# Computational Thinking for Actuaries and Financial Professionals

**With Applications in Julia**

Alec Loudenback and Yun-Tien Lee

2024-12-16

# Table of contents

# 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.

> Warning
>
> This book is currently being drafted. ANY AND ALL content is subject to change, including the license. To report issues or other feedback, please email Alec at firstnamelastname@gmail.com.

# 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.

Even if your direct responsibility does not entail hands-on-coding, be it management or "low-code", the ideas and language should prove useful in guiding the work to a cleaner, more efficient solution.

## The Journey Ahead

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

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

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

### What to Expect

This book will guide you through:

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

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

## Prerequisites

Basic experience with financial modeling is not strictly required, but 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 4 introduces the basic syntax and concepts while **?@sec-julia-basics** covers setting up your environment to follow along. For readers with background in programming, we recommend skimming Chapter 4.

## 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 **?@sec-julia-basics**) 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 **?@sec-environments**), 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 and LaTeX.

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

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

The cover was designed by Alec Loudeback using Affinity Designer with the graphic used under permission by user cormullion on Github.

This book was rendered on October 25, 2024. The system used to generate the code and benchmarks was:

```julia
versioninfo()
```

```
Julia Version 1.11.1
Commit 8f5b7ca12ad (2024-10-16 10:53 UTC)
Build Info:
  Official https://julialang.org/ release
Platform Info:
  OS: macOS (arm64-apple-darwin22.4.0)
  CPU: 8 × Apple M3
  WORD_SIZE: 64
  LLVM: libLLVM-16.0.6 (ORCJIT, apple-m3)
Threads: 4 default, 0 interactive, 2 GC (on 4 virtual cores)
Environment:
  JULIA_NUM_THREADS = auto
```

## License and Copyright

# 1. 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)

## 1.1. 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.

## 1.2. Introduction

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

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

At the forefront of this shift is the growing importance of programming skills. In the beginning of the computer era in finance, the differentiating skill was being able to utilize digital computing, data processing, and calculation engines to automate, analyze, and report on the business. These skills required low level programming and the success of many of those early programs is evident in their legacy: many of them are still around in the 2020s!

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

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

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

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

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

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

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

### 1.2.1. Market Forces

Today, there is a trend towards 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[1]; Airlines are so dependent on their systems that the skies become quieter on the rare occasion that their computers give way. 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 challenging yield environments have reduced companies' comparative advantage to "manage money". Spread compression and the explosion of consumer-oriented investment services means a more competitive focus on the ability to manage the entire asset or policy's 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.

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

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

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

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

## 1.3. Why Programming Matters Now

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

1. **Enhanced Analysis Capabilities**: Programming allows for more complex analyses, handling of larger datasets, and application of advanced statistical and machine learning techniques.

2. **Automation and Efficiency**: Repetitive tasks can be automated, freeing up time for more value-added activities.

3. **Customization**: Bespoke solutions can be developed to address unique business needs, risk frameworks, and regulatory requirements.

4. **Data Handling**: As data volumes grow, programming provides tools to efficiently process, analyze, and derive insights from vast amounts of information.

5. **Integration**: Programming skills enable better integration across different systems and data sources, providing a more holistic view of financial operations.

6. **Competitive Edge**: In an increasingly technology-driven industry, programming skills can be a significant differentiator.

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

## 1.4. The Spectrum of Programming in Finance

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

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

## 1.5. Avoiding Red Herrings

One tantalizing path to contemplate is to avoid *really* learning how to code. Between artificial intelligence (AI) solutions being developed and low-code offerings, is there really a need to learn the fundamentals of coding? We argue that there is, for the basic reason that coding is not about mechanically typing out lines in an editor, but both a tool and a craft that is designed to enhance and apply your own creative and logical thinking.

The current generation of AI is fundamentally limited. Yann LeCun, Meta's (Facebook's) Chief AI Scientist describes the large-language model (LLM) approach as not even at the level of intelligence of a cat, and that we are decades away from true artificial general intelligence (AGI). These models have a "very limited understanding of logic . . . do not understand the physical world, do not have persistent memory, cannot reason in any reasonable definition of the term and cannot plan . . . hierarchically" (Murphy and Criddle 2024).

An important role for AI to play will be to *support* modelers in boilerplate, syntactical hurdles ("in VBA I would do it like this, but in Julia how do I do X, Y, or Z"), and basic algorithmic support. What is not likely to change in the short term is most of the value that a modeler brings to the table: creative thinking, understanding of company and market dynamics, and capability to understand broader architecture and conceptual aspects of modeling.

A similarly fraught path is low-code solutions. Low-code solutions are inherently limiting in their capabilities and lock you into a particular vendored solution. If you know enough about what you are trying to do to be able to state it in clearly in plain English, then you are most of the way to being able to program in a full coding solution (AI can actually help bridge this gap here). As soon as you hit a limitation of the system ("I'd like to use XYZ optimization algorithm at each timestep"), you are reliant on the vendor to implement that option in the "low code" solution. Further, you are out-sourcing a lot of the important inner-workings of the model to someone else and not building that expertise yourself of in-house somewhere.

## 1.6. The 10x Modeler

The increasingly complex business needs will highlight a large productivity difference between a financial modeler who can code and one who can't — simply because the former can react, create, synthesize, and model faster than the latter. From the efficiency of transforming administration extracts, summarizing and aggregating valuation output, to analyzing available data in ways that spreadsheets simply can't handle, you can become a "**10x Modeler**"[2].

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

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 take it a step further. Tasks like visualizations, sensitivity analysis, summary statistics, stochastic analysis, or process automation, when done programmatically, are often just a few lines of additional code over and above the primary model.

Should you drop 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 professionals' work and this trend appears to be accelerating.

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 and

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

inner source to support their custom modeling, risk analysis/monitoring, and reporting workflows.

**i** Note

**Open source** refers to software whose source code is freely available for anyone to view, modify, and distribute. It promotes collaboration, transparency, and innovation by allowing developers worldwide to contribute to and improve the codebase. Open source projects often benefit from diverse perspectives and rapid development cycles, resulting in robust and widely-adopted solutions.
**Inner source** applies open source principles within a single organization. It encourages internal collaboration, code sharing, and transparency across different teams or departments. By adopting inner source practices, companies can reduce duplication of effort, improve code quality, and foster a culture of knowledge sharing. This approach can lead to more efficient development processes and better utilization of internal resources.

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

## 1.7. 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

These aspects of business processes are what technology companies *excel* at. There is a litany of highly robust, battle-tested tools used in the information services sectors. This book will introduce much of this to the financial professional (specifically Chapter 11, Chapter 20, and Chapter 22).

## 1.8. Managing and Leading the Transformation

For managers: 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 as well as the most essential skill in programming. This perspective also allows for a vision of where the organization *should be* instead of thinking about where it is now.

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

## 1.9. 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 their expertise into new forms of analysis - doing it faster and more robustly than the competition.

# 2. Why use Julia?

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

Julia is a relatively new[1], productive, and fast programming language. It is evident in its pragmatic, productivity-focused design choices, pleasant syntax, rich ecosystem, thriving communities, and its ability to be both very general purpose and power cutting edge computing.

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

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

Large financial services organizations have already started realizing gains: BlackRock's Aladdin portfolio modeling, the Federal Reserve's economic simulations, and Aviva's Solvency II-compliant modeling[2]. The last of these has a great talk on YouTube by Aviva's Tim Thornham, which showcases an on-the-ground view of what difference the right choice of technology and programming language can make. Moving from their vendor-supplied modeling solution was **1000x faster, took 1/10 the amount of code, and was implemented 10x faster**.

The language is not just great for data science — but also modeling, ETL, visualizations, package control/version management, machine learning, string manipulation, web-backends, and many other use cases.

---

[1] 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.

[2] Aviva Case Study

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

> **Tip**
>
> **The two language problem** is a term describing processes and teams that separate "domain expertise" coding from "production" coding. This isn't always a "problem", but the "two language problem" describes the scenario where this arises not out of intention but out of the necessity of dealing with limitations of the programming languages used. The most common combination is when the domain experts utilize Python, while the quants or developers write C++. This arises because the productive, high level language hits a barrier in terms of speed, efficiency, and robustness. Then, as a necessary step to achieve the end goals of the business, the domain experts hand off the logic to be re-implemented into the lower level language. Not only does this effectively limit the architecture, it essentially defines a required staffing model which may introduce a lot of cost and redundancy in expertise. Julia solves this to a large extent, allowing for a high level, productive language to be very fast.
>
> A similar, related dichotomy is the **two culture problem** wherein domain experts (e.g. financial analysts) exist in a different sphere from developers. This manifests in many ways, such as restricting the tools that each group is permitted to use (e.g. Excel for domain experts, codebases and Git for developers). This is less of a technical problem and more of a social one. However, Julia is also one of the better languages in this regard, because much of the associated tooling is made as straightforward as possible (e.g. packaging, distribution, workflows, etc.). See Chapter 11 and Chapter 20 through Chapter 23 for more on this.

## 2.1. Julia and This Book

Julia is introduced insofar that a basic understanding is necessary to illustrate certain concepts. Julia is ideal in this context, because it is generally straightforward and concise, allowing the presented idea to have the spotlight (as opposed to language boiler plate or obtuse keywords and variables). The point of structuring the book like this is to allow us to introduce a wide variety of computer science concepts to the financial professional, *not* to introduce Julia as a programming language (there are many other resources which do that just fine).

This chapter seeks to motivate to the skeptical professional why we choose Julia for teaching and for work. Then, in Chapter 4 we provide an introduction to core language concepts and syntax. After this chapter,the content is focused on illustrating a number of key concepts, with Julia taking a secondary role, serving simply as a backdrop. It's not until Chapter 20 where Julia regains the spotlight and we discuss particulars which generally matter only to those heavily using Julia more vigorously.

## 2.2. Expressiveness and Syntax

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

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

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

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

### 2.2.1. Example 1: Retention Analysis

This is a really simple example relating `Cessions`, `Policys`, and `Lives` to do simple retention analysis. Retention is a measure of how much risk an insurance company holds on a policy after it's own reinsurance risk transfer (ceded amount of coverage are called "cessions").

First, let's define our data:

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

struct Policy
    face
    cessions
end

struct Cession
    ceded
end
```

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

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

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

And then what retention means for a `Life`:

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

```
retained (generic function with 2 methods)
```

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 **dispatch** based on what you give it (this is a more powerful, generalized version of method dispatch commonly used in object-oriented programming, see Chapter 6 for more).

Let's use the above code in practice then.

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

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

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

```
retained(pol_1)
```

```
400
```

```
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
```

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

### 2.2.2. Example 2: 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 2.1.: A comparison of random outcome generation in Julia, R, and Python.

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

## 2.3. The Speed

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

Earlier we described Aviva's Solvency II compliance modeling, which ran 1000x faster than the prior vendor solution 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.

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

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

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

```julia
using Distributions

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

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

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

S,K,τ,σ,r = 300, 250, 1, 0.15, 0.03

Call(S,K,τ,r,σ) # 58.81976813699322
```

```python
from scipy import stats
import math
```

```
def d1(S,K,τ,r,σ):
    return (math.log(S/K) + (r + σ**2/2) * τ) / (σ * math.sqrt(τ))

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

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

S = 300
K = 250
τ = 1
σ = 0.15
r = 0.03

Call(S,K,τ,r,σ) # 58.81976813699322

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

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

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

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

We find in Table 2.2 that despite the syntactic similarity, Julia is much faster than the other two.

Table 2.2.: Julia is nearly 20,000 times faster than Python, and two orders of magnitude faster than R.

| Language | Median (*nanoseconds*) | Mean (*nanoseconds*) | Relative Mean |
|---|---|---|---|
| Python | *not calculated by benchmarking library* | 817000.0 | 19926.0 |
| R | 3649.0 | 3855.2 | 92.7 |
| Julia | 41.0 | 41.6 | 1.0 |

### 2.3.1. Development Speed

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

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

> **Note**
>
> Sometimes you will see less of a speed difference; R and Python have already circumvented this and written much core code in low-level languages. This is 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? In technical terms, libraries like NumPy are not able to handle custom data types, and instead limit use to pre-built types within the library. In contrast, all types in Julia are effectively equal, even ones that you might create yourself.
> Julia packages you are using are almost always written in pure Julia: you can see what's going on, learn from them, or even contribute a package of your own!

## 2.4. More of Julia's benefits

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

- It's free and open-source
  - Very permissive licenses, facilitating the use in commercial environments (same with most packages)
- Large and growing set of available packages
- Write how you like because it's multi-paradigm: vector-izable (R), object-oriented (Python), functional (Lisp), or detail-oriented (C)
- Built-in package manager, documentation, and testing-library
- Jupyter Notebook support (it's in the name! **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 `PythonCall.jl` and RCall packages
- Error messages are helpful and tell you *what line* the error came from, not just the type of error
- Debugger functionality so you can step through your code line by line

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

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

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

## 2.5. Tradeoffs when Using Julia

### 2.5.1. Just-Ahead-of-Time Compilation

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 pre-compile 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.

### 2.5.2. Static Binaries

Static binaries are self contained executable programs which can run very specific bits of code. Simple programs which can compile down to small (in terms of size on disk) binaries which accomplish just their pre-programmed tasks. Another use case for this is to create shared libraries which could be called from other languages (this can already be done, but again requires bundling the runtime).

Julia's dynamic nature means that it needs to include the supporting infrastructure in order to run general code, similar to how running Python code needs to be bundled with the Python runtime.

Development is happening at the language level which would allow Julia to be compiled to a smaller, more static set of features for use in environments which are memory constrained and can't bundle a supporting runtime.

## 2.6. 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.

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

Another powerful aspect of the package ecosystem is that due to the language design, packages can be combined/extended in ways that are difficult for other common languages. This means that Julia packages often interoperable 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.

## 2.7. Tools in Your Toolbox

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 we have sufficiently piqued your interest and we can turn to introducing important concepts.

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!

Being a productive financial profession means 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.

# 3. Elements of Financial Modeling

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

## 3.1. In this Chapter

We explain what constitutes a financial model and what are common uses of a model. We explore the key attributes of models, discuss different modeling approaches and their trade-offs, and examine how to work effectively with data that feeds your models. We also explain what makes an adept practitioner.

## 3.2. What is a Model?

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

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

| Use | Description |
|---|---|
| **R**eason | To identify conditions and deduce logical implications. |
| **E**xplain | To provide (testable) explanations for empirical phenomena. |
| **D**esign | To choose features of institutions, policies, and rules. |
| **C**ommunicate | To relate knowledge and understandings. |
| **A**ct | To guide policy choices and strategic actions. |
| **P**redict | To make numerical and categorical predictions of future and unknown phenomena. |
| **E**xplore | To investigate possibilities and hypotheticals. |

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

Whether we are using this model of equity returns to estimate available retirement income or replicate an exotic option price, a key aspect of the model is the **assumptions** used therein. For the retirement income scenario we might *assume* a healthy eight percent return on stocks and conclude that such a return will be sufficient to retire at age 53. Alternatively, we may assume that future returns will follow a stochastic path with a certain distribution of volatility and drift. These two assumption sets will produce **output** - results from our model that must be inspected, questioned, and understood in the context of the "small world" of the model's mechanistic workings. Lastly, to be effective practitioners we must be able to contextualize the "small world" results within the "large world" that exists around us.

### 3.2.1. "Small world" vs "Large world"

The distinction between the modeled "small world" and real life "large world" modeling is fundamental to understanding model limitations:

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

Say that our model is one that discounts a fixed set of future cashflows using the US Treasury rate curve. If I run my model using current rates today, and then re-run my model tomorrow with the same future cashflows and the present value of those cashflows has increased by 5%, I may ask: "why has 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 discount rates have fallen substantially. The small world answer is that the inputs have changed which produced a mechanical change in the output. The large world answer may be that the Federal Reserve lowered the Federal Funds Rate to prevent the economy from entering a deflationary recession.

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

## 3.3. What is a *Financial* Model?

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

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

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

Examples of this:

- Projecting a retiree's savings through time (1), and determining how much they should be saving today for their retirement goal (2)
- Projecting the obligation of an exotic option across different potential paths (1), and determining the premium for that option (2)

Models are sometimes taken a step further, such as transforming the underlying **economic view** into an accounting or regulatory view (such as representing associated debits and credits, capital requirements, or associated intangible, capitalized balances).

