A Bayesian approach to defining a statistical model is an approach that allows for explicitly incorporating actuarial 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.

10.4.4.2. 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 credibility[31] and therefore should be similarly reconsidered.

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

10.4.4.3. Maximum Entropy Distributions

Further, when assigning a prior assumption to a random variable, there are mathematically most conservative choices to pull from. These are called Maximum Entropy Distributions (MED) and it can be shown that for certain minimal constraints these are the information-theoretic least informative choices. Least informative means that the prior will have the least influence on the resulting posterior distribution.

For example, if all you know is that the mean of a random process is positive, then the Exponential Distribution is your MED. If you know that a mean and variance must exist for the process, then the Normal distribution is your MED. If you know nothing at all, you can use a Uniform distribution for the possible values.

10.4.4.4. 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 article:

- 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.

10.4.5. Implications for Risk Management

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 tradeoffs between different choices of model.

10.4.6. Paving the Way Forward for Actuaries

Bayesian approaches to statistical problems are rapidly changing the professional statistical field. To the extent that the actuarial profession incorporates statistical procedures we should consider adopting the same practices. The benefits of this are a better understanding of the distribution of risks, 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 actuarial best practices related to understanding and communicating about risk.

For actuaries interested in learning more, there are number of available resources to be found. Textbooks recommended by the author are:

- Statistical Rethinking (McElreath)
- Bayes Rules! (Johnson, Ott, Dogucu)
- Bayesian Data Analysis (Gelman, et. al.)

Additionally, the author has published a few examples of Bayesian analysis in an actuarial context on JuliaActuary.org.

Part III.

Computational Thinking in an Actuarial and Financial Context

11. Modeling

11.1. 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.

11.2. Parsimony

12. Automatic Differentiation

12.1. In This Chapter

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

12.2. 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 let's you optimize things faster (see Chapter 13). You can test properties and implications (monotonicy, maxima/minima).

12.3. 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 point 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:

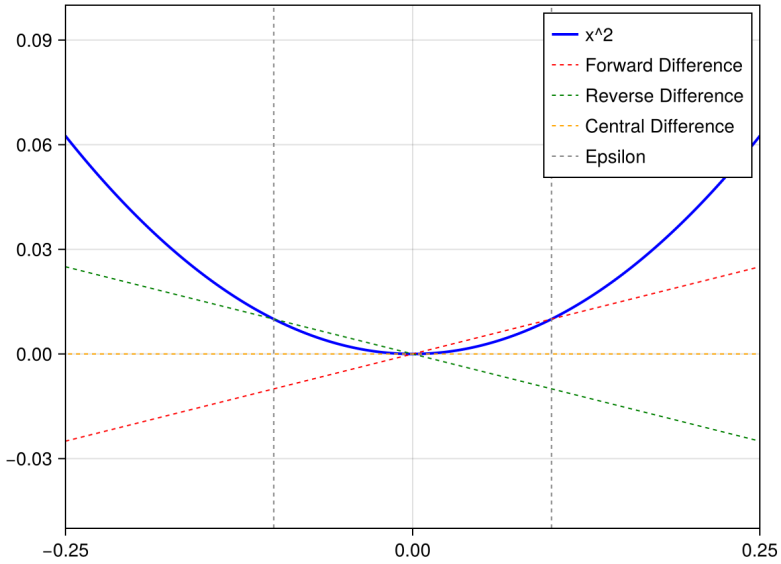
$$f'(x) = \lim_{\epsilon \to 0} \frac{f(x_0 + \epsilon) - f(x_0)}{\epsilon}$$

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:

- forward difference is as defined in the above equation, where ϵ is added to x_0
- forward 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:



One benefit of the central difference method is that 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 evaluating a function at a gas determining what the derivative of the function is at that point and using both items to update to help better guess. At each step you need to evaluate the function three times for $x4, $x + \epsilon$, 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 further 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 amore 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
```

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

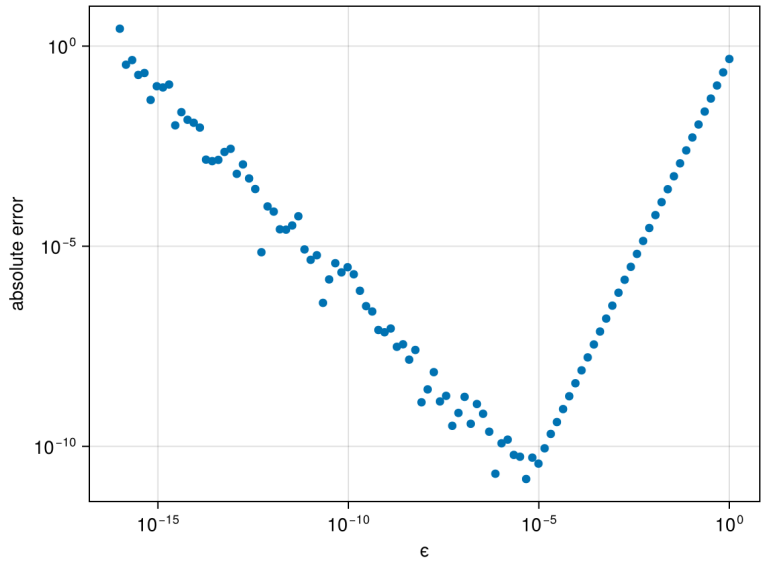


Figure 12.1.: A log-log plot showing the absolute error of the finite differences. Further to the left, roundoff error dominates while further to the right, truncation error dominates.

> **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 $\epsilon \to 0$. The 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 depend-

ing on the function and the point we are attempting to evaluate. This presents a number of practical diffuclties 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 depends on the ability to accurate 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.

12.4. Automatic Differentiation

Automatic differentiation ("autodiff" or "AD" for short) is essentially the practice of defining algorithmically what the derivatives of function should be. We are able to do this through a creative application of the chain rule. Recall that the **chain rule** allows us to compute the derivative of a composite function using the derivatives of the component functions:

$$h(x) = f(g(x))$$

$$h'(x) = f'(g(x))g'(x)$$

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.

12.4.1. 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 numbers[32]. 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.

[32] Richard Feynman has a wonderful, short lecture on algebra here: https://www.feynmanlectures.caltech.edu/I_22.html

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
```