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

### 3.3.1. Difference from Data Science

While practice and practitioners of financial modeling often substantially overlap, modeling in the sense used in this book is distinct from data science and statistics in all but the most general sense. To quickly define some terms as used in this book:

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

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

- Financial modeling generally incorporates a good deal of theory that assumes certain behaviors/relationships, instead of trying to infer those relationships from the data.
- Financial modeling generally contains more unique 'objects' than a statistical model. The latter may have derived numerical relationships between data, however a financial model would have things like the concept of a company or portfolio, or sets of individually identifiable assets, or distinct actors in a system.
- Financial modeling often involves a lot more simulation and hypothecating, while data science is focused on drawing conclusions from what data has already been observed.

Nonetheless, there is substantial overlap in practice. For example, the assumption in a financial model (volatility, economic conditions, etc.) may be derived statistically from observed data. Given the overlap in topics, statistical content is sometimes covered in this book (including a from-the-ground-up view of modern Bayesian approaches in Chapter 13).

## 3.4. Key Considerations for a Model

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

| Consideration | Description |
|---|---|
| Objective | Clearly define what the model aims to achieve. |
| Boundaries | Specify the limits and constraints of the model to avoid scope creep. |
| Variables | Identify and define all variables involved in the model. |
| Parameters | Include constants or coefficients that influence the variables. |
| Dependencies | Describe how variables interact with each other. |
| Relationships | Detail the connections between different components of the model. |
| Inputs | Specify the data or resources required for the model to function. |
| Outputs | Define what results or predictions the model produces. |
| Underlying Assumptions | Document any assumptions made during the model's development to clarify its limitations and validity. |
| Validation Criteria | Outline how the model's accuracy and reliability are tested. |

| Consideration | Description |
| --- | --- |
| Performance Metrics | Define the metrics used to evaluate the model's performance. |
| Scalability | Ensure the model can handle increased data or complexity if needed. |
| Adaptability | Allow for adjustments or updates as new information or requirements arise. |
| Documentation | Provide comprehensive documentation explaining how the model works, including algorithms, data sources, and methods. |
| Transparency | Make the model's workings understandable to stakeholders or users. |
| User Interface | Design an intuitive interface if the model is interactive. |
| Ease of Use | Ensure that the model is user-friendly and accessible to its intended audience. |
| Ethics | Address any ethical concerns related to the model's application or impact. |
| Regulations | Ensure compliance with relevant laws and regulations. |

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

## 3.5. Types of Models

Different modeling approaches come with their own sets of trade-offs. Common modeling approaches, and the inherent trade-offs, may include:

1. Statistical Models

Examples: Linear regression, logistic regression

Trade-offs: - Simplicity vs. Accuracy: Statistical models are often simpler and more interpretable but may not capture complex relationships as well as more sophisticated models. - Assumptions: These models typically rely on assumptions (e.g., linearity, normality) that may not always hold true, potentially affecting their accuracy.

2. Machine Learning Models

Examples: Decision trees, random forests, neural networks

Trade-offs: - Complexity vs. Interpretability: Machine learning models, especially deep learning models, can capture complex patterns but are often less interpretable. - Overfitting: More complex models risk overfitting the training data, requiring careful validation and tuning to ensure generalizability. - Data Requirements: These models often require large amounts of data to perform well, and their performance can degrade with limited or noisy data.

3. Simulation Models

Examples: Monte Carlo simulations, agent-based models

Trade-offs: - Accuracy vs. Computational Expense: Simulations can model complex systems and scenarios but can be computationally expensive and time-consuming. - Detail vs. Generalization: High-fidelity simulations can be very detailed but might be overkill for problems where a simpler model would suffice.

4. Theoretical Models

Examples: Economic models, physical models

Trade-offs: - Precision vs. Practicality: Theoretical models provide a foundational understanding but may rely on idealizations or simplifications that don't fully capture real-world complexities. - Applicability: They may be highly accurate in specific contexts but less applicable to broader or more variable situations.

5. Hybrid Models

Examples: Combining statistical and machine learning approaches, or combining theoretical and simulation models

Trade-offs: - Complexity vs. Versatility: Hybrid models aim to leverage the strengths of different approaches but can be complex to design and manage. - Integration Challenges: Combining different types of models may present challenges in integrating them effectively and ensuring consistency in their outputs.

6. Empirical Models

Examples: Time series forecasting, econometric models

Trade-offs: - Data Dependence vs. Predictive Power: Empirical models rely heavily on historical data and may not perform well in scenarios where patterns change or data is sparse. - Context Sensitivity: These models can be very accurate for the specific data they are trained on but might not generalize well to different contexts or conditions.

7. Probabilistic Models

Examples: Bayesian networks, probabilistic graphical models

Trade-offs: - Flexibility vs. Computational Complexity: Probabilistic models can handle uncertainty and complex relationships but often require more sophisticated computations and can be harder to implement and interpret.

8. Summary of Common Trade-offs:

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

Understanding these trade-offs helps in selecting the most appropriate modeling approach based on the specific needs of the problem at hand and the resources available.

## 3.6. How to work with data that feeds the models

Working effectively with data that feeds the models involves several key steps to ensure the data is suitable for modeling and that the model performs well. The steps may include:

1. Data Collection

- Source Identification: Identify and gather data from relevant and reliable sources.
- Data Acquisition: Use appropriate methods for collecting data, such as web scraping, surveys, sensors, or databases.

2. Data Exploration and Understanding

- Descriptive Statistics: Generate summary statistics (mean, median, standard deviation) to understand the data's central tendencies and variability.
- Visualization: Use plots (histograms, scatter plots, box plots) to visually inspect distributions and relationships between variables.
- Data Profiling: Assess data quality, completeness, and consistency.

3. Data Cleaning

- Handling Missing Values: Decide how to address missing data—options include imputation, interpolation, or removing incomplete records.
- Outlier Detection: Identify and handle outliers that may affect model performance. Outliers can be treated or removed based on their cause and impact.
- Data Transformation: Normalize or standardize data if needed, especially if the model requires data in a specific format or scale.

4. Feature Engineering

- Feature Selection: Choose relevant features that contribute to the model's predictive power. This may involve techniques like correlation analysis or feature importance scores.
- Feature Creation: Create new features from existing data that might provide additional insights or improve model performance. This could include polynomial features, interaction terms, or domain-specific transformations.

5. Data Splitting

- Training and Testing Sets: Split the data into training and testing sets (and sometimes a validation set) to evaluate model performance and avoid overfitting.
- Cross-Validation: Use cross-validation techniques (e.g., k-fold cross-validation) to assess model performance on different subsets of the data.

6. Data Preprocessing

- Scaling and Normalization: Apply techniques such as min-max scaling or z-score normalization to ensure features are on a similar scale.
- Encoding Categorical Variables: Convert categorical variables into numerical formats using methods like one-hot encoding or label encoding.
- Data Augmentation: For certain applications (e.g., image processing), augment the data to increase the size and variability of the dataset.

7. Data Integration

- Combining Datasets: If using multiple data sources, merge datasets carefully, ensuring consistent formats and handling discrepancies.
- Data Alignment: Ensure that the data from different sources are aligned in terms of timing, units, and granularity.

8. Data Storage and Management

- Data Warehousing: Store data in a structured format that facilitates easy access and management, such as databases or data lakes.
- Version Control: Track changes to datasets over time to maintain reproducibility and manage updates.

9. Ethical Considerations

- Bias and Fairness: Evaluate data for biases and ensure that the model does not perpetuate or amplify them.
- Privacy: Protect sensitive information and comply with data privacy regulations such as GDPR or CCPA.

10. Continuous Monitoring and Updating

- Performance Monitoring: Regularly assess the model's performance using new data and update the model as needed.
- Data Drift: Monitor for changes in data distribution over time (data drift) and retrain the model if necessary.

By following these steps, one can effectively manage data for your model, ensuring that it is clean, relevant, and capable of delivering accurate and reliable results.

## 3.7. 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.

### 3.7.1. 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[1].

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 3.1). Retrograde motion is the term used to describe the apparent, temporarily reversed motion of a planet as viewed from Earth when the Earth is overtaking the other planet in orbit around the sun. This was accurate enough to match the obersvational data that described the position of the planets in the sky.

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.*

[1] 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

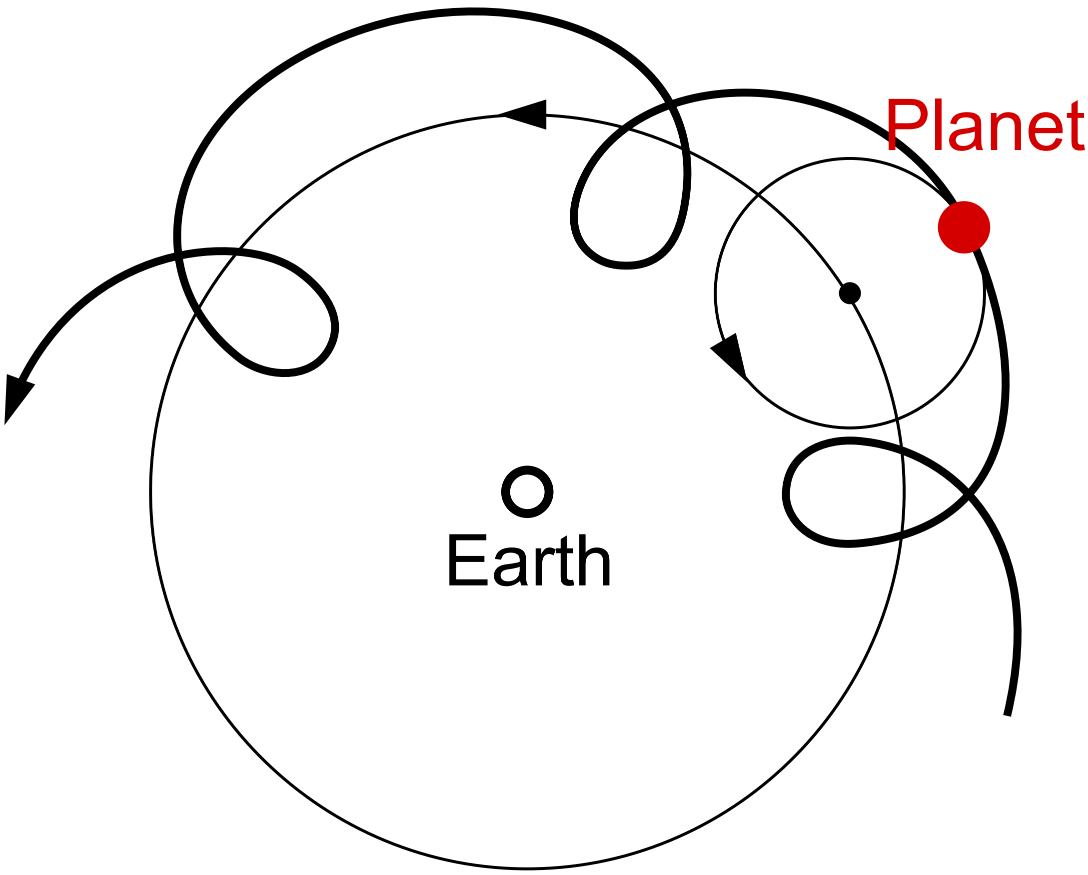



Figure 3.1.: In the Ptolemic solar model, the retrograde motion of the planets was explained by adding an epicycle to the circular orbit around the earth.

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:

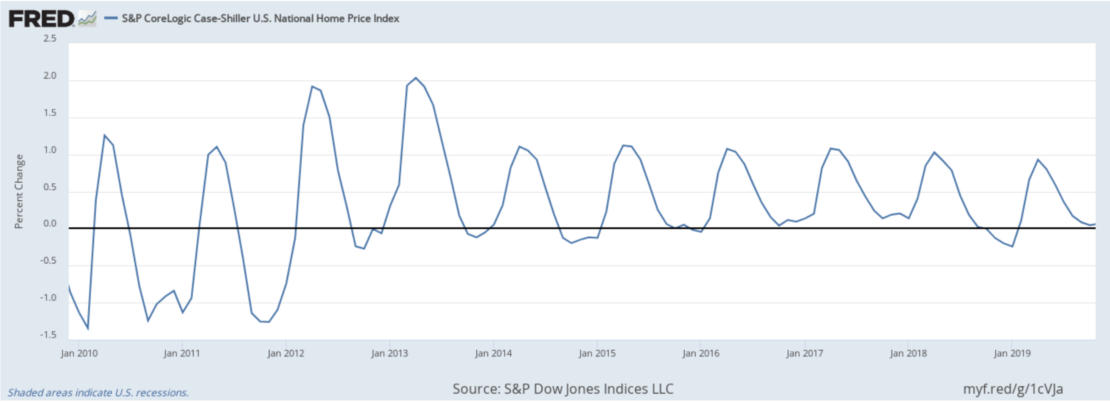
1. Predictive models need not have a scientific, causal structure to make accurate predictions.
2. It is difficult to capture the complete scientific inter-relationships of a system and much care and thought needs to be given in what aspects are included in our model.
3. We should look at, or seek out,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

### 3.7.2. Examples in the Financial Context

#### 3.7.2.1. 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*.




:::

:::

#### 3.7.2.2. 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[2]. 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.

## 3.8. What makes a good model?

The answer is: *it depends*.

### 3.8.1. Achieving original purpose

A model is built for a specific set of reasons and therefore we must evaluate a model in terms of achieving that goal. We should not critique a model if we want to use it outside of what it was intended to do. This includes: contents of output and required level of accuracy.

A model may have been created to for scenario analysis to value all assets in a portfolio to within half a percent of a more accurate, but much more computationally expensive model. If we try to add a never-before-seen asset class or use the model to order trades we may be extending the design scope of the original model.

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

### 3.8.2. 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 3.2 illustrates this concept and shows that if your goal is very high capability that you may need to expect to develop training materials and support the more complex model. On this view, a better model is one that is able to have a shorter amount of time and experience to achieve the same level of capability.

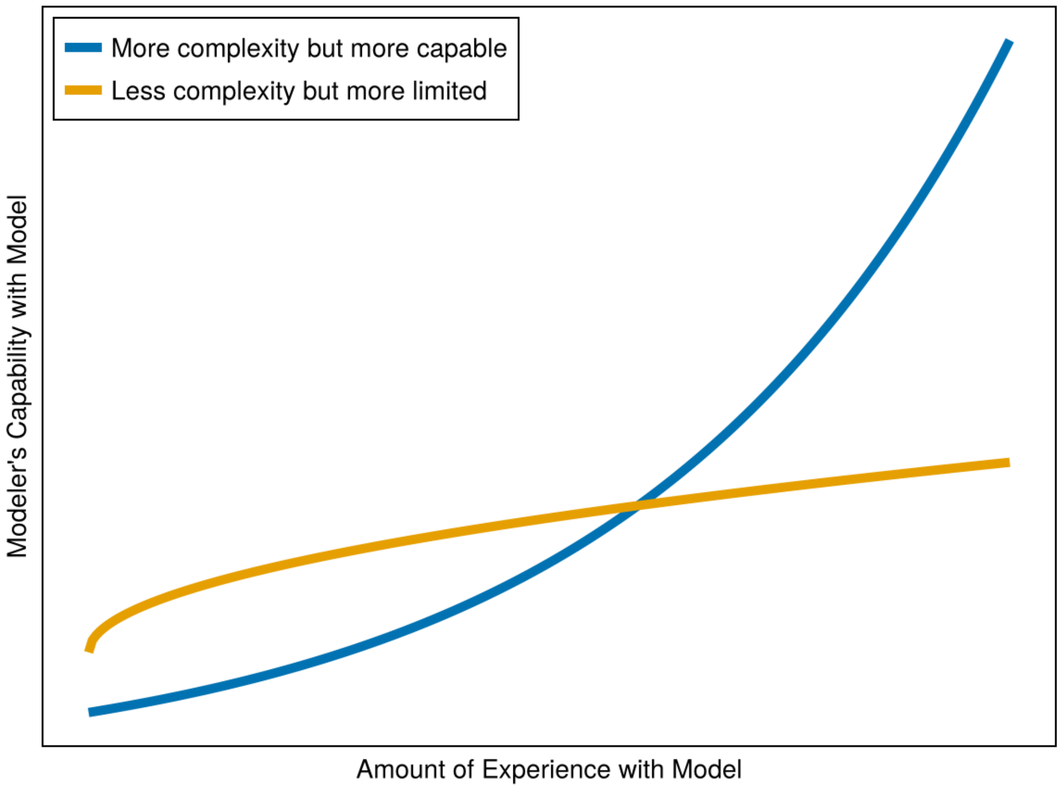



Figure 3.2.: Tradeoff between complexity and capability

### 3.8.3. 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.

### 3.8.4. 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.

### 3.8.5. Abstraction of Modeled Systems

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

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

Table 3.3.: 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 |
| | 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 3.3, 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.