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:

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

```
Base.:*(d::DualNumber, e::DualNumber) = DualNumber(d.a * g.a, d.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`.

> **Note**
>
> When calcualting the derivative, why do we start with `1` in the dual part of the number? Because the derivative of a variable with respect to itself is 1. From this unitary starting point, the various operations applied accumulate the derivative of the various operations in the b part of $a + \epsilon b$.

We can also define this for things like transcendental functions:

```
Base.exp(d::DualNumber) = DualNumber(exp(d.a), exp(d.a) * d.b)
Base.sin(d::DualNumber) = DualNumber(sin(d.a), cos(d.a) * d.b)
Base.cos(d::DualNumber) = DualNumber(cos(d.a), -sin(d.a) * d.b)
exp(DualNumber(1, 1))
```

```
DualNumber{Float64, Float64}(2.718281828459045, 2.718281828459045)
```

```
sin(DualNumber(0, 1))
```

```
DualNumber{Float64, Float64}(0.0, 1.0)
```

```
cos(DualNumber(0, 1))
```

```
DualNumber{Float64, Float64}(1.0, -0.0)
```

And finally, to put it all together in a more usable wrapper, we can define a function which will calcualte 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 interst as well.

12.5. Performance of Automatic Differentiation

Recall that in the finite difference method, we generally had to evalue 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?

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

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

```
7.9487355427201685
```

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

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

```
DualNumber{Float64, Int64}(6.674097289031293, 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 pronouced 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.

12.6. 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 vect

- `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                    ①
    iszero(τ) && return max(zero(S), S - K)
    d₁ = d1(S, K, τ, r, σ, q)
```

```
    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 derivaties with respect to each parameter. The first, third, fourth, fifth, and

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
```

12.6.1. 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)
```

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

```
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.

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

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

12.7. Forward Mode and Reverse Mode

The approach of autodiff outlined about is called *forward mode **auto-differetiation 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.

12.8. Practical tips for Automatic Differentiation

Here are a few practical tips to keep in mind.

12.8.1. 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.

12.8.2. 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.

12.8.3. Custom Rules

Custom rules for new or unusal 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.

12.8.4. 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.
- **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.

12.9. References

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

13. Optimization

13.1. In This Chapter

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

14. Sensitivity Analysis

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

14.1. 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.

14.2. 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
```

```
    risk::Risk
end
```

14.3. The Data

```
sample_csv_data =
    IOBuffer(
        raw"id,sex,benefit_base,COLA,mode,issue_date,issue_age,ris
         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, 0.02257
 Female => [0.00745, 0.00745, 0.00745, 0.00745, 0.00745, 0.00745, 0.0
```

```
policies = let

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

```
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 $p_x = p_(x-1)*(1-mort_(age+y))/(1+COLA)$ for $x >= y$, and 0 for $x < y-1$ or $age+x >= 120$, 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
```

```
    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
end
```

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

14.4. Common Sensitivity Analysis Methodologies

14.4.1. 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)
```

14.4.2. Scenario Analyses

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

14.4.3. 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
```

14.4.4. 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 $\mathrm{Var}(Y)$.

```
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
```

14.4.5. 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:

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

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
```

14.4.6. Fourier Amplitude Sensitivity Tests

```
gsa(r1_wrt_r, eFAST(), [[1, 1.01], [0.025, 0.035], [10000.0, 10000
```

```
GlobalSensitivity.eFASTResult{Matrix{Float64}}([4.0660048390871604
```

14.5. Benchmarking

15. 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. Kalos[33] (c. 1960)

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

16. Visiualizations

16.1. 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.

17. Matrices and Their Uses

17.1. 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.

18. Learning from Data

18.1. 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.

Part IV.

Applications in Practice

19. Stochastic Mortality Projections

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

19.1. 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.

19.2. 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).

```
@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
```

19.3. The Data

```
sample_csv_data =
    IOBuffer(
        raw"id,sex,benefit_base,COLA,mode,issue_date,issue_age,ris
         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=t
```

```
policies = let

    # read CSV directly into a dataframe
    # df = CSV.read("sample_inforce.csv",DataFrame) # use local st
    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


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 acheiving 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.

```
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
        else
            # pay benefit, add a life to the output count, and inc
            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,
)
```

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

Check the number of threads we're using:

```
Threads.nthreads()
```

```
1
```

```
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)
```

19.4. Running the projection

Example of a single projection:

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

```
(benefits = [5.6286304076594894e10, 5.673113897968274e10, 5.708777000920091e10, 5.742102220964411e1
```

19.4.1. Stochastic Projection

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

```
function stochastic_proj(policies, params, n)

    ThreadsX.map(1:n) do i
        project(policies, params)
    end
end
```

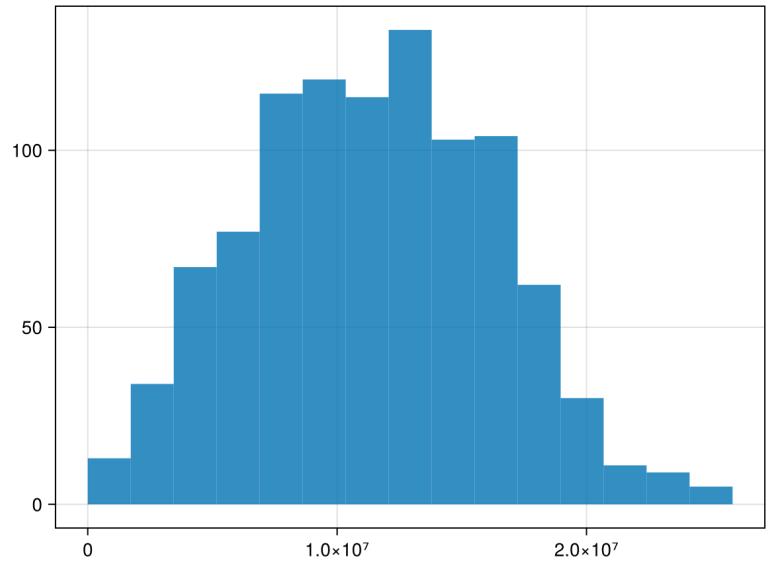
```
stochastic_proj (generic function with 1 method)
```

```
stoch = stochastic_proj(policies, params, 1000)
```

```
1000-element Vector{@NamedTuple{benefits::Vector{Float64}, lives::V
 (benefits = [383220.68177215644, 394717.3022253211, 406558.82129208
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 ⋮
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 (benefits = [574831.0226582347, 592075.9533379817, 406558.821292080
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
 (benefits = [574831.0226582347, 592075.9533379817, 609838.231938121
```

```
let
    v = [pv(0.03, s.benefits) for s in stoch]
```

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



19.5. 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)
```

```
BenchmarkTools.Trial: 19 samples with 1 evaluation.
 Range (min … max):  207.715 ms … 735.127 s  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     210.744 ms              ┊ GC (median):    0.00%
 Time  (mean ± σ):    38.891 s ± 168.601 s   ┊ GC (mean ± σ):  0.00% ± 0.00%

  █
  █▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▄ ▁
  208 ms          Histogram: frequency by time          735 s <

 Memory estimate: 9.28 KiB, allocs estimate: 79.
```

19.6. Further Optimization

In no particular order:

- the RNG could be made faster: https://bkamins.github.io/julialang/202
- 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
- ...

20. Scenario Generation

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

20.1. In This Chapter

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

20.2. Setup

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

20.3. The Data

20.4. 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

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.

20.4.1. Common PRNGs

20.4.1.1. 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.

20.4.1.2. 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

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.

20.4.1.3. 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.

20.4.2. 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}:
 -2601283040775479866  -4343244971702185926  -7307871539997467818
  3992464865188821787  -5025293388170100914   6043741968597453537
```

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

```
2×3 Matrix{Float64}:
 0.0743715  0.189755  0.581641
 0.373538   0.91915   0.126339
```

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