## 3.9. 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?

### 3.9.1. 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.

### 3.9.2. 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[3] "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[4].

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.
- 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

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

[4] 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.

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 (and is training that this book covers). The last two are unique to our model and have "real world" meaning that our domain expert modeler would understand which analogizes very directly to the way we would suggest implementing the details of `value` or `Policy`. The benefit of this, again, is to provide tools and concepts which let us more easily develop model theory.

### 3.9.3. 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.

Therefore, with "Curiosity" we mean:

1. Continuous learning: Stay updated with the latest developments in finance, mathematics, and technology.
2. Interdisciplinary approach: Explore connections between finance and other fields for new insights.
3. eQuestioning assumptions: Regularly challenge and re-evaluate model assumptions.
4. Investigating anomalies: Pay close attention to unexpected results or outliers for potential insights.
5. Experimenting with new techniques: Try out innovative modeling methodologies or tools.
6. Seeking feedback: Engage in discussions with peers and experts to gain new perspectives.
7. Scenario analysis: Explore a wide range of possible scenarios to understand model behavior.
8. Root cause analysis: Dig deep to understand underlying causes when encountering issues.

By cultivating curiosity, modelers can drive innovation, uncover new insights, and continuously improve their models and understanding of financial systems.

### 3.9.4. 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. Regardless of the applicable standards, many of them share these aspects of the best modelers:

1. Methodological consistency: Align models with established financial and mathematical theories.

2. Robust validation: Implement thorough testing procedures, including backtesting and out-of-sample testing.
3. Sensitivity analysis: Conduct comprehensive analyses on key parameters to understand model limitations.
4. Data quality assurance: Implement rigorous data cleaning and validation processes.
5. Peer review: Engage in review processes to validate methodologies and results.
6. Regulatory compliance: Ensure models meet relevant regulatory requirements and standards.
7. Ethical considerations: Consider the ethical implications of model assumptions and outputs.
8. Continuous improvement: Regularly review and update models based on new information and methodologies.

At times, a bad model can be worse than no model at all. Through rigorous efforts, a minimum standard of quality can be obtained.

### 3.9.5. 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.

1. Precise language: Use well-defined terms and avoid ambiguity in communications.
2. Clear documentation: Provide comprehensive explanations of models, including assumptions and limitations.
3. Visual communication: Utilize diagrams and visualizations to explain complex concepts.
4. Consistent terminology: Establish and maintain a standardized vocabulary within the organization.
5. Audience-appropriate communication: Tailor explanations to the technical level of the audience.
6. Transparent assumptions: Clearly state and explain the rationale behind key model assumptions.
7. Regular review: Periodically update documentation to ensure ongoing clarity and accuracy.

By prioritizing clarity, modelers can reduce misunderstandings, improve collaboration, and increase the overall effectiveness of their work.

### 3.9.6. Humble

Irreducible & epistemic/reducible uncertainty are critical concepts for a modeler to understand and communicate:

1. Irreducible uncertainty: Also known as aleatoric uncertainty, this refers to the inherent randomness in a system that cannot be reduced by gathering more information.
    - Examples include: future market fluctuations, individual policyholder behavior, or natural disasters.
2. Epistemic/reducible uncertainty: This type of uncertainty stems from a lack of knowledge and can potentially be reduced through further study or data collection.
    - Examples include: parameter estimation errors, model specification errors, or data quality issues.

A humble modeler acknowledges these uncertainties and communicates them clearly to stakeholders. This avoids overconfidence in model predictions and keeps one open to new information and alternative perspectives. By maintaining a humble attitude, modelers can build trust with stakeholders and make more informed decisions based on model outputs.

### 3.9.7. Architecture

Any sufficiently complex project benefits from architectural thinking. Data should be separate from the logic and the model itself should not contain any substantial datum itself - instead dynamically load data from appropriate data stores and leave the "model" as the implementation of data *types* and algorithms.

1. Modular design: Break complex models into reusable, independent components.
2. Separation of concerns: Keep data, logic, and presentation layers distinct for better maintainability.
3. Scalability: Design models to handle increasing data volumes and complexity.
4. Maintainability: Implement version control and clear documentation practices.
5. Flexibility: Create designs that allow for easy updates and modifications.
6. Performance optimization: Use efficient data structures and algorithms to enhance model speed.

7. Error handling and logging: Implement robust systems for debugging and auditing.
8. Security: Ensure proper data protection and regulatory compliance.

### 3.9.8. Planning

When tackling a large problem, it helps to have a well-structured planning process. Specific to building a financial model, one should take steps that include:

1. Defining clear objectives: Understand the purpose of the model and what questions it needs to answer.
2. Scope definition: Determine the boundaries of the model, including what to include and what to exclude.
3. Data assessment: Identify required data sources, assess data quality, and plan for data preparation.
4. Methodology selection: Choose appropriate modeling techniques based on the problem and available data.
5. Resource allocation: Estimate time and resources needed for model development, testing, and implementation.
6. Stakeholder engagement: Identify key stakeholders and plan for their involvement throughout the modeling process.
7. Risk assessment: Anticipate potential challenges and develop mitigation strategies.
8. Timeline development: Create a realistic timeline with key milestones and deliverables.
9. Documentation strategy: Plan for comprehensive documentation of assumptions, methodologies, and limitations.
10. Validation and testing approach: Outline strategies for model validation and testing to ensure reliability.
11. Implementation and maintenance plan: Consider how the model will be deployed and maintained over time.

Time invested at the planning stage often pays dividends through shorter model build times, fewer errors, and clarity from stakeholders at the start of the project. Additionally, it's often easier to make changes to a well-planned project halfway through since the necessary accomodations are more clearly defined.

### 3.9.9. 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. This includes both soft and hard skills, such as those in **?@tbl-toolset**

| Category | Examples |
|---|---|
| Diverse Modeling Techniques | • Statistical methods (e.g. regression, time series analysis, machine learning)<br>• Optimization techniques (e.g. linear, non-linear, black-box)<br>• Simulation methods (e.g. monte-carlo, agent-based, seriatim) |
| Software Profeciency | • Programming languages<br>• Database and data handling<br>• Proprietary tools (e.g. Bloomberg) |
| Financial Theory | • Asset pricing<br>• Portfolio theory<br>• Risk Managment frameworks |
| Quantitative techniques | • Numerical methods and algorithms<br>• Bayesian inference<br>• Stochastic calculus |
| Soft Skills | • Verbal and written communication<br>• Stakeholder engagement<br>• Project Management |

::: A variety of skills have their place in the proficient financial modeler's toolbelt. {#tbl-toolset}

## 3.10. Conclusion

The art and science of financial modeling require a unique blend of skills, knowledge, and personal qualities. A proficient modeler combines domain expertise, theoretical understanding, and practical skills with a curious and rigorous mindset. They leverage a diverse toolset, employ sound architectural principles, and communicate with clarity. The ability to navigate the complexities of financial systems while maintaining humility in the face of irreducible uncertainties is paramount.

As the financial world continues to evolve, so too must the modeler's approach. By cultivating these attributes and continuously refining their craft, financial modelers can create more robust, insightful, and valuable models that drive informed decision-making in an increasingly complex economic landscape. The journey of a financial modeler is one of perpetual learning and adaptation, where each challenge presents an opportunity for growth and innovation.

# Part II.

# Foundations: Programming and Abstractions

> Out of intense complexities intense simplicities emerge. - Winston Churchill (1923)

The next several chapters build up essential concepts that enable us to create sophisticated financial models while maintaining clarity and manageability. We'll start with core programming building blocks - the vocabulary and grammar of communicating with computers. From there, we'll explore different approaches to breaking down complex problems into simpler components through abstraction: functions, types, programming paradigms, and more.

Abstraction is a form of selective ignorance - focusing attention on what matters for a particular purpose while hiding irrelevant details. Just as financial statements abstract away the details of individual transactions to reveal the bigger picture, we'll see how thoughtful abstraction in programming allows us to manage complexity by working at different levels of detail.

The goal is not to make you a computer scientist, but rather to equip you with the mental models and practical techniques needed to effectively leverage programming in your financial work. By understanding these foundations, you'll be better equipped to design clean, maintainable models that can evolve with your needs.

Think of this section as building your modeling toolkit, one concept at a time. We'll introduce ideas progressively, with plenty of concrete examples to ground the theory in practical application. The concepts build on each other, so take time to ensure your understanding before moving forward. Let's begin with the fundamental elements of programming.

# 4. Elements of Programming

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

## 4.1. In this section

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

> **On Your First Readthrough**
>
> This chapter is intended to be an introductory reference for most of the basic building blocks for which we will build abstractions on top of in chapters that follow. We want this chapter to essentially be an easy and mildly opinionated stepping-stone on your journey.
>
> At some point, you will likely find yourself seeking more precise or thorough documentation and will begin directly searching or reading the documentation of a language or library itself. However, it may be intimidating or frustrating reading reference documentation due to the density and terminology - let this chapter (and book writ large) be a bridge for you.
>
> If reading this book in a linear fashion and new to programming, we suggest skipping the following sections and returning when encountering the concept or term later in the book:
>
> - Section 4.4.4 through Section 4.4.9 which covers advanced and custom data types
> - After Section 4.5.3 which deals with advanced function usage and program organization via scope

> **Caution**
>
> This introductory chapter is intended to provide a survey of the important concepts and building blocks, not to be a complete reference. For full details on available functions, more complete definitions, and a more complete tour of all language

features, see the Manual at docs.julialang.org.

## 4.2. 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[1].

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

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

In this light, the financial modeling that we do can be considered a type of computer program. It takes as input abstract information (data), performs calculations (an algorithm), and returns new data as an output. 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. However, the programming fundamentals are there: 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.
- In the following chapters about abstraction, we will step back and discuss higher level concepts: the "schools of thought" around organizing the relationship between data and functions (functional versus object-oriented programming), design patterns, computational complexity, and compilation.

[1] 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 ($3^2 + 4^2 = 5^2$).

> **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 the first part of Chapter 20 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...
> ```

## 4.3. Expressions and Control Flow

### 4.3.1. Assignment and Variables

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

```
feet_per_yard = 3
yards_per_mile = 1760

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

```
0.5681818181818182
```

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

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

> **Tip**
>
> We will dive a little bit deeper into variables and assignment in Section 4.3.4, distinguishing between assignment and references.

### 4.3.2. Expressions

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

```julia
1 + 2
```

```
3
```

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

#### 4.3.2.1. Compound Expression

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

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

a, b, c
```

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

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

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

```
3
```

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

#### 4.3.2.2. Conditional Expressions

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

```
true
```

> **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")
```

### 4.3.3. Equality

The "Ship of Theseus[2]" 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:

[2]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.

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

```
(true, false)
```

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

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

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

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

```
s == t
```

```
false
```

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

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

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

```
true
```

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

### 4.3.4. 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
```

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

```
const PHI =  π * 2 # <1>
```

① Capitalizing constant variables is a convention in Julia.

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

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

### 4.3.5. Loops

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

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

```
# 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 increment `i` by 1. `i` is defined outside the scope of the `while` loop (see Section 4.6).

```
# 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.

### 4.3.6. 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!*

## 4.4. 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`[3].

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[4].

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

[4]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.

### 4.4.1. Numbers

Numbers are usually grouped into two categories: **integers** and **floating-point**[5] numbers. Integers are like the mathematical set of integers while floating-point is a way of representing decimal numbers. Both have some limitations since computers can only natively represent a finite set of numbers due to the hardware (more on this in Chapter 8). Here are three integers that are input into the **REPL** (Read-Eval-Print-Loop)[6] and the result is **printed** below the input:

```
2
```

```
2
```

```
423
```

```
423
```

```
1929234
```

```
1929234
```

And three floating-point numbers:

```
0.2
```

```
0.2
```

```
-23.3421
```

```
-23.3421
```

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

```
14000.0
```

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

[6]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.

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[7], while on a 32-bit system `0.2` would be interpreted as a `Float32`. Given that there are a finite amount of bits attempting to represent a continuous, infinite set of numbers means that some numbers are not able to be represented with perfect precision. For example, if we ask for `0.2`, the closest representations in 64 and 32 bit are:

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

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 4.3.4.
② `big(x)` is a arbitrary precision floating point number and by default prints the full precision that was embedded in our variable `x`, which was originally `Float64`.

```
0.200000000000000011102230246251565404236316680908203125
```

> **Note**
>
> Note the difference in what printed between the last example and when we input `0.2` earlier in the chapter. The former had the same (not-exactly equal to 0.2) *value*, but it printed an abbreviated set of digits as a nicety for the user, who usually doesn't want to look at floating point numbers with their full machine precision. The system has the full precision (`0.20...3125`) but is truncating the output.
> In the last example, we've converted the normal `Float64` to a `BigFloat` which will not truncate the output when printing.

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

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

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

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

**Floating Point and Excel**

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

**Currencies and Decimals**

Note that floating point numbers should **not** be used in storing transaction records! The intricacies of floating point math described above to not lend itself to accurate record-keeping, since with operations like `0.11 + 0.12` don't precisely equal `0.23`

```
BigFloat(0.11 + 0.12)
```

```
0.2299999999999999982236431605997495353221893310546875
```

As you can see, if we were adding US Dollar cents here, we would have destroyed a fraction of a penny. Do that for millions of transactions in a day and you have a problem!
Generally, when doing *modeling* or even creating a *valuation model*, it's okay to use floating point math. As an example, if you are trying to determine the value of an exotic option, your model is likely just fine outputting a value like `101.987087`. If you go and sell this option, you'll have to settle for either `101.98` or `101.99` when booking it. In most contexts this imprecision is likely okay!

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

### 4.4.2. 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.

**?@fig-julia-numeric-types** shows the type hierarchy for most built-in Julia number types.

TODO: Once Quarto Issue #10961 is resolved, render the mermaid diagram.

```
%%| label: fig-julia-numeric-types
%%| fig-cap: "Numeric Type Hierarchy in Julia. Leafs of the tree are concrete types."
%%| fig-width: 6.5
graph TD
    Number --> Real
    Number --> Complex

    Real --> Integer
    Real --> AbstractFloat
    Real --> Rational
    Real --> Irrational

    Integer --> Signed
    Integer --> Unsigned

    Signed --> Int8
    Signed --> Int16
    Signed --> Int32
    Signed --> Int64
    Signed --> Int128
    Signed --> BigInt

    Unsigned --> UInt8
    Unsigned --> UInt16
    Unsigned --> UInt32
```

```
Unsigned --> UInt64
Unsigned --> UInt128

AbstractFloat --> Float16
AbstractFloat --> Float32
AbstractFloat --> Float64
AbstractFloat --> BigFloat
```

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.