```
2×3 Matrix{Bool}:
 0  0  0
 0  1  1
```

20.5. 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.

20.5.1. Interest Rate Models

20.5.1.1. 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.

- $dW(t)$ is a Wiener process or Brownian motion, representing a random shock.

And CIR is defined as

$$dr(t) = \kappa(\theta - r(t))\,dt + \sigma\sqrt{r(t)}\,dW(t)$$

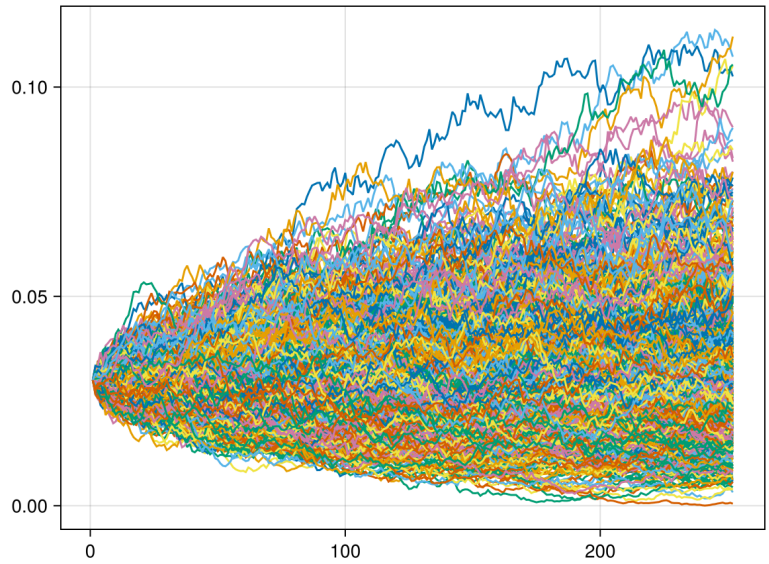
where

- $r(t)$ 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
```

```
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 *
            interest_rate_paths[i, j] = max(interest_rate_paths[i-
        end
    end
    return interest_rate_paths
end

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

# 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
```



20.5.1.2. 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

$$dr(t) = (\theta(t) - ar(t))\,dt + \sigma(t)\,dW(t)$$

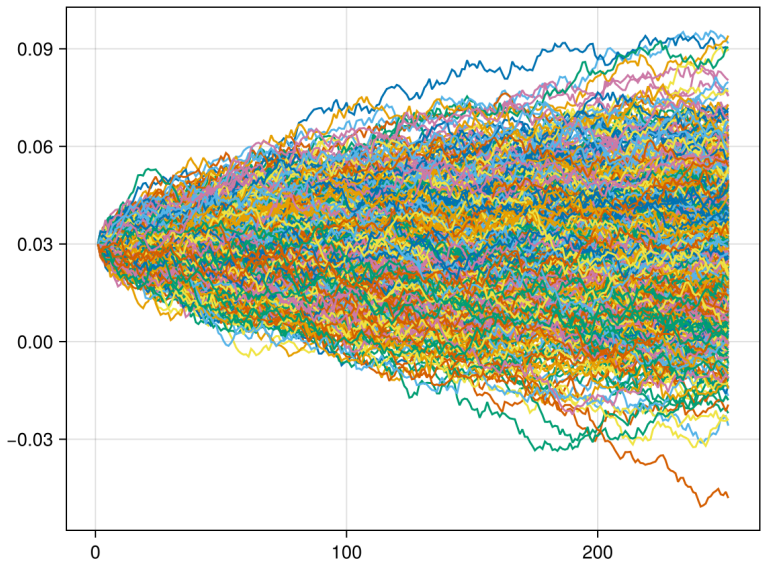
where

- $r(t)$ 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.

```
# 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_simula
    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 *
            interest_rate_paths[i, j] = interest_rate_paths[i-1, j
        end
    end
    return interest_rate_paths
end

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

# 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



20.5.2. Stock Models

20.5.2.1. Geometric Brownian Motion (GBM)

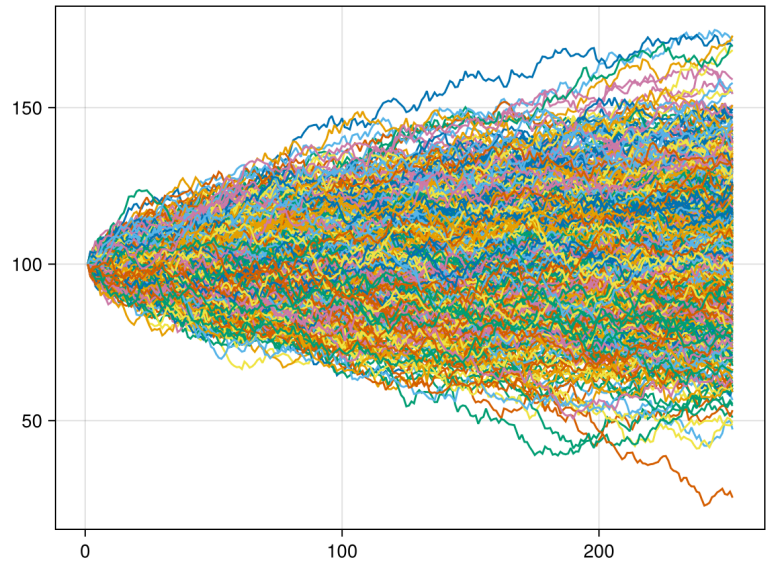
GBM is a stochastic process commonly used to model the price movement of financial instruments, including stocks. It assumes constant volatility and is characterized by a log-normal distribution. It is defined as

$$dS(t) = \mu S(t)\,dt + \sigma S(t)\,dW(t)$$

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)

# 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] + 
        end
    end
    return stock_price_paths
end

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

# 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
```