### 4.4.3. Collections

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

#### 4.4.3.1. Arrays

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

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

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

And a floating point array:

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

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

Note the above two arrays are different types of arrays. The first is `Vector{Int64}` and the second is `Vector{Float64}`. These are arrays of concrete types and so Julia will know that each element of an array is the same amount of bits which will enable more efficient computations. With the following set of mixed numbers, Julia will **promote** the integers to floating point since the integers can be accurately represented[8] in floating point.

```
[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] =
 0.636827   0.598882   0.368017
 0.0987708  0.0986006  0.225914
 0.637245   0.0100418  0.957667

[:, :, 2] =
 0.521364  0.900446  0.439745
 0.872272  0.263656  0.667518
 0.959902  0.621212  0.550598

[:, :, 3] =
 0.743887   0.790807  0.662174
 0.0939974  0.653103  0.0481541
 0.754063   0.273387  0.625755
```

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

[8]Accurate only to a limited precision, as described in Section 4.4.1.

```
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 convenient way to create arrays in Julia.

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

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

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

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

#### 4.4.3.2. Array indexing

Array elements are accessed with the integer position, starting at `1` for the first element[9] [10]:

[9]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.

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

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

```
20
```

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

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

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

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

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

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

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

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

```
9-element Vector{Int64}:
 10
 -1
 30
 40
 50
  5
  1
  2
  3
```

#### 4.4.3.3. Array Alignment

When you have an MxN matrix (M rows, N columns), a choice must be made as to which elements are next to each other in memory. Typical math convention and fundamental computer linear algebra libraries (dating back decades!) are column major and Julia follows that legacy. **Column major** means that elements going down the rows of a column are stored next to each other in memory. This is important to know so that (1) you remember that vectors are treated like a column vector when working with arrays (that is: a N element 1D vector is like a Nx1 matrix), and (2) when iterating through an array, it will be faster for the computer to access elements next to each other column-wise. A 10x10 matrix is actually stored in memory as 100 elements coming in order, one after another in single file.

This 3x4 matrix is stored with the elements of columns next to each other, which we can see with `vec`:

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

```
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
```

#### 4.4.3.4. Ranges

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

We don't have to actually store all of these numbers on the computer somewhere as in an `Array`. Instead, this is an object that *represents* the ordered set of numbers. So for example, we can sum up 1 through the number of atoms on the earth instantaneously:

```julia
sum(1:big(100_000_000_000_000_000_000_000_000_000_000_000_000_000_000_000_000))
```

```
5000000000000000000000000000000000000000000000000050000000000000000000000000000000000000000000
```

This is possible due to two things:

1. not needing to actually store that many numbers in memory, and
2. Julia being smart enough to apply the triangular number formula[11] when `sum` is given a consecutive range.

There are more general ways to construct ranges:

Step by another number instead of the default 1:

```julia
1:2:7
```

```
1:2:7
```

Specify the number of values within the range, inclusive of the first number[12]:

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

```
0.0:0.5:10.0
```

#### 4.4.3.5. Characters, Strings, and Symbols

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

```julia
'a'
```

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

[11]The triangular numbers (sum of integers from 1 to $n$) are:

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

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

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[13] 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.

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

#### 4.4.3.6. Tuples

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

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

```
(3, 4)
```

Tuple's values can be accessed like arrays:

```
p1[1]
```

```
3
```

Tuples fill a middle ground between scalar types and arrays in more ways 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[14], meaning that a lot of times they can be faster than arrays.

##### 4.4.3.6.1. Named Tuples

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

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

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

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

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
```

#### 4.4.3.7. Dictionaries

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

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

```
d = Dict(
    "Joelle" => 10,
    "Monica" => 84,
    "Zaylee" => 39,
)
```

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

Then we can look up an age given a name:

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

```
39
```

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

### 4.4.4. Parametric Types

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

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

```
typeof(p1)
```

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

`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.

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

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

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

Table 4.1.: 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` |

### 4.4.5. Types for things not there

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

`missing` is to represent 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[15]. This is analogous to `Float64` being a type and `2.0` being a value.

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

### 4.4.6. Union Types

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

### 4.4.7. Creating User Defined Types

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

```julia
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:

```julia
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!

```julia
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 Ju-

lia 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 sophisticated about the program design or need to extract more performance from the code.

The above `structs` that we have defined are examples of **concrete types** types which hold data. **Abstract types** don't directly hold data themselves but are used to define a hierarchy of types which we will later exploit (Chapter 7) 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>
```

### 4.4.8. Mutable structs

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

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

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

### 4.4.9. Constructors

**Constructors** are functions that return a data type (functions will be covered more generally later in the chapter). When we declare a `struct`, an implicit function is defined that takes a tuple of arguments and returns the data type that was declared. In the following example, after we define `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[56]:2
 [2] MyDate(year, month, day)
     @ In[56]: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
```

## 4.5. Functions

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

### 4.5.1. Special Operators

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

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

```
!true, -5
```

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

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

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

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

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

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

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

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

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

**Methods** are specific compiled versions of the function for specific types. This is important because at a hardware level, operations for different types (e.g. integers versus floating point) differ considerably. By optimizing for the specific types Julia is able to achieve nearly ideal performance without the same sacrifices of other dynamic languages. We will develop more with respect to methods when we talk about dispatch in Chapter 7.

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

```
methods(factorial)
```

```
# 7 methods for generic function "factorial" from Base:
 [1] factorial(n::UInt128)
     @ combinatorics.jl:26
 [2] factorial(n::Int128)
     @ combinatorics.jl:25
 [3] factorial(x::BigFloat)
     @ Base.MPFR mpfr.jl:769
 [4] factorial(n::BigInt)
     @ Base.GMP gmp.jl:703
 [5] factorial(n::Union{Int16, Int32, Int8, UInt16, UInt32, UInt8})
     @ combinatorics.jl:33
 [6] factorial(n::Union{Int64, UInt64})
     @ combinatorics.jl:27
 [7] factorial(n::Integer)
     @ intfuncs.jl:1135
```

### 4.5.2. Defining Functions

Funcitons more generally are defined like so:

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

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

```
function value_range(collection)

    hi = maximum(collection)
```

```
    lo = minimum(collection)
    return hi - lo                                    ①
end
```

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

### 4.5.3. Defining Methods on Types

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

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

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

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

> **Note**
>
> An alternate, simpler function syntax for `distance` would be:
>
> ```
> distance(point) = sqrt(point.x^2 + point.y^2)
> ```

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

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

```
1.4142135623730951
```

The above value should be $\sqrt(3)$, or approximately $1.73205$.

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

```julia
"""
    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:

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

```
1.7320508075688772
```

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

In Chapter 7 we will see how dispatch (single and multiple) can provide very nice abstractions to simplify the design of a model.

**Docstrings (Documentation Strings)**

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

**Defining Methods for Parametric Types**

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

### 4.5.4. Keyword Arguments

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

### 4.5.5. Default Arguments

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

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

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

### 4.5.6. Anonymous Functions

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

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

① The `x -> x^2` is the anonymous function in this example.

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

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

### 4.5.7. First Class Nature

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

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

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

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

# obs meaning "observations"
function loss(modeled_obs,
    actual_obs,
    loss_function                                                    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)
```

### 4.5.8. Broadcasting

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

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

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

#### 4.5.8.1. Broadcasting Rules

What happens if one of the collections is not the same size as the others? When broadcasting, singleton dimensions (i.e. the 1 in 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)

```
#| error: true
[1, 2, 3] .+ [1 2 3; 4 5 6]
```

This 2x4 isn't compatible with the 2x3 (M and P match, but N nor Q is 1):

```
#| error: true
[1 2; 3 4] .+ [1 2 3; 4 5 6]
```

#### 4.5.8.2. Not Broadcasting

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

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

```
true
```

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

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

```
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
```

### 4.5.9. Passing by Sharing

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

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

function double!(v)
    for i in eachindex(v)
        v[1] = 2 * v[i]
    end
end

double!(v)

v
```

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

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

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

## 4.6. Scope

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

```
i = 1          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 in `Main`
Suggestion: check for spelling errors or missing imports.
UndefVarError: `j` not defined in `Main`
Suggestion: check for spelling errors or missing imports.
```

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

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

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

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

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

Here is an example with functions:

```
x = 2
base = 10
foo() = base^x                  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).

### 4.6.1. Modules and Namespaces

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

```
module Shape                                          ①

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

function area(t::Triangle)                            ②
    return 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`).

# 5. Functional Abstractions

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

## 5.1. In this section

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

## 5.2. Introduction

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

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

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

Where

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

```
cf_bond = [10, 10, 10, 10, 110];
rate = [0.05, 0.06, 0.05, 0.04, 0.05];                                    ①
```

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

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

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

## 5.3. Imperative Style

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

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

```
let
    pv = 0.0                                   ①
    discount = 1.0

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

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

```
121.48888490821489
```

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

### 5.3.1. Iterators

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

1. We are making the *assumption* that the indices of the vectors start with one, when in reality Julia arrays *can* be defined to start at `0` or another arbitrary index.
2. We manually perform the lookup of the values within each iteration.

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

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

```
eachindex(cf_bond)
```

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

> **Lazy Programming**
>
> The result, Base.OneTo(5) is a **lazy** object which represents a collection that does not get fully instantiated until asked to (which may not actually be necessary). Many (most?) iterators are lazy but we can interact with them without fully instantiating the data that they represent. **Instantiating** means fully loading the values into memory.
> An analogy is that we can write the "set of all numbers from 1 to 100" without writing out each of the 100 numbers, but we are referring to the same thing.
> An example of oprating on a lazy iterator, is that we could find the largest index:
>
> ```
> maximum(eachindex(cf_bond))
> ```
>
> ```
> 5
> ```
>
> The point is if we have an object that *represents* a set, we need not actually enumerate each element of the set to interact with it.
> We can fully instantiate an iterator with `collect`
>
> ```
> collect(eachindex(cf_bond))
> ```
>
> ```
> 5-element Vector{Int64}:
>  1
>  2
> ```

```
3
4
5
```

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

And when used in context:

```
let
    pv = 0.0
    discount = 1.0

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

```
121.48888490821489
```

Here Julia gave us the index associated with the bond cashflows, but we are still looking up the values (why not just ask for the values instead of their index?) as well as assuming that the indices are the same for the discount rates.

We can get the value and the associated index with `enumerate`:

```
collect(enumerate(cf_bond))
```

```
5-element Vector{Tuple{Int64, Int64}}:
 (1, 10)
 (2, 10)
 (3, 10)
 (4, 10)
 (5, 110)
```

This would allow us to skip the step of needing to look up the bond's cashflows. However, we can go even further by just asking for value associated with both collections. With `zip` (named because it's sort of like zipping up two collections together), we get an iterator that provides the values of the underlying collections:

```
collect(zip(cf_bond, rate))
```

```
5-element Vector{Tuple{Int64, Float64}}:
 (10, 0.05)
 (10, 0.06)
 (10, 0.05)
 (10, 0.04)
 (110, 0.05)
```

This provides the simplest implementation of the imperative approaches:

```
let
    pv = 0.0
    discount = 1.0

    for (cf, r) in zip(cf_bond, rate)
        discount = discount / (1 + r)
        pv = pv + discount * cf
    end
    pv
end
```

```
121.48888490821489
```

The primary downsides to iterative approaches to algorithms are:

1. Needing to keep track of state is fine in simple cases, but can quickly become difficult to reason about and error prone as the number and complexity of variables grows.
2. Program flow is explicitly stated, leaving fewer places that the compiler can automatically optimize or parallelize.

> **Tip**
>
> In the imperative style, we mentioned needing to explicitly handle program state. In general, it's advisable to minimize as many temporary state variables as possible - more mutability tends to produce more complex, difficult to maintain code. An example of maintaining state in the examples above is keeping track of the current index as well as interim `pv` and `discount` variables.
> Avoiding modifying values is often avoidable by restructuring the logic, using functional techniques, or finding the right abstractions. However, sometimes for performance reasons, clarity, or expediency you may find modifying state to be the

> preferred option and that's okay.

## 5.4. Functional Techniques and Terminology

**Functional** programming is a paradigm which attempts to minimize state via composing functions together.

Table 5.1 introduces a set of core functional methods to familiarize yourself with. Note that anonymous functions (Section 4.5.6) are used frequently to define intermediary steps.

Table 5.1.: Important Functional Methods.

| Function | Desciption | Example |
|---|---|---|
| `map(f,v)` | Apply function `f` to each element of the collection `v`. | `map(`<br>`    x->x^2,`<br>`    [1,3,5]`<br>`) # [1,9,25]` |
| `reduce(op,v)` | Apply binary `op` to pairs of values, reducing the dimension of the collection `v`.<br><br>Has a couple of important, optional keyword arguments to note (which also apply to other variants of `reduce` below):<br><br>• `init` defines the identity element (e.g. the initial value of `+` and `*` is `0` and `1` respectively)<br>• `dims` defines which dimension to reduce across (if the dimension of `v` is more than one). | `reduce(`<br>`    *,`<br>`    [1,3,5]`<br>`) # 15` |

| Function | Desciption | Example |
|---|---|---|
| `mapreduce(op,f,v)` | Maps `f` over collection `v` and returns a reduced result using `op`. | `mapreduce(`<br>`    *,`<br>`    x->x^2,`<br>`    [1,3,5]`<br>`) # 35` |
| `foldl(op,v)foldr(op,v)` | Like `reduce`, but applies `op` from left to right (`foldl`) or right to left (`foldr`). Also has `mapfoldl` and `mapfoldr` versions. | `foldl(`<br>`    *,`<br>`    [1,3,5]`<br>`) # 15` |
| `accumulate(op,v)` | Apply `op` along `v` , creating a vector with the cumulative result. | `accumulate(`<br>`    +,`<br>`    [1,3,5]`<br>`) # [1, 4, 9]` |
| `filter(f,v)` | Apply `f` along `v` and return a copy of `v` with elements where `f` is true | `filter(`<br>`    >=(3),`<br>`    [1,3,5]`<br>`) # [3, 5]` |

This paradigm is very powerful in a few ways:

1. It provides a language for talking about what a computation is doing. Instead of “looping over a collection called `portfolio` and calling a `value` function” we can more concisely refer to this as `mapreduce(value,portfolio)`.
2. Often times you are forced to think about the design of the program more deeply, recognizing the core calculations and data used within the model.
3. The compiler is free to apply more optimizations. For example, with `reduce`, the compiler could drive the calculation in any order since the operation is associative.
4. The lack of mutable state.

Let’s build a version of the present value calculation using the functional building blocks described above. We will work up to it by discussing the core functional programming building blocks, culminating in combining `mapreduce` and `accumulate` to do the bond valuation.

### 5.4.1. `map`

**`map`** is so named for the mathematical concept of mapping an input to an output. Here, it’s effectively the same thing. We take a collection and use the given function to calculate an output. The size of the output equals the size of the input.

First, we will use `map` to compute the one-period discount factors:

```
map(x -> 1 / (1 + x), rate)
```

```
5-element Vector{Float64}:
 0.9523809523809523
 0.9433962264150942
 0.9523809523809523
 0.9615384615384615
 0.9523809523809523
```

`map` transforms the `rate` collection by applying the anonymous function `x -> 1 / (1 + x)`, which is the single period discount factor. This operation is conveyed visually in Figure 5.1.

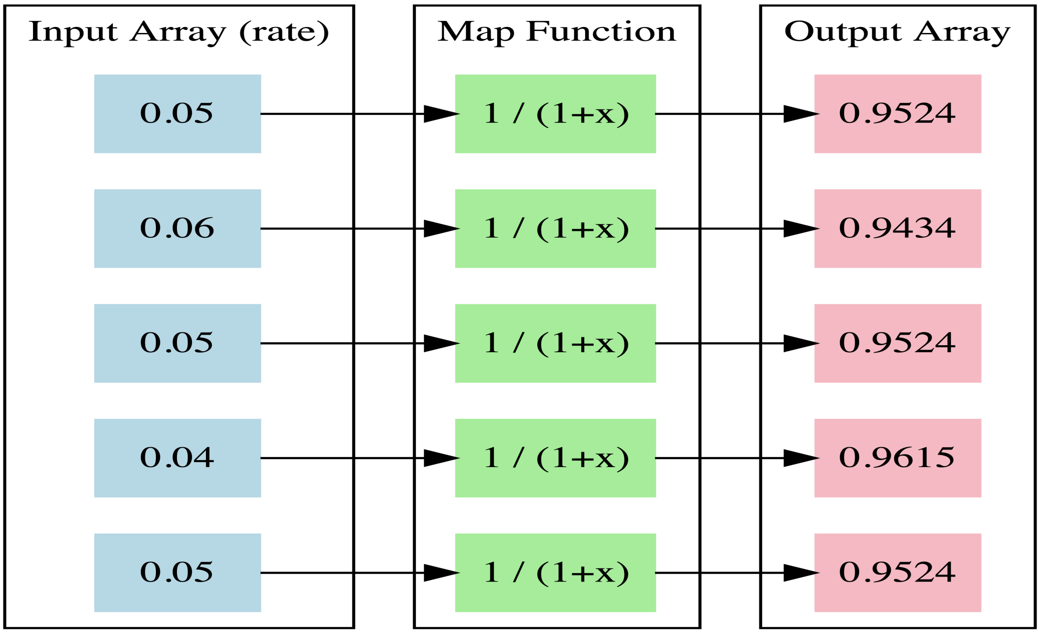



Figure 5.1.: A diagram showing that `map` creates a new collection mirroring the old one, after applying the given function to each element in the original collection.