20.5.2.2. 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

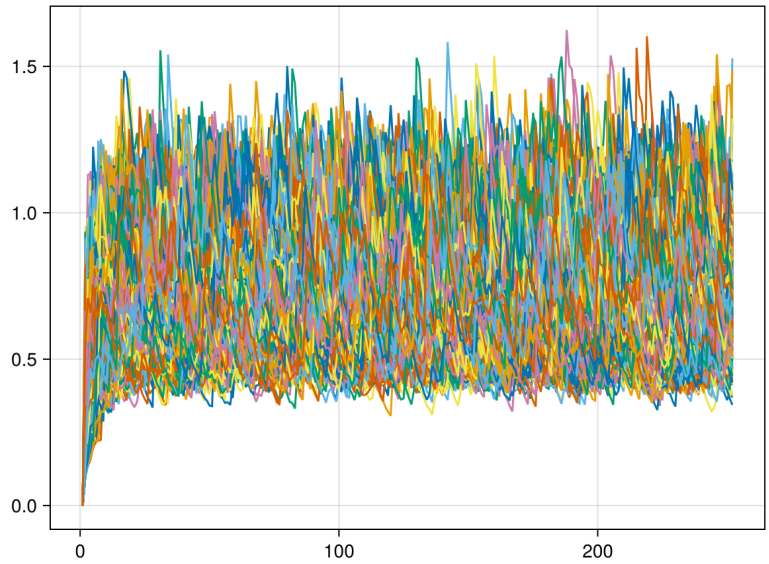
$$\sigma_t^2 = \omega + \alpha_1 r_{t-1}^2 + \beta_1 \sigma_{t-1}^2$$

$$r_t = \varepsilon_t \sqrt{\sigma_t^2}$$

- 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
- ω, α_1, β_1 are model parameters

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

# GARCH(1,1) parameters
α₀ = 0.01       # Constant term
```

```
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
            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_simulati

# 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
```

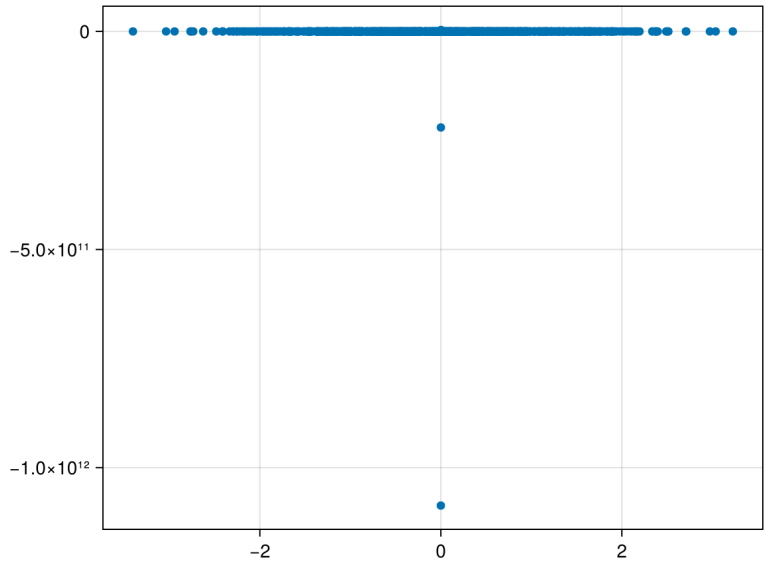


20.5.3. Copulas

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)
```

```julia
# Clayton copula simulation
function clayton_copula_simulation(u1, u2, theta)
    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
```



20.6. Benchmarking

21. Similarity Analysis

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

21.1. 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.

21.2. Setup

```
using CSV, DataFrames
using LinearAlgebra
using StatsBase, TableTransforms
using CairoMakie
using NearestNeighbors
```

21.3. 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

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
	Float64	Float64
1	0.25	0.25
2	0.75	0.0
3	0.5	0.5
4	0.0	1.0
5	1.0	0.5
6	0.75	0.25

```
arr_cat = hcat(indicatormat(df.sex)', indicatormat(df.education)',
```

```
6×9 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.

21.4. Common Similarity Measures

The following measures are commonly used to calculate similarities.

21.4.1. 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 = √(∑((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
```

21.4.2. 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.

```
#d₁₂ = ∑(abs.(Array(df_num[1, :]) .- Array(df_num[2, :])))
d₁₂ = LinearAlgebra.norm1(Array(df_num[1, :]) .- Array(df_num[2, :
```

```
0.75
```

21.4.3. 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.

```
d₁₂ = (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.

```
d₁₃ = (arr_cat[1, :] ⋅ arr_cat[3, :]) / norm(arr_cat[1, :]) / norm
```

```
0.33333333333333337
```

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.

21.4.4. 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
```

21.4.5. Hamming Distance

Hamming distance is defined as d = Number of positions at which w and v 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, :])
```

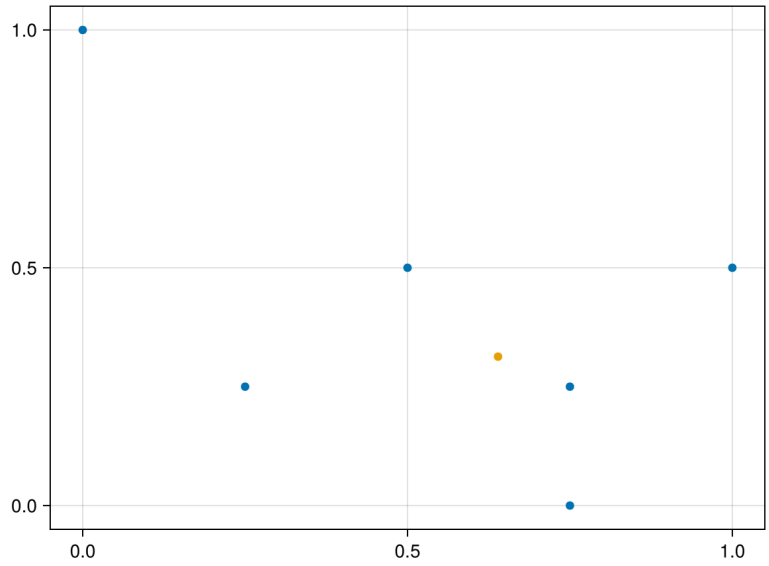
```
4
```

21.5. 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: [0.6393453058053142, 0.31316648559296734]
Nearest Neighbors Indices: [2]
Distances to Neighbors: [0.4996629012156935]
```



21.6. Benchmarking

22. Portfolio Optimization

22.1. In This Chapter

Optimization in a portfolio context with examples of asset selection under different constraints and objectives.