**Tip**

`map` is an absolute workhorse of a function and the authors recommend using it liberally within your code. We find ourselves using `map` frequently, usually avoiding defining an explicit loop (unless we are modifying some existing collection). `map` would likely be a better tool for a loop like this:

```
output = []
```

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

Instead, `map` simplifies this to:

```julia
map(collection) do x
    # ... do stuff
end
```

Not only does this have the advantage of being clearer, more concise, and less work, it also let's Julia infer the output type of your computation so you don't have to worry about the type of `output`.

### 5.4.2. `accumulate`

`accumulate` takes an operation and a collection and returns a collection where each element is the cumulative result of applying the operation from the first element to the current one. For example, to calculate the cumulative product of the one-period discount factors:

```julia
accumulate(*, map(x -> 1 / (1 + x), rate))
```

```
5-element Vector{Float64}:
 0.9523809523809523
 0.898472596585804
 0.8556881872245752
 0.822777103100553
 0.7835972410481457
```

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

**Note**

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

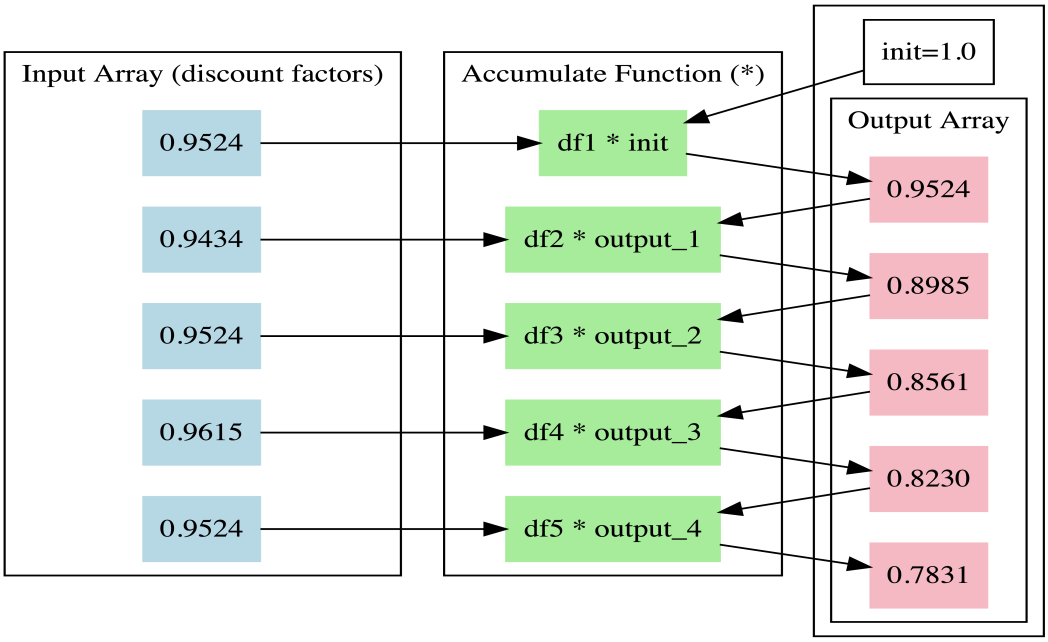



Figure 5.2.: A diagram showing that `accumulate` creates a new collection where each element is the cumulative result of applying the given operation to all previous elements.

Another example is string concatenation. In Julia, two strings are concatenated with `*` (like in mathematics, $a * b$ is also written as $ab$). The identity element for strings where the binary operation $\odot = *$ is `""`. For example:

```
accumulate(*, ["a", "b", "c"], init="")
```

```
3-element Vector{String}:
 "a"
 "ab"
 "abc"
```

*This is a taste of a branch of mathematics known as Category Theory, a very rich subject but largely beyond the immediate scope of this book. The category theoretical term for sets of things that work with the binary operator and identity elements as described above is a **monoid**. There will not be a quiz on this trivia.*

### 5.4.3. `reduce`

`reduce` takes an operation and a collection and applies the operation repeatedly to pairs of elements until there is only a single value left.

For example, we start with the calculation of the vector of discounted cashflows

```
dfs = accumulate(*, map(x -> 1 / (1 + x), rate))
discounted_cfs = map(*, cf_bond, dfs)
```

```
5-element Vector{Float64}:
  9.523809523809524
  8.98472596585804
  8.556881872245752
  8.22777103100553
 86.19569651529602
```

Then we can sum them with reduce:

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

```
121.48888490821487
```

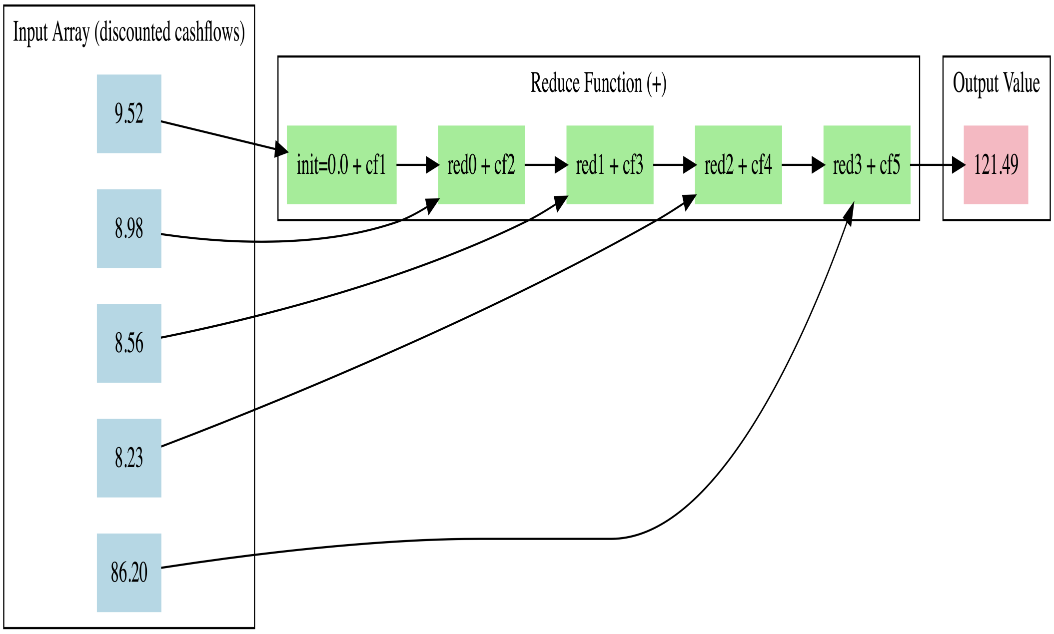



Figure 5.3.: A diagram showing how `reduce` applies the given operation to pairs of elements, ultimately reducing the collection to a single value.

### 5.4.4. `mapreduce`

We can combine `map`, `accumulate` and `reduce` to concisely calculate the present value in a functional style. This calculates the discount factors, applies them to the cashflows with map, and sums the result with a reduction:

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

(1) Multiplicatively accumulate a discount factor derived from the rate vector.

```
121.48888490821487
```

Contrast this example with the earlier imperative styles:

- This functional approach is more concise.
- The functions used are more descriptive and obvious (once familiar with them, of course!).
- There is no state that the user/programmer keeps track of.
- The compiler is able to potentially optimize the code, as it can deduce that certain operations are associative.

This completes the example of using a functional approach to determine the present value of bond cashflows.

### 5.4.5. `filter`

For completeness, we will also cover `filter` even though it's not necessary for the bond cashflow example.

`filter` does what you might think - filter a collection based on some criterion that can be determined as true or false.

For example filtering out even numbers using the `isodd` function:

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

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

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

```
filter(x -> ~(x == 5), 1:6)
```

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

While we didn't need `filter` to calculate a bond's present value in the example above, one can imagine how you may want to filter dates that a bond might pay a cashflow, say last day of a quarter:

```
using Dates
let d = Date(2024, 01, 01)
    filter(d -> lastdayofquarter(d) == d, d:Day(1):lastdayofyear(d))
end
```

```
4-element Vector{Date}:
 2024-03-31
 2024-06-30
 2024-09-30
 2024-12-31
```

### 5.4.6. More Tips on Functional Styles

#### 5.4.6.1. do Syntax for Function Arguments

In more complex situations such as with multiple collections or multi-line logic, there is a clearer syntax that is often used. `do` is a reserved keyword in Julia that creates an anonymous function and passes its arguments to a function like `map`. For example, this (terrible) code which decides if a number is prime. The anonymous function requires a `begin` block since the logic of the function is extended into multiple lines.

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

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

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

#### 5.4.6.2. Multiple Collections

`map` and the other functional operators discussed in this section can take multiple arguments. This is convenient if you have multiple arguments to a function:

```
discounts = [0.9, 0.81, 0.73]
cashflows = [10, 10, 10]

map((d, c) -> d * c, discounts, cashflows)
```

```
3-element Vector{Float64}:
 9.0
 8.100000000000001
 7.3
```

Or an example with the `do` syntax:

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

```
3-element Vector{Float64}:
 9.0
 8.100000000000001
 7.3
```

#### 5.4.6.3. Using More Functions

At the risk of sounding obvious, an easy way to make the program more "functional" is to simply use more functions. Do this one thing and it will improve the model's organization, maintainability, and reduce bugs!

Take the example from earlier:

```
pv = 0.0
discount = 1.0

for (cf, r) in zip(cf_bond, rate)
    discount = discount / (1 + r)
    pv = pv + discount * cf
end
pv
```

We can easy turn this code into a function so that it can operate on data beyond the single pair of `cf_bond` and `rate` previously defined:

```julia
function pv(rates,cashflows)
    pv = 0.0
    discount = 1.0

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

① Here, `cf_bond` and `rate` would refer to whatever was passed as arguments to the function instead of any globally defined values.

Now we could use this definition of `pv` on other instances of rates and cashflows.

#### 5.4.6.4. Mixing Funcitonal And Imperative Styles

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

> **i** Flexibility and the Lisp Curse
>
> **Lisp** ("list processing") is another, much older language than Julia (created in the 1950s!). One of it's claims to fame is how flexible and powerful the tools are within the language to build upon. There's a couple aspects of this curse that we wish to describe because we can learn from it while Julia is still a relatively young language. Part of the "curse" is that: because there's so much freedom in what can be expressed in the language, there's not an obvious "best" way of doing things. This can lead to decision paralysis where you are trying to over-analyze what's the best way to write part of your code. Our advice: *don't worry about it!* A working implemntation of something is better than an over-optimized idea.
> The other part of the "curse" is that because is that it's relatively easy to implement so many things from the building blocks that Julia provides and compose them together to do what you want. This has a downside becuase the general approach to packages is smaller, standalone pieces that you call as needed. For example, consider Python's Pandas library, upon which Python's data science community was built. It came bundled with a CSV reader, Excel reader, Database reader, DataFrame type, visualization library, and statistical functions. In Julia, each of those are separate packages that specialize for the respective topics. This is advantageous in that they can progress independently from one another, you don't have

to include functionality that you don't need, and you can mix and match libraries depending on your preference.

## 5.5. Array-Oriented Styles

Another paradigm is **array-oriented,** which is a style that relies heavily on putting similar data into arrays and operating on the entire array at the same time (as opposed to going element-by-element).

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

1. GPU programming
2. Python numerical computing

The former because GPUs want large blocks of similar data to operate in parallel. The latter is because native Python is too slow for many modeling problems so libraries like NumPy,SciPy,and tensor libraries utilize C++ (or similar) libraries for users to call out to.

Array-oriented programming is not always natural for financial and actuarial applications. Differences in behavior or timing of underling cashflows can make a set of otherwise similar products difficult to capture in nicely gridded arrays. Nonetheless, certain applications (scenario generation, some valuation routines) fit very naturally into this paradigm. Furthermore, for those that work well it's often a great way to extract additional performance due to the parallelization offered via CPU or GPU array programming.

Table 5.2 shows the bond present value example in this style.

Table 5.2.: The two code examples demonstrate the same logic using Julia and Numpy (Python's most popular array package). Julia's broadcasting facilitate an array-oriented style, similar to the approach that would be used with Python's NumPy.

| Julia | Python (NumPy) |
|---|---|
| `cf_bond = [10, 10, 10, 10, 110]`<br>`rate = [0.05, 0.06, 0.05, 0.04, 0.05]`<br>`discount_factors = cumprod(1 ./ (1 .+ rate))`<br>`result = sum(cf_bond .* discount_factors)` | `import numpy as np`<br><br>`cf_bond = np.array([10, 10, 10, 10, 110])`<br>`rate = np.array([0.05, 0.06, 0.05, 0.04, 0.05])`<br>`discount_factors = np.cumprod(1 / (1 + rate))`<br>`result = np.sum(cf_bond * discount_factors)` |

The downsides to this style are:

1. Sometimes it is unnatural because of non-uniformity of the data we are working with. For example if the length of the cashflows were shorter than the discount rates, we would have to perform intermediate steps to shorten or lengthen arrays in order to get them to be the same size.
2. A good bit of runtime performance is lost because the computer needs to allocate and fill many intermediate arrays (note how in Table 5.2, the `discount_factors` needs to instantiate an entirely new vector even though it's only temporarily used). See more on allocations in Chapter 8.

## 5.6. Recursion

A **recursive function** which is a pattern where current steps are defined in a way that depends on previous steps. Typically, an explicit starting condition is also required to be specified.

The Fibbonaci sequence is a classic example of a recursive algorithm, with the starting conditions of $n$ specified for the first two steps:

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

In code, this translates into a function definition that refers to itself:

```
function fibonacci(n)
    if n == 0
        return 0
    elseif n == 1
        return 1
    else
        return fibonacci(n-1) + fibonacci(n-2)
    end
end
```

How could a recursive pattern be defined for valuing our bond? A possible pattern is defining the present value to be the discounted value of:

- the current period's cashflow, *plus*
- the accumulated cashflows up to that point in time

Here's how that might be defined:

```
function pv_recursive(rates,cashflows,accumulated_value=0.0,discount_factor=1.0)
  if isempty(cashflows)                                                    ①
    return accumulated_value
  else
    discount_factor = discount_factor / (1+first(rates))                   ②
    av = first(cashflows) * discount_factor + accumulated_value             ③
    remaining_rates = rates[begin+1:end]
    remaining_cfs = cashflows[begin+1:end]
    return pv_recursive(remaining_rates,remaining_cfs, av,discount_factor) ④
  end
end
```

① Add a terminating condition, that if we have no more cashflows then return the accumulated value.
② Decrement the discount factor as we step forward in time.
③ Take the prior accumulated value and add the first value in the given cashflows.
④ Pass the remaining subset of the cashflow vector, the running total, and the current discount factor to the next call of the recursive function.

```
pv_recursive (generic function with 3 methods)
```

And an example of its use:

```
pv_recursive(rate,cf_bond)
```

```
121.48888490821489
```

The recursive pattern often works very nicely for simpler examples. However, more complex logic and conditionals can make this approach unwieldy. Nonetheless, attempting to distill the desired functionality into a single function can be a beneficial thought exercise.

# 6. Data and Types

> I am only one, but I am one. I can't do everything, but I can do something. The something I ought to do, I can do. And by the grace of God, I will - Edward Everett Hale (1902)

## 6.1. In this section

The powerful benefits that using assigning types to data has within the model's system, some examples of utilizing types to simplify a programs logic, and comparing aspects of different type related program organization (such as object oriented design versus composition).

## 6.2. Using Types to Value a Portfolio

We will assemble the tools and terminology to value a portfolio of assets by leverage types (@sec-data-types). Using the constructs introduced in the prior chapter, we can describe the portfolio valuation as additively reducing the mapped value of assets in the portfolio. If `value` is our valuation function), we are trying to do the following:

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

The challenge is how do design an all-purpose `value` function? In `portfolio`, the assets may be heterogeneous, so we will need to define what the valuation semantics are for the different kinds of assets. To get to our end goal, we will need to:

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

We will accomplish this by utilizing data types.

## 6.3. Benefits of Using Types

As a preview of why we want to utilize types in our program, there are a number of benefits:

1. **Separate concerns**. For example, deciding how to value an option need not know how we value a bond. The code and associated logic is kept distinct which is easier to reason about and to test.
2. **Re-use code**. When a set of types within a hierarchy all share the same logic, then we can define the method at the highest relevant level and avoid writing the method for each possible type. In our simple example we won't get as much benefit here since the hierarchy is simple and the set of types small.
3. **Dispatch on type**. By defining types for our assets, we can use multiple dispatch to define specialized behavior for each type. This allows us to write generic code that works with any asset type, and the Julia compiler will automatically select the appropriate method based on the type of the asset at runtime. This is a powerful feature that enables extensibility and modularity in our code.
4. **Improve readability and clarity**. By defining types for our assets, we make our code more expressive and self-documenting. The types provide a clear indication of what kind of data we are working with, making it easier for other developers (or ourselves in the future) to understand and maintain the codebase.
5. **Enable type safety**. By specifying the expected types for function arguments and return values, we can catch type-related errors at compile time rather than at runtime. This helps prevent bugs and makes our code more robust.

With these benefits in mind, let's start by defining the types for our assets. We'll create an abstract type called Asset that will serve as the parent type for all our asset types. If you haven't read it already, Section 4.4.7 is a good reference for details on types at the language level (this section is focused on organization and building up the abstracted valuation process).

## 6.4. Defining Types for Portfolio Valuation

We will define five types of assets in this simplified universe:

- Cash
- Risk Free Bonds (coupon and zero-coupon varieties)

To do the valuation of these, we need some economic parameters as well: risk free rates for discounting.

Here's the outline of what follows to get an understanding of types, type hierarchy, and multiple dispatch.

1. Define the Cash and Bond types.
2. Define the most basic economic parameter set.
3. Define the `value` functions for Cash and Bonds.