23. Bayesian Mortality Modeling

```
### A Pluto.jl notebook ###
# v0.19.19

using Markdown
using InteractiveUtils

# ╔═╡ b513bc8a-f08d-4d27-8d05-d60885bb03df
begin
    using MortalityTables
    using Turing
    using UUIDs
    using DataFramesMeta
    using MCMCChains, Plots, StatsPlots
    using LinearAlgebra
    using PlutoUI; TableOfContents()
    using Pipe
    using StatisticalRethinking
    using StatsFuns
end

# ╔═╡ 74e4511f-cf5f-4544-8bd2-ea228dfa700e
md"""

## Generating fake data

The problem of interest is to look at mortality rates, which are given in terms of exposures (whe

We'll grab some example rates from an insurance table, which has a "selection" component: When so

Addtionally, there may be additional groups of interest, such as:
```

```
- high/medium/low risk classification
- sex
- group (e.g. company, data source, etc.)
- type of insurance product offered

The example data will start with only the risk classification abov
"""

# ╔═╡ 3249443a-a8e5-48f1-9eed-379c86144e81
src = MortalityTables.table("2001 VBT Residual Standard Select and

# ╔═╡ c931c097-57a1-4f51-857c-b02d3547456f
src.select[50]

# ╔═╡ fdfd1e29-5f8a-405b-9212-7ac08e52ffab
n = 10_000

# ╔═╡ da661ce2-eadf-4ddb-a6c4-5c00dc2caae4
function generate_data_individual(tbl,issue_age=rand(50:55),inforc
    # risk_factors will scale the "true" parameter up or down
    # we observe the assigned risklevel, but not risk_factor
    risk_factors = [0.7,1.0,1.5]
    rf = risk_factors[risklevel]
    deaths = rand(inforce_years) .< (tbl.select[issue_age][issue_a

    endpoint = if sum(deaths) == 0
        last(inforce_years)
    else
        findfirst(deaths)
    end
    id= uuid1()
    map(1:endpoint) do i
        (
        issue_age=issue_age,
        risklevel = risklevel,
        att_age = issue_age + i -1,
        death = deaths[i],
        id = id,
    )
    end
```

```
end

# ╔═╡ 4a77aad1-1a1f-484d-b128-526ee9f3a4a8
exposures = vcat([generate_data_individual(src) for _ in 1:n]...) ▷ DataFrame

# ╔═╡ c7d8c2fe-838d-4521-beeb-e471e443107a
data = combine(groupby(exposures,[:issue_age,:att_age])) do subdf
    (exposures = nrow(subdf),
    deaths = sum(subdf.death),
    fraction = sum(subdf.death)/ nrow(subdf))
end


# ╔═╡ 45237199-f8e8-4f61-b644-89ab37c31a5d
data2 = combine(groupby(exposures,[:issue_age,:att_age,:risklevel])) do subdf
    (exposures = nrow(subdf),
    deaths = sum(subdf.death),
    fraction = sum(subdf.death)/ nrow(subdf))
end


# ╔═╡ d23aa389-edfe-4a0d-9924-451b88beb83b
md"
## 1: A single binomial parameter model

Estiamte $p$, the average mortality rate, not accounting for any variation within the population/
"

# ╔═╡ c52bbfab-07f6-40d0-a666-24fbda2435c2
@model function mortality(data,deaths)
    p ~ Beta(1,1)
    for i = 1:nrow(data)
        deaths[i] ~ Binomial(data.exposures[i],p)
    end
end

# ╔═╡ 18abd59e-ef16-462b-8357-157afc64812b
m1 = mortality(data,data.deaths)

# ╔═╡ 7e739879-241c-49fe-b48c-4245942edda4
num_chains = 4
```

```
# ╔═╡ dfa6c8c4-14b3-4c1b-922b-8582cc3243fb
md"### Sampling from the posterior

We use a No-U-Turn-Sampler (NUTS) technique to sample multile chai

# ╔═╡ 8184820b-0a52-431b-b000-243c7ea9e1ea
chain = sample(m1, NUTS(), 1000)

# ╔═╡ 0edd523e-92bd-4c9c-9cd9-0cd990e72706
plot(chain)

# ╔═╡ 35de8c2c-8e33-4f76-92ba-ce3dfa635cd8
md"### Plotting samples from the posterior

We can see that the sampling of possible posterior parameters doesn

This is saying that for the observed data, if there really is just

# ╔═╡ 74af0a79-292a-4fba-a052-991d3a74c9eb
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 exp
        xlabel="age",
        ylim=(0.0,0.25),
        ylabel="mortality rate",
        title="Parametric Bayseian Mortality"
    )

    # show n samples from the posterior plotted on the graph
    n = 300
    ages = sort!(unique(data.att_age))

    for i in 1:n
```

```
        p_posterior = sample(chain,1)[:p][1]
        hline!([p_posterior],label="",alpha=0.1)
    end
    p

end

# ╔═╡ 5faa1505-dbde-48d7-a370-3215c5d73a8c
md"The posterior mean of `p` is of course very close to the simple proportoin of claims to exposu

# ╔═╡ 2c3a27a1-b626-4654-9822-2a391c55371d
mean(chain,:p)

# ╔═╡ 84d2dc52-a0c9-4c30-a3cf-7c174f4a046b
sum(data.deaths) / sum(data.exposures)

# ╔═╡ 49bf0bd3-ad5a-409e-a25e-54f7aea44eb3
md"## 2. Parametric model

In this example, we utilize a [MakehamBeard](https://juliaactuary.github.io/MortalityTables.jl/st

The **prior** values for `a`,`b`,`c`, and `k` are chosen to constrain the hazard (mortality) rate

This isn't an ideal parameterization (e.g. we aren't including information about the select under
"

# ╔═╡ 599942d8-a8d9-40b0-b7fe-b893306dcbcf
@model function mortality2(data,deaths)
    a ~ Exponential(0.1)
    b ~ Exponential(0.1)
    c = 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 the data
    # 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

# ╔═╡ ef109946-7e12-4959-9cac-13bbbd436504
md" We combine the model with the data:"

# ╔═╡ ff8616f8-b813-4498-844b-04608986d970
m2 = mortality2(data,data.deaths)

# ╔═╡ cb1a6c73-1464-44a7-97a1-f66b7210f09d
md"### Sampling from the posterior

We use a No-U-Turn-Sampler (NUTS) technique to sample:"

# ╔═╡ 251b0b70-5ba5-415b-970b-e55280cba222
chain2 = sample(m2, NUTS(), 1000)

# ╔═╡ 229d8a8e-197d-42fd-8107-a7826a394c6a
summarize(chain2)

# ╔═╡ 657abb55-f4eb-44f9-83e1-6cfef7c2516d
plot(chain2)

# ╔═╡ d72f1350-83ca-4b4a-b16d-3d00b61b97b2
md"### Plotting samples from the posterior

We can see that the sampling of possible posterior parameters fits

# ╔═╡ 85c29fd3-0045-4468-966f-64c2ccb9ce8b
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 ex
        xlabel="age",
```

```
        ylim=(0.0,0.25),
        ylabel="mortality rate",
        title="Parametric Bayseian 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)
        plot!(ages,age -> MortalityTables.hazard(m,age), alpha = 0.1,label="")
    end
    p
end

# ╔═╡ a4f7fefa-3f18-4d1c-a8d7-8ed334552966
md"## 3. Parametric model

This model extends the prior to create a multi-level model. Each risk class (`risklevel`) gets it
"

# ╔═╡ 964df467-234c-4aac-a12b-c22f3ff1e07c
@model function mortality3(data,deaths)
    risk_levels = length(levels(data.risklevel))
    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 the data
    # and how consistent the parameters are with the data
```

```
    for i = 1:nrow(data)
        risk = data.risklevel[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

# ╔═╡ 5f4ddd19-86ae-4e05-81a2-cc730f4bc0c0
m3 = mortality3(data2,data2.deaths)

# ╔═╡ da15bdb5-c872-4837-a6bb-afe164d1d4cf
chain3 = sample(m3, NUTS(), 1000)

# ╔═╡ e98641db-b2f6-4783-8c29-e6d8b8b4c86d
summarize(chain3)

# ╔═╡ 30d0bd29-415b-4d48-a6b3-52d46fed246c
PRECIS(DataFrame(chain3))

# ╔═╡ 7b026314-3118-42c5-9214-2d5675df769d
let data = data2

    data_weight = data.exposures ./ sum(data.exposures)
    data_weight = .√(data_weight ./ maximum(data_weight) .* 20)
    color_i = data.risklevel

    p = scatter(
        data.att_age,
        data.fraction,
        markersize = data_weight,
        alpha = 0.5,
        color=color_i,
        label = "Experience data point (size indicates relative ex
        xlabel="age",
        ylim=(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(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)
            if i == 1
                plot!(ages,age -> MortalityTables.hazard(m,age),label="risk level $r", alpha = 0.
            else
                plot!(ages,age -> MortalityTables.hazard(m,age),label="", alpha = 0.2,color=r)
            end
        end
    end
    p
end

# ╔═╡ b888b185-9797-4b4a-8862-c320d427e828
md"## Handling non-unit exposures

The key is to use the Poisson distribution:
"

# ╔═╡ 6d2a0fdc-7627-4942-9b8a-5a6da3aebe85
@model function mortality4(data,deaths)
    risk_levels = length(levels(data.risklevel))
    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.risklevel[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

# ╔═╡ 80102168-610d-4956-9ea4-4f6e45e9968a
m4 = mortality4(data2,data2.deaths)

# ╔═╡ b7fda1be-4e6d-4c16-bfe0-4486c46c3d49
chain4 = sample(m4, NUTS(), 1000)

# ╔═╡ 210ca9dd-22c4-426f-86f9-6ef2b6f78a1a
PRECIS(DataFrame(chain4))

# ╔═╡ 7054d2a2-6fda-4d26-9583-f325ac9a5d9c
risk_factors4 = [mean(chain4[Symbol("a[$f]")]) for f in 1:3]

# ╔═╡ a84fa049-3465-4d19-8bc1-5b622362da63
risk_factors4 ./ risk_factors4[2]

# ╔═╡ 64b1f9d6-6249-438d-aacf-a185914601a8
let data = data2

    data_weight = data.exposures ./ sum(data.exposures)
    data_weight = .√(data_weight ./ maximum(data_weight) .* 20)
    color_i = data.risklevel

    p = scatter(
        data.att_age,
        data.fraction,
        markersize = data_weight,
        alpha = 0.5,
        color=color_i,
        label = "Experience data point (size indicates relative ex
```

```
        xlabel="age",
        ylim=(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
                plot!(ages,age -> MortalityTables.hazard(m,age),label="risk level $r", alpha = 0.
            else
                plot!(ages,age -> MortalityTables.hazard(m,age),label="", alpha = 0.2,color=r)
            end
        end
    end
    p
end

# ╔═╡ da87eff5-7da8-4c55-9f1c-bbbc0ada980e
md"## Predictions

We can generate predictive estimates by passing a vector of `missing` in place of the outcome var

We get a table of values where each row is the the prediction implied by the corresponding chain
"

# ╔═╡ 5e4b89af-a1f4-4917-aa10-960d8899de5a
preds = predict(mortality4(data2,fill(missing,length(data2.deaths))),chain4)

# ╔═╡ d3aa1aaa-b7c9-411e-bf8a-1583728e3734
```

```
size(preds)
```

24. Other Useful Techniques

24.1. 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.

24.2. 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?

24.3. 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 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.

Part V.

Appendices

25. Set up Julia and the Computing Environment

25.1. Installation

Julia is open source and can be downloaded from 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.

25.2. Package Management

Julia comes with `Pkg`, a built-in package manger. 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.

- 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.

25.3. 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:

25.3.1. 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.

25.3.2. 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).

26. Environment and Package Management

26.1. 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.

26.2. 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`.

26.2.1. 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.

> **Note**
>
> TOML (Tom's Obvious Markup Language) is a modern configuration file format used to store settings and data in a human-readable, plaintext format.

This is a bit abstract, so here is a quick, annotated tour of an example Project.toml file:

```
name = "FinanceCore"                                          ①
uuid = "b9b1ffdd-6612-4b69-8227-7663be06e089"                 ②
authors = ["alecloudenback <alecloudenback@users.noreply.github.co
version = "2.1.0"                                             ③

[deps]                                                        ④
Dates = "ade2ca70-3891-5945-98fb-dc099432e06a"
LoopVectorization = "bdcacae8-1622-11e9-2a5c-532679323890"
Roots = "f2b01f46-fcfa-551c-844a-d8ac1e96c665"

[compat]                                                      ⑤
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 commit[34].
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.

[34] When registering a package to a repository, the repository will record the version indicated in the `Project.toml` file to the git commit id of the package when it is registered.

When you `instantiate` a project (see Section 26.3 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

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).

26.2.2. 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
```

> **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.

26.2.3. 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.

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 9.

26.3. Environemnts

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>
```

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.)

26.4. Packages

26.4.1. Packages versus Projects

26.4.2. Basic Package Structure

26.4.3. Extension Packages

26.5. Regisries

26.5.1. Local Registries

27. 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 Packages[^2]

27.0.1. 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 for more packages and information.

27.0.2. 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 it's main downside is that it has not yet been optimized to minimize the time-to-first-plot.

27.0.3. 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.

27.0.4. 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.

27.0.5. 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:

```
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 =

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.

27.0.6. 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%

  33.5 ns       Histogram: log(frequency) by time      45.6 ns <

 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.

27.0.7. 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/.

27.0.8. 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.

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.