```
## Data type definitions
abstract type AbstractAsset end                                    ①

struct Cash <: AbstractAsset                                       ③
    balance::Float64
end

abstract type AbstractBond <: AbstractAsset end                    ②

struct CouponBond <: AbstractBond                                  ③
    par::Float64
    coupon::Float64
    tenor::Int
end

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

① General convention is to name abstract types beginning with `Abstract...`
② There can exist an abstract type which is a subtype of another abstract type.
③ We define concrete data types (`structs`) with the fields necessary for valuing those assets.

Now to define the economic parameters:

```
struct EconomicAssumptions{T}
  riskfree::T
end
```

This is a parametric type because later on we will vary what objects we use for `riskfree`. For now, we will use simple scalar values, like in this potential scenario:

```
econ_baseline = EconomicAssumptions(0.05)
```

```
EconomicAssumptions{Float64}(0.05)
```

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

```julia
value(asset::Cash, ea::EconomicAssumptions) = asset.balance
```

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

Risk free bonds are the discounted present value of the riskless cashflows. We first define a method that generically operates on any fixed bond, all that's left to do is for different types of bonds to define how much cashflow occurs at the given point in time by defining `cashflow` for the associated type.

```julia
function value(asset::AbstractBond, r::Float64)              # (2)
    discount_factor = 1.0
    value = 0.0
    for t in 1:asset.tenor
        discount_factor /= (1 + r)                            # (1)
        value += discount_factor * cashflow(asset, t)
    end
    return value
end

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

function value(bond::ZeroCouponBond, r::Float64)             # (3)
    return bond.par / (1 + r)^bond.tenor
end
```

(1) `x /= y`, `x += y`, etc. are shorthand ways to write `x = x / y` or `x = x + y`
(2) `value` is defined for `AbstractBond`s in general...
(3) ... and then more specifically for `ZeroCouponBond`s. This will be explained when discussing "dispatch" below.

```
value (generic function with 3 methods)
```

### 6.4.1. Dispatch

When a function is called, the computer has to decide which method to use. In the example above, when we want to value a `ZeroCouponBond`, does the `value(asset::AbstractBond, r)` or `value(bond::ZeroCouponBond, r)` version get used?

**Dispatch** is the process of determining 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 most specific `value(bond::ZeroCouponBond, r)`.

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

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

With dispatch, the compiler does this lookup for us, and more efficiently than enumerating a list of possible codepaths.

In our "don't do this" definition of `value` above, we used a simple scalar interest rate to determine the rate to discount the cash flows. Note how in the definition of `value` for `ZeroCouponBond`, we have defined a *more specific* signature: both the first and second arguments are specific, concrete types. When we call `value(ZeroCouponBond(100.0,3),0.05)`, we avoid the loop that's defined in the generic case and jump immediate to a more efficient definition of its `value`. This is dispatching on the combination of types and picking the most relevant (specific) version for what has been passed to it.

Despite the definitions above, the following will error because we haven't defined a method for `value` which takes as it's second argument a type of `EconomicAssumptions`:

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

Let's fix that by defining a method which takes the economic assumption type and just relays the relevant risk free rate to the `value` methods already defined (which take an `AbstractBond` and a scalar `r`).

```
value(bond::AbstractBond,econ::EconomicAssumptions) = value(bond,econ.riskfree)
```

```
value (generic function with 4 methods)
```

Now this following works:

```
value(ZeroCouponBond(100.0, 5), econ_baseline)
```

```
78.35261664684589
```

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

```
portfolio = [
    Cash(50.0),
    CouponBond(100.0, 0.05, 5),
    ZeroCouponBond(100.0, 5),
]

map(asset-> value(asset,econ_baseline), portfolio)
```

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

This is very close to the goal that we set out at the end of the section. We can complete it by reducing over the collection to sum up the value:

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

```
228.3526166468459
```

> **Note**
>
> This code:
>
> `mapreduce(asset-> value(asset,econ_baseline), +, portfolio)`
>
> is more verbose than what we set out do at the start

(`mapreduce(value,+,portfolio)`) due to the two-argument `value` function requiring a second argument for the economic variables. This works well! However, there is a way to define it which avoids the anonymous function, which in some cases will end up needing to be compiled more frequently than you want it to. Sometime we want a lightweight, okay-to-compile-on-the-fly function. Other times, we know it's something that will be passed around in compute-intensive parts of the code. A technique in this situation is to define an object which "locks in" one of the arguments but behaves like the anonymous version. There is a pair of types in the Base module, `Fix1` and `Fix2`, which represent partially-applied versions of the two-argument function f, with the first or second argument fixed to the value "x".
This is, `Base.Fix1(f, x)` behaves like `y->f(x, y)` and `Base.Fix2(f, x)` behaves like `y->f(y, x)`.
In the context of our valuation model, this would look like:

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

```
228.3526166468459
```

#### 6.4.1.1. Multiple Dispatch

A more general concept is that of **multiple dispatch**, where the types of *all arguments* are used to determine which method to use. This is a very general paradigm, and in many ways is more extensible than traditional object oriented approaches, (more on that in Section 6.5). What if instead of a scalar interest rate value we wanted to instead pass an object that represented a term structure of interest rates?

Extending the example, we can use a time-varying risk free rate instead of a constant. For fun, let's say that the risk free rate has a sinusoidal pattern:

```
econ_sin = EconomicAssumptions(t -> 0.05 + sin(t) / 100)
```

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

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

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

We can extend our methods to account for this:

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

① The `r::T ... where {T<:Function}` says use this method if `r` is any concrete subtype of the (abstract) `Function` type.
② `r` is a fuction, where we call the time to get the zero coupon bond (a.k.a. spot) rate for the given timepoint.

```
value (generic function with 5 methods)
```

Now it works:

```julia
value(ZeroCouponBond(100.0, 5), econ_sin)
```

```
82.03058910862806
```

The important thing to note here is that the compiler is using the most specific method of the function (`value(bond::ZeroCouponBond, r::T) where {T<:Function}`). Both the types of the arguments are influencing the decision of which method to use. We could go on to define the appropriate method for `CouponBond` to complete the example.

## 6.5. Objected Oriented Design

Object oriented (OO) type systems use 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[1], however there are a number of its flaws in over-relying on OO patterns. Julia does not natively have traditional OO classes and types, but much of OO design can be emulated in Julia except for data inheritance.

We bring up object oriented design for comparison's sake, but think that ultimately choosing a data driven or functional believe is better for financial modeling. Of course, many robust, well used financial models have been built this way but in our experience the abstractions become unnatural and maintenance unwieldy beyond simple examples. We'll now discuss some of the aspects of OO design and why the overuse of OO is not preferred.

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

**Note**

For readers without background in OO programming, the main features of OO languages are:

- Hierarchical type structures, which include concrete and abstract (often called classes instead of types).
- Sub-classes inherit both behavior *and* data (in Julia, subtypes only inherit behavior, not data).
- Functions that depend on the type of the object need to be ascribed to a single class and then can dispatch more specifically on the given argument's type.

## 6.6. Assigning Behavior

Needing to assign methods to a single class can lead to awkward design limitations - when multiple objects are involved in a computation, why dictate that only one of them "controls" the logic?

The `value` function is a good example of this. If we had to assign `value` to one of the objects involved, should it be the economic parameters of the asset contracts? The choice is not obvious at all. Isn't it the market (economic parameters) that determines the value? But then if `value` were to be a method wholly owned by the economic parameters, how could it possible define in advance the valuation semantics of all types of assets? What if one wanted to extend the valuation to a new asset class? Downstream users or developers would need to modify the economic types to handle new assets they wanted to value. However, because the economic types were owned by an upstream package, they can't be extended this way.

This is an issue with traditional OO designs and that resolves itself so elegantly with multiple dispatch.

## 6.7. Inheritance

We discussed the type hierarchy in Chapter 4 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.

### 6.7.1. Composition over Inheritance

To mitigate some of the problems associated with inheritance, there's a growing preference for *composition*. Composition involves creating objects that contain instances of other objects to achieve complex behaviors. This approach is more flexible than inheritance as it allows for the creation of more modular and reusable code. There is a general preference for "composition over inheritance" among professional developers these days.

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

Composition looks like this:

```
struct CUSIP
    code::string
end

struct FixedBond
    coupon::Float64
    tenor::Float64
end

struct FloatingBond
    spread::Float64
    tenor::Float64
end
```

```
struct MunicipalBond
    cusip::CUSIP
    fi::FixedBond
end

struct Swap
    float_leg::FloatingBond
    fixed_leg::FixedBond
end

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

struct UnlistedBond
    fi::FixedIncome
end
```

```
# define behavior which relies on delegation to components
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). There's no clear hierarchy here.

Composition lets us bundle the data and behavior together without needing complex chains of inheritance.

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

# 7. Higher Levels of Abstraction

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

## 7.1. In this section

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

## 7.2. Introduction

In programming and modeling, as in mathematics, abstraction permits the definition of interchangeable components and patterns that can be reused. Abstraction is a selective ignorance—focusing on the aspects of the problem that are relevant, and ignoring the others. The last two chapters described what we might call "micro" level abstractions: specific functions or types.

In this chapter, we zoom out and examine some principles that guide good model development, manifesting in architectural conerns such as how different parts of the code are organized, what parts of the program are considered 'public' versus 'private', and patterns themselves.

Chapter 4 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

## 7.3. Principles for Abstraction

Here is a list of some principles that arise when developing a particular abstraction. Not all abstractions serve all of these purposes but generally fit one or more of them.

Table 7.1.: Finding abstractions generally means finding patterns that fit into one of these principles.

| Principle | What | Why | Example |
|---|---|---|---|
| Separation of Concerns | Divide the system into distinct parts, each addressing a separate concern | Promote modularity and reduce high degree of dependence (coupling) between components | Separating data retrieval, data processing, and output generation steps in a process |
| Encapsulation | Hide the internal details of a component and expose only a clean, well-defined set of functionality (interface) | Don't let other parts of the program modify internal data and make the system easier to understand and maintain | Defining a type or module with well defined behavior and responsibiltiy |
| Composability | Design simple components that can be combined to create more complex behaviors, as opposed to a single component that attempts to handle all behavior. | Promote reuse and allow for the components to be combined creatively | Separate details about economic conditions into different types than contracts/instruments |
| Generalization | Identify common patterns and create generic components that can be specialized as needed. Often this means idenfifying the common behavior that arises repeatedly in a model | Avoid duplication and make the system more expressive and extensible | Defining a generic `Instrument` type that can be specialized for different asset classes |

These principles provide guidance for creating abstractions that are modular, reusable, and maintainable. By following these principles, developers can create financial models that are easier to understand, extend, and adapt to changing requirements.

### 7.3.1. Pragramatic Considerations for Model Design

#### 7.3.1.1. 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.

#### 7.3.1.2. 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.

#### 7.3.1.3. 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.

## 7.4. Interfaces

**Interfaces** are the boundary between different encapsulated abstractions. The user-facing interface is the set of functionality and details that the user of the package or model must consider, which is separate from the intermediate variables, logic, and complexity that may be contained within.

> **i** Example of an interface
>
> When looking up a ticker for a market quote, one need not be mindful of the underlying realtime databases, networking, rendering text to the screen, memory management, etc. The interface is "put in symbol, get out number". By design, there are multiple layers of interfaces and abstractions used under the hood, but the financial modeler need only be actively concerned about the points that he or she comes in contact with, not the entire chain of complexity.

For a financial model 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.

### 7.4.1. Defining Good Interfaces

A well-designed interface should follow these principles:

1. **Be minimal and focused.** The interface should provide only the essential functionality needed, without unnecessary clutter or features. This makes the interface

easier to understand and and faciliates building the necessary complexity through digestable, composable components.
2. **Be consistent and intuitive.** The interface should use consistent naming conventions, parameter orders, and behaviors. It should match the user's mental model and expectations.
3. **Hide implementation details.** The interface should abstract away the internal complexity and expose only what the user needs to know. This of details allows the implementation to change without affecting users of the interface.
4. **Be documented and contractual.** The interface should clearly specify what inputs it expects and what outputs or behaviors it provides. It forms a contract between the implementation and the users.
5. **Be testable.** A good interface allows the functionality to be easily tested through the public interface, without needing to access internal details.

### 7.4.2. Interfaces: A Financial Modeling Case Study

As a case study, we'll look at the FinanceModels.jl and related packages to discuss some of the background and design choices that went into the functionality. This suite was written by one of the authors and is publically available as set of installable Julia packages.

#### 7.4.2.1. Background

In actuarial work, it is common to need to work with interest rate and bond yield curves to determine current forward rates, estimates of the shape of future yield curves, or discount a series of cashflows to determine a present value. Determining things like "given a par yield curve, what's the implied discount factor for a cashflow at time 10" or "what is the 10 year BBB public corporate rate implied by the current curve in five years' time" is cumbersome at best in a spreadsheet.

For example, to determine the answer to the first one ("a discount factor for time 10") actually requires quite a bit of detail and assumption to derive:

- Reference market data and a specification for how that market data should be interpreted. For example, if given the rate `0.05` for time 10, quoted as a continuous rate or annual effective? Is that a par rate, a zero-coupon bond (spot) rate, or a one-year-forward rate from time 10?
- Smoothing, interpolation, or extrapolation for noisy or sparse data. Should the rates be bootstrapped or fit to a parametrically specified curve?

This is the type of complexity that we wish to save the user from needing to keep front of mind when the primary goal is, e.g., valuation of a stream of riskless life insurance payments, which might look like this:

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

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

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

#### 7.4.2.2. Initial Versions

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

##### 7.4.2.2.1. Rates

Utilizing the benefit of the type system, it was decided that it would be most useful to represent rates not as simple floating point numbers (e.g. `0.05`) but instead with dedicated types to distinguish between rate conventions. The abstract type `CompoudingFrequency` had two subtypes: `Continuous` and `Periodic` sop that a 5% rate compounded continuously versus an effective per period rate would be distinguished via `Continuous(0.05)` versus `Periodic(0.05,1)`. The two could be converted between by extending the built-in `Base.convert` function.

This was useful because once rates were converted into `Rates` within the ecosystem, that data contained within itself charateristics that could distinguish how downstream functionality should treat the rates.

##### 7.4.2.2.2. Yield Curves

At first, only bootstrapping was supported as a method to construct curve objects. This required that there was only one rate given per time period (no noisy data) and only supported linear, quadratic, and cubic splines.

Further, there was a specific constructor for different common types of instruments. From the old documentation:

- `Yields.Zero(rates,maturities)` using a vector of zero, or spot, rates
- `Yields.Forward(rates,maturities)` using a vector of one-period
- `Yields.Constant(rate)` takes a single constant rate for all times

- `Yields.Par(rates,maturities)` takes a series of yields for securities priced at par.Assumes that maturities <= 1 year do not pay coupons and that after one year, pays coupons with frequency equal to the CompoundingFrequency of the corresponding rate.
- `Yields.CMT(rates,maturities)` takes the most commonly presented rate data (e.g. Treasury.gov) and bootstraps the curve given the combination of bills and bonds.
- `Yields.OIS(rates,maturities)` takes the most commonly presented rate data for overnight swaps and bootstraps the curve.

This covered a lot of lightweight use-cases, but made a lot of implicit assumptions about how the given rates should be interpreted.

#### 7.4.2.3. The Birth of FinanceModels

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

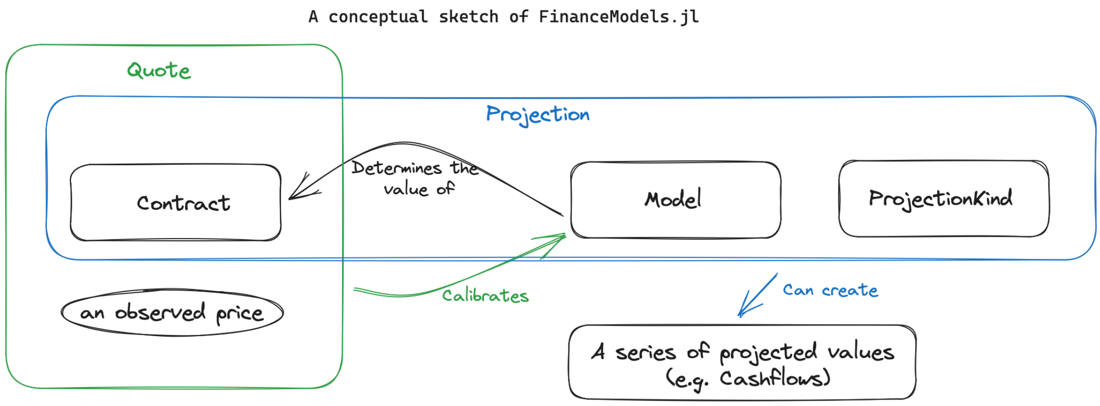



Figure 7.1.: A conceptual sketch of FinanceModels.jl components.

First, realizing that yield curves were just a particular kind of model - one that used interest rates to discount cashflows. But you can have different kinds of models - such as Black-Scholes option valuation or a Monte Carlo valuation approach. Likewise, the cashflows need not simply be a vector of floating point values, and instead it could be the representation of a generic financial contract. As long as the model knew how to value it, an appropriate present value could be derived.

Where previously it was:

```
present_value(yield_curve,cashflow_vector)
```

Now, it was

```
present_value(model,contract)
```

Second, that a model was simple some generic box that had been "fit" to previously observed prices for similar types of contracts we would be trying to value in the model. The combination of a contract and a price constituted a "quote" and with multiple quotes a model could be fit using various algorithms.

With these changes, the package that was originally called Yields.jl was renamed to FinanceModels.jl. The updated code from the earlier example now would e implemented like this:

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

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

It's slightly more verbose, but notice how much more powerful and extensible `fit(Spline.Cubic(), quotes, Fit.Boostrap()` is than `Yields.Par(risk_free_rates,tenors)`. The end result is the same, but now the same package and interface can clearly interchange other options, such as a `NelsonSiegelSvennson` curve instead of a spline. And the `quotes` could be a combination of observed bonds of different technical parameters (though still sharing characteristics which make it relevant for the model being constructed).

The same pattern also applies for option valuation, such as this example of vanilla euro options with an assumed constant volatility assumption:

```
a = Option.EuroCall(CommonEquity(), 1.0, 1.0)                          ①
b = Option.EuroCall(CommonEquity(), 1.0, 2.0)

qs = [                                                                  ②
    Quote(0.0541, a),
    Quote(0.072636, b),
]

model = Equity.BlackScholesMerton(0.01, 0.02, Volatility.Constant())    ③

m = fit(model, qs)                                                      ④

present_value(m,qs[1].instrument)                                       ⑤
```

① The arguments to `EuroCall` are the underlying asset type, strike, and maturity time.
② A vector of observed option prices.
③ A BSM model with a given risk free rate, dividend yield, and a to-be-fit constant volatility component.
④ Fits the model and derives an approximate volatility of `0.15` .
⑤ Values the contract and in such a simple, noiseless model we recover the original price of `0.0541`

With a consistent interface able to handle a wide variety of situations, the modeler is free to expand the model in new directions of analysis with the built in functionality allowing him or her to compose pieces together that was not possible with the less abstracted design. For example, the equity option example had no parallel when all of the available constructors were `Yields.Zero` or `Yields.Par` and would have required a completely from-scratch implementation with newly defined functions.

Further, and critically, the new design allows modelers to create their own models or contracts[1] and extend the existing methods rather than needing to create their own: the function signature `fit(model,quotes)` handles a very wide variety of cases, as does `present_value(model,contract)`.

## 7.5. Macros & Homoiconicity

We've talked about transforming data and restructuring logic in order to make the model more effective. We can go still deeper!(Or is it higher level?) We can actually abstract the process of writing code itself! This subject is a bit advanced, so we are simply going to introduce it because you will likely find many convenient instances of it as a *user* even if you never find a need to implement this yourself.

**Homoiconicity** refers to the property of a programming language where the language's code can be represented and manipulated as a data structure in the language itself. In other words, the code is data and can be treated as such. This enables powerful metaprogramming (i.e. code that writes other code) capabilities, where code can be generated or transformed during the compilation process.

**Macros** are a metaprogramming feature that leverage homoiconicity in Julia. They allow the programmer to write code that generates or manipulates other code at compile-time. Macros take code as input, transform it based on certain rules or patterns, and return the modified code which then gets compiled.

For example, a built-in macro is `@time` which will measure the elapsed runtime for a piece of code[2].

[1] And projections, which is handled by defining a `ProjectionKind` , such as a cashflow or accounting basis. This topic is covered in more detail in the FinanceModels.jl documentation.
[2] (**time?**) is a simple, built-in function. For true benchmarking purposes, see **?@sec-benchmarking**.

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

Will effectively expand to:

```
t0 = time_ns()
value = exp(rand())
t1 = time_ns()
println("elapsed time: ", (t1-t0)/1e9, " seconds")
value
```

Here it is when we run it:

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

```
  0.000002 seconds
```

```
1.1358533220262552
```

### 7.5.1. Metaprogramming in Financial Modeling

In the context of financial modeling, macros can be used to simplify repetitive or complex code patterns, enforce certain conventions or constraints, or generate code based on data or configuration.

Here are a few potential use cases of macros in financial modeling. Again, these are more advanced use-cases but knowing that these paths exist may benefit your work in the future.

1. Defining custom DSLs (Domain-Specific Languages): Macros can be used to create expressive and concise DSLs tailored to financial modeling. For example, a macro could allow defining financial contracts using a syntax closer to the domain language, which then gets expanded into the underlying implementation code.
2. Automating boilerplate code: Macros can help reduce code duplication by generating common patterns or boilerplate code. This can include generating accessor functions[3], constructors, or serialization logic based on type definitions.
3. Enforcing conventions and constraints: Macros can be used to enforce coding conventions, such as naming rules or type checks, by automatically transforming code that doesn't adhere to the conventions. They can also be used to add runtime assertions or checks based on certain conditions.

[3] Accessor functions are useful when working with nested data structures. For example, if you have a `struct` within a `struct` and want to conveniently access an inner structs field.

4. Optimizing performance: Macros can be used to perform code optimizations at compile-time. For example, a macro could unroll loops, inline functions, or specialize generic code based on specific types or parameters, resulting in more efficient runtime code.
5. Generating code from data: Macros can be used to generate code based on external data or configuration files. For example, a macro could read a specification file and generate the corresponding financial contract types and functions.

# Part III.

# Foundations: Building Performant Models

> "Premature optimization is the root of all evil (or at least most of it) in programming." - Donald Knuth

After establishing foundational programming concepts, we turn our attention to performance - a critical consideration for real-world financial models. While modern computers are remarkably powerful, thoughtlessly constructed models can still grind to a halt when faced with large portfolios or complex analyses. This section explores how to harness computational resources effectively, starting from the hardware fundamentals that both constrain and enable our work.

Understanding performance requires looking beneath the abstractions we've built. Just as a financial modeler benefits from understanding the mechanics of markets and instruments rather than treating them as black boxes, knowledge of computational infrastructure allows us to make informed decisions about model architecture and implementation. We'll examine how hardware characteristics influence algorithm design, memory usage patterns, and execution speed.

We'll introduce when optimization *does* matter, and equally important when it *doesn't*. The goal isn't to optimize prematurely or pursue performance at all costs. Rather, we aim to build models that scale gracefully as demands grow, whether through larger datasets, more sophisticated analyses, or tighter time constraints. We'll progress from single-threaded optimization techniques to parallel processing approaches, always with an eye toward practical application in financial contexts.

By the end of this section, you'll have the knowledge needed to diagnose performance bottlenecks and implement appropriate solutions, ensuring your models remain responsive and reliable as they evolve. Let's begin by examining the hardware foundation upon which all our computational work rests.

# 8. Hardware and Its Implications

> a CPU is literally a rock that we tricked into thinking.
>
> not to oversimplify: first you have to flatten the rock and put lightning inside it.
>
> - Twitter user daisyowl, 2017

## 8.1. In this section

In this chapter, we'll explore why a basic understanding of computing hardware is essential for optimizing financial models and working efficiently with data. Understanding how data is stored and processed can help you make better design decisions, improve performance, and avoid common pitfalls. We'll cover topics like memory architecture, data storage, and the impact of hardware on computational speed.

## 8.2. Introduction

The quote that opens the chatper is a silly way of describing that most modern computers are made with silicone, a common mineral found in rocks. However, we will not concern ourselves with the raw materials of computers and instead will focus on the key architectural aspect.

A computer handles data at rest (in memory) or data being acted upon (processed). This chapter will try to explain both of those processes in a way that reveals key reasons why different approaches to programming can yield different results in terms of processing speed, memory usage, and compiled outputs.

## 8.3. Memory and Moving Data Around

The core of modeling on computers is to perform computations on data, but unfortunately the speed at which data can be *accessed* has grown much slower than the rate the actual computations can be performed. Further, the size of the available persistent data storage (HDDs/SSDs) has ballooned, exacerbating the problem: the overall throughput of memory is the typical workflow constraint. To try to address the bottleneck (memory throughput), solutions have been developed to create a pipeline to efficiently shuttle data to and from the processor and the persistent storage. This memory and processing architecture applies at both the single computer level as well as extending to workflows between different data stores and computers.

We will focus primarily on the architecture of a single computer, as even laptop computers today contain enough power for most modeling tasks, *if the computer is used effectively*. Further, learning how to optimize a program for a single computer/processesor is almost always a precursor step to effective parallelization as well.

### 8.3.1. Memory Types and Location

Memory has an inverse relationship between size and proximity to the central processor unit (CPU). The closer the data is to the processor units, the smaller the storage and the less likely the data will persist at that location for very long.

| Kind | Rough Size | Lifecycle |
|---|---|---|
| Solid State Disk (SSD) or Hard Disk Drive (HDD) | TBs | Persistent/Permanent |
| Random Access Memory (RAM) | Dozens to Hundereds of GBs | Seconds to Hours (while computer is powered on) |
| CPU Cache - L3 | 8 MB to 128 MB | Microseconds to Milliseconds |
| CPU Cache - L2 | 2 MB to 16 MB | Nanoseconds to Microseconds |
| CPU Cache - L1 | ~16 KB | Nanoseconds |

After requesting data from a persistent location like a Solid State Drive (SSD), the memory is read into Random Access Memory (RAM). The advantage of RAM over a persistent location is speed - typically that memory can be accessed and modified many times faster than the persistent data location. The tradeoff is that RAM is not persistent: when the computer is powered down, the RAM loses the information stored within.

When data is needed by the CPU, data is read from RAM into a small hierarchy of caches before being accessed by the CPU. The **CPU Caches** are small (physically and in capacity), but very fast. The caches are also phyiscally colocated with the CPU for efficiency. Data is organized and funneled through the caches as an intermediary between the CPU and RAM and is fed from Level 3 (L3) cache in steps down to L1 cache as the data gets closer to the processor.

**Note**

For reference, memory units are:

- 1 bit is a single binary digit (`0` or `1`)
- 8 bits = 1 byte
- 8 bytes = 1 word
- 1024 bytes = 1 Kilobyte (KB)
- 1024 KB = 1 Megabyte (MB)
- 1024 MB = 1 Gigabyte (GB)
- 1024 GB = 1 Terabyte (TB)
- 1024 TB = 1 Petabyte (PB)

Sometimes, you might see Kb, which is Kilo*bits*, or 1024 bits. Therefore 1 KB is 8 times larger than a Kb.
The increments are 1024 and not the usual 1000, because 1024 is $2^{10}$. The even binary multiple of 1024 is more convienient than 1000 when working with bits.

### 8.3.2. Stack vs Heap

Sitting within the RAM region of memory are two sections called Stack and Heap. These are places where variables created from our program's code can be stored. In both cases, the program will request memory space but they have some differences to be aware of.

The **stack** stores small, fixed-size (known bit length), data and program components. The stack is a last-in-first-out queue of data that is able to be written to and read from very quickly. The **heap** is a region which can be dynamically sized and has random read/write (you need not access the data in a particular order). The heap is much slower but more flexible.

#### 8.3.2.1. Garbage Collector

The **garbage collector** is a program that gets run to free up previously requested/allocated memory. It accomplishes this by keeping track of references to data in memory by section of your code. If a section of code is no longer reachable (e.g. inside a function that will never get called again, or a loop that ran earlier in the program but is now complete), then, periodically, the garbage collector will pause execution of the primary program in order to "sweep" the memory. This step marks the space as able to be reused by your program or the operating system.

## 8.4. Processor

The processor reads lines (groups of bits) from the cache into registers and then executes instructions. An example would be to take the bytes from register 10 and add the bytes from register 11 to them. This is really all a processor does at the lowest level: combining bits of data using logical circuits.

Logical circuits (**transistors**) are an arrangement of wires that output a new electrical signal that varies depending on the input. From a collection of smaller building block gates (e.g. AND, OR, NOR, XOR) more complex operations can be built up[1], into operations like addition, multiplication, division, etc. Electric impulses move the state of the program forward once time per CPU cycle (controlled by a master "clock" ticking billions of times per second). CPU cycle speed i what's quoted for chip performance, e.g. when a CPU is advertised as as 3.0 GHz (or 3 billion cycles per second).

The programmer (or compiler, if we are working in a higher level language like Julia) tells the CPU which instruction to run. The set of instructions that are valid for a given processor are called the Instruction Set Architecture, or ISA. In computer Assembly language (roughly one-level above directly manipulating the bits), the instructions are given names like `ADD`, `SUB`, `MUL`, `DIV`, and `MOV`. These instructions mirror the raw instruction that is part of the ISA.

All instructions are not all created equal, however. Some instructions take many CPU cycles to complete. For example, floating point `DIV` (division) takes 10-20 CPU cycles while `ADD` only takes a single CPU cycle.

Some architecture examples that may be familiar:

- Intel x86-64 (a.k.a. AMD64) are common computer processors that use registers that are 64 bits wide (the prior generation was 32 bits wide) and use the **x86** instruction set developed by Intel.
- ARM chips, including the Apple M-Series processors are characterized by the use of the **ARM** instruction set and recent processors of this kind are also 64 bit.

The ARM architecture is known as a **reduced instruction set chip** (RISC), which means that it has fewer available instructions compared to, e.g. the x86 architecture. The benefit of the reduced instruction set is that it is generally much more power efficient, but comes at the cost of sacrificing specialized instructions such as string manipulation, or in lower-end chips, even the division operation (which have to be implemented via software routines instead of CPU operations). However, for specialized workloads, the availability of a key instruction can make a program run on the CPU 10-100x faster at

[1] In fact, only two logical gates are needed to reproduce all boolean logical gates: NAND (Not AND) and NOR gates can be composed to create AND, OR, NOR, etc. gates.

times. An example of this is that at the time of writing, AVX512 processors are becoming available (see Chapter 10 for a discussion of vectorization) which can benefit some workloads greatly.

> **Tip**
>
> Trying to optimize your program via selecting specialized chips should be one of the *last* ways that you seek to optimize runtime, as generally a similar order of magnitude speedup can be achieved through more efficient algorithm design or general parallelization techniques. Developing programs in this way makes the performance *portable*, able to be used on other systems and not just special architectures.

When writing in Julia, you need not be concerned with the low-level instructions as the compiler will optimize the execution for you. However, should it be useful, it is easy to inspect the compiled code. For example, if we create a function to add three numbers, we can see that the `ADD` instruction is called twice: first adding the first and second arguments, and then adding the third argument to that intermediate sum.

```julia
myadd(x, y, z) = x + y + z
@code_native myadd(1, 2, 3)
```

```
    .section    __TEXT,__text,regular,pure_instructions
    .build_version macos, 15, 0
   .globl  _julia_myadd_5721            ; -- Begin function julia_myadd_5721
    .p2align    2
_julia_myadd_5721:                      ; @julia_myadd_5721
; Function Signature: myadd(Int64, Int64, Int64)
; ┌ @ In[2]:1 within `myadd`
; %bb.0:                                ; %top
; │ @ In[2] within `myadd`
    ;DEBUG_VALUE: myadd:x <- $x0
    ;DEBUG_VALUE: myadd:x <- $x0
    ;DEBUG_VALUE: myadd:y <- $x1
    ;DEBUG_VALUE: myadd:y <- $x1
    ;DEBUG_VALUE: myadd:z <- $x2
    ;DEBUG_VALUE: myadd:z <- $x2
; │ @ In[2]:1 within `myadd`
; │┌ @ operators.jl:596 within `+` @ int.jl:87
    add x8, x1, x0
    add x0, x8, x2
    ret
; └└
                                        ; -- End function
```

```
.subsections_via_symbols
```

Compilers are complex, hyper-optimized programs which turn your source code into the raw bits executed by the computer. Key steps in the process of converting Julia code you write all the way to binary machine instructions include the items in Table 8.2. Note the Julia `@code_...` macros allow the programmer to inspect the intermediate representations.

Table 8.2.: Part the key to Julia's speed is to be able to compile down to a different, specialized version of the machine code depending on the types given to a function. As described in the table above, the instructions for adding floating point together or integer numbers together are different. Julia code can reflect that distinction by compiling a different method for each combination of input types.

| Step | Description | Example |
|---|---|---|
| Julia Source Code | The level written by the programmer in a high level language. | myadd(x,y,z) = x + y + z |
| Lowered Abstract Syntax Tree (AST) | An intermediate representation of the code after the first stage of compilation, where the high-level syntax is simplified into a more structured form that's easier for the compiler to work with. | julia> @code_lowered myadd(1,2,3)<br>CodeInfo(<br>1 ─ %1 = x + y + z<br>└──      return %1<br>) |
| LLVM | Low-Level Virtual Machine language, which is a massively popular compiler used by languages like Julia and Rust. The core logic are the three lines with `add`, `add`, and `ret`.<br>Note that the the `add` instruction is `add i64` which means an addition operation of 64 bit integers. | julia> @code_llvm myadd(1,2,3)<br>;  @ REPL[7]:1 within \`myadd\`<br>define i64 @julia_myadd_2022(i64 signe<br>top:<br>; ┌ @ operators.jl:587 within \`+\` @ in<br>   %3 = add i64 %1, %0<br>   %4 = add i64 %3, %2<br>   ret i64 %4<br>; └<br>} |

<table>
<tr><th>Step</th><th>Description</th><th>Example</th></tr>
<tr><td>Native</td><td>The final machine code output, specific to the target CPU architecture. This is at the same level as Assembly language. The core logic are the three lines beginning with <code>add</code>, <code>add</code>, and <code>ret</code>. If we used floating point addition instead, the CPU instruction would be <code>fadd</code> instead of <code>add</code>.</td><td><pre><code>julia> @code_native myadd(1,2,3)
        .section        __TEXT,__text,
        .build_version macos, 14, 0
        .globl  _julia_myadd_1851
        .p2align        2
_julia_myadd_1851:
; ┌ @ REPL[7]:1 within `myadd`
; %bb.0:
; │┌ @ operators.jl:587 within `+` @ i
        add     x8, x1, x0
        add     x0, x8, x2
        ret
; └└</code></pre></td></tr>
</table>

#### 8.4.0.1. Increasing Complexity in Search of Performance

Transistors are the building-block that creates the CPU and enables the physical process which governs the computations. For a very long time, the major source of improved computer performance was simply to make smaller transistors, allowing more of them to be packed together to create computer chips. This worked for many years and the propensity for the transistor count to double about every two years. In this way, software performance improvements came as side effect of the phenomenal scaling in hardware capability. However, raw single core performance and clock frequency (CPU cycle speed) dramatically flattened out starting a bit before the year 2010. This was due to the fact that transistor density has been starting to be limited by:

1. Pure physical constraints (transistors can be measured in width of atoms) where we have limited ability to manufacture something so small.
2. Thermodynamics, where heat can't be removed from the CPU core fast enough to avoid damaging the core and therefore operations per second are capped.

To obtain increasing performance, two main strategies have been employed in lieu of throwing more transistors into a single core:

1. Utilize multiple, separate cores and operate in an increasingly parallel way.
2. Use clever tricks to predict, schedule, and optimize the computations to make better use of the memory pipeline and otherwise idle CPU cycles.

We will cover techniques to utilize concurrent/parallel processing in Chapter 10. As for the second technique, it is capable of very impressive accelerations (on the order 2x to 100x faster than a naive implementation. However, it has sometimes caused issues. There have been some famous security vulnerabilities such as Spectre and Meltdown, which exploited speculative execution – a technique used to optimize CPU performance which will execute code *before being explicitly asked to* because the scheduler *anticipates* the next steps (with very good, but imperfect accuracy).

## 8.5. Logistics Warehouse Analogy

The problem is analogous to a logistics warehouse (persistent data) which needs to package up orders (processor instructions). There's a conveyor belt of items being constantly routed to the packaging station. In order to keep the packing station working at full capacity, the intermediate systems (RAM & CPU caches) are funneling items they *think* will be needed to the packager (data that's *expected* to be used in the processor). Most of the time, the necessary item (data) is optimally brought to the packaging station(process), or a nearby holding spot (CPU cache).

This system has grown very efficient, but sometimes the predictions miss or a never-before-ordered item needs to be picked from the far side of the warehouse and this causes significant delays to the system. Sometimes a package will start to be assembled before the packager has even gotten to that order (branch prediction) which can make the system faster most of the time, but if the predicted package isn't actually what the customer ordered, then the work is lost and has to be redone (branch mispredict).

There are a lot more optimizations along the way:

- Since the items are already mostly arranged so that related items are next to each other, the conveyor belt will bring nearby items at the same time it brings the requested item (memory blocks).
- If an item usually ordered after another one is, the conveyor system will start to bring that second item as soon as the first one is ordered (prefetching).
- Different types of packaging stations might be used for specialized items (e.g. vector processing or cryptography instructions in the CPU).

## 8.6. Speed of Computer Actions

In a financial model, even small delays (such as main memory references vs. L1 cache access) can accumulate quickly in high-frequency trading or risk calculation routines. Understanding these timings can guide decisions on structuring data access patterns and deciding what data to cache or load in memory for optimal performance.

Representative time is given in Table 8.3 for a variety of common actions perfromed on a computer. It's clear that having memmory access from a local source is better for computer performance!

Table 8.3.: How long different actions take on a computer. As an interesting yardstick, the distance a beam of light travels is also given to provide a sense of scale (this comparison originally comes from Admiral Grace Hopper). Source for the timings comes from: https://cs.brown.edu/courses/csci0300/2022/assign/labs/lab4.html

| Operation | Time (ns) | Distance Light Traveled |
|---|---|---|
| Single CPU Cycle (e.g. one `ADD` or `OR` operation on a register) | 0.3 | 9 centimeters |
| L1 cache reference | 1 | 30 centimeters |
| Branch mispredict | 5 | 150 centimeters |
| L2 cache reference | 5 | 150 centimeters |
| Main memory reference | 100 | 30 meters |
| Read 1MB sequentially from RAM | 250,000 | 75 km (~2 marathons) |
| Round trip within a datacenter | 500,000 | 150km (the thickness of Earth's atmosphere) |
| Read 1MB sequentially from SSD | 1,000,000 | 300km (distance Washington D.C. to New York City) |
| Hard disk seek | 10,000,000 | 3,000km (width of continental United States) |
| Send packet CA->Netherlands->CA | 150,000,000 | 45,000km (circumference of earth) |

# 9. Writing Performant Single-Threaded Code

> Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away. - Antoine de Saint-Exupéry, Airman's Odyssey

## 9.1. In This Chapter

Understanding single-threaded performance, strategies for efficient sequential code, recognizing when parallelization is needed, practical comparisons of single-threaded and parallel approaches, optimizing code in Julia.

## 9.2. Introduction

With today's hardware, the highest throughput computations utilize GPUs for massive parallelization. However, writing parallel code, let alone performant parallel code, typically relies heavily on understanding patterns of non-parallel code. Secondly, many problems are not "massively parallelizable" and a sequential model architecture is required. For these reasons, it's critical to understand sequential patterns before moving onto parallel code.

For those coming from fundamentally slower languages (such as R or Python), the common advice to speed things up is often to try to parallelize code. With many high level languages the *only* way to achieve reasonable runtime is to utilize parallelism. In contrast, fast languages like Julia can output surprisingly quick single threaded prgrams

Lastly, it may be that a simpler, easier to maintain sequential model is preferable to a more complex parallel version if maximum performance is not required. Like the quote that openend the chapter, you may prefer a simpler sequential version of a model to a more complex parallel one.

Developer time (your time) is often more expensive than runtime, so be prepared to accept a good-enough but sub-optimal code instead of spending a lot of time optimizing every last nanosecond out of it!

## 9.3. Patterns of Performant Sequential Code

### 9.3.1. Minimize Memory Allocations

Allocating memory onto the Heap takes a lot more time than (1) not using intermediate memory storage at all, or (2) utilizing the Stack. Each allocation requires time for memory management and requires the garbage collector, which can significantly impact performance, especially in tight loops or frequently called functions.

> **Note**
>
> **Tight loops** or **hot loops** are the performance critical section of the code that are performed many times during a computation. They are often the "inner-most" loop of a nested loop algorithm.

In Julia, a general rule of thumb is that dynamically sizable or mutable objects (arrays, mutable structs) will be heap allocated while small fixed size objects can be stack allocated. For mutable objects, a common technique is to pre-allocate an array and then re-use that array for subsequent calculations. In the following example, note how we pre-allocate the output vector instead of creating vectors for each bond *and then* summing the vectors together at the end:

```
end_time = 10
cashflow_output = zeros(end_time)

par_bonds = map(1:1000) do i
    (tenor=rand((3, 5, 10)), rate=rand() / 10)
end

for asset in par_bonds
    for t in 1:end_time
        if t == asset.tenor
            cashflow_output[t] += 1 + asset.rate
        else
            cashflow_output[t] += asset.rate
        end
    end
end
cashflow_output
```

```
10-element Vector{Float64}:
 48.44969328880841
 48.44969328880841
```

```
 379.4496932888087
  48.44969328880841
 381.4496932888087
  48.44969328880841
  48.44969328880841
  48.44969328880841
  48.44969328880841
 384.44969328880865
```

Julia's `@allocated` macro will display the number of bytes allocated by an expression, helping you identify and eliminate unnecessary allocations.

```julia
random_sum() = sum([rand() for _ in 1:10])
@allocated random_sum()
```

```
144
```

### 9.3.2. Optimize Memory Access Patterns

Optimizing memory access patterns is essential for leveraging the CPU's cache hierarchy effectively. Modern CPUs have multiple levels of cache (L1, L2, L3), each with different sizes and access speeds. By structuring your code to access memory in a cache-friendly manner, you can significantly reduce memory latency and improve overall performance.

What is cache-friendly memory access? Essentially it boils down to spatial and temporal locality.

#### 9.3.2.1. Spatial Locality

Spatial locality refers to accessing data that is physically near each other in memory (e..g contiguous blocks of data in an array).

For example, it's better to access data in a linear order rather than random order. For example, if we sum up the elements of an array in order it will be significantly faster than if we do it randomly:

```julia
using BenchmarkTools, Random

# Create a large array of structs to emphasize memory access patterns
struct DataPoint
    value::Float64
    # Add padding to make each element 64 bytes (a typical cache line size)
```

```
    padding::NTuple{7,Float64}
end

function create_large_array(n)
    [DataPoint(rand(), tuple(rand(7)...)) for _ in 1:n]
end

# Create a large array
const N = 1_000_000
large_array = create_large_array(N)

# Function for sequential access
function sequential_sum(arr)
    sum = 0.0
    for i in eachindex(arr)
        sum += arr[i].value
    end
    sum
end

# Function for random access
function random_sum(arr, indices)
    sum = 0.0
    for i in indices
        sum += arr[i].value
    end
    sum
end

# Create shuffled indices
shuffled_indices = shuffle(1:N)

# Benchmark
println("Sequential access:")
@btime sequential_sum($large_array)

println("\nRandom access:")
@btime random_sum($large_array, $shuffled_indices)
```

```
Sequential access:
  903.000 μs (0 allocations: 0 bytes)

Random access:
```

```
  3.130 ms (0 allocations: 0 bytes)
```

```
500064.1279314945
```

When the data is accessed in a linear order, it means that the computer can load chunks of data into the cache and it can operate on that cached data for several cycles before new data needs to be loaded into the cache. In contrast, when accessing the data randomly, then the cache frequently needs to be populated with a different set of bits from a completely different part of our array.

#### 9.3.2.1.1. Column vs Row Major Order

All multi-dimensional arrays in computer memory are actually stored linearly. When storing the multi-dimensional array, an architectural decision needs to be made at the language-level and Julia is column-major, similar to many performance-oriented languages and libraries (e.g. LAPACK, Fortran, Matlab). Values are stored going down the columns instead of across the rows.

For example, this 2D array would be stored as `[1,2,3,...]` in memory, which is made clear via `vec` (which turns a multi-dimensional array into a 1D vector):

```
let
    array = [
        1 4 7
        2 5 8
        3 6 9
    ]

    vec(array)
end
```

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

When working with arrays, prefer accessing elements in column-major order (the default in Julia) to maximize spatial locality. This allows the CPU to prefetch data more effectively.

You can see how summing up values across the first (column) dimension is much faster than summing across rows:

```julia
@btime sum(arr, dims=1) setup = arr = rand(1000, 1000)
@btime sum(arr, dims=2) setup = arr = rand(1000, 1000)
```

```
  85.625 μs (3 allocations: 7.95 KiB)
  133.667 μs (3 allocations: 7.95 KiB)
```

```
1000×1 Matrix{Float64}:
 491.9795493825407
 497.47696127478537
 501.2474881082924
 503.7154272458403
 509.94899367139425
 495.75551031311215
 498.87077490062387
 486.6200849808757
 497.5282471523891
 505.7432920387825
 505.860172845917
 514.070136095101
 501.23822056747315
   ⋮
 512.5211957848306
 490.4446585123589
 505.46131548119365
 494.6225136362439
 486.62682205880407
 496.1345226700908
 487.09278001260776
 512.8091335113438
 490.4921427423747
 504.9788222390574
 491.3766680101595
 499.3738682694348
```

#### 9.3.2.2. Temporal Locality

The scheduler and branch prediction will recognize data that is accessed together closely in time and prefetch relevant blocks of data. This is an example of keeping “hot” data more readily accessible to the CPU than “cold” data.

Sometimes this happens at the operating system level - if you load a dataset that exceeds the available RAM, some of the “active” memory will actually be kept in a space on the persistent disk (e.g. your SSD) to avoid the computer crashing from being out of memory. Segments of the data that have been accessed recently will be in RAM while section of the data not recently accessed are likely to be in the portion stored on the persistent disk.

### 9.3.3. Use Efficient Data Types

The right data type can lead to more compact memory representations, better cache utilization, and more efficient CPU instructions. This is another case of where having a smaller memory footprint allows for higher utilization of the CPU since computers tend to be memory-constrained in speed.

On some CPUs, you may find performance if use the smallest data type that can accurately represent your data. For example, prefer Int32 over Int64 if your values will never exceed 32-bit integer range. For floating-point numbers, use Float32 instead of Float64 if the reduced precision is acceptable for your calculations. These smaller types not only save memory but also allow for more efficient vectorized operations (see **?@sec-parallelism**) on modern CPUs. Sometimes using the smaller datatype isn’t beneficial: if you have a 64-bit architecture machine then the overhead of converting to/from 64-bit numbers may outweigh any speedup from higher memory throughput.

For collections, choose appropriate container types based on your use case. Arrays are efficient for calculations that loop through all or most elements, while Dictionaries are better for sparse look-ups or outside of the “hot loop” portion of a computation.

Consider using StaticArrays for small, fixed-size arrays, as they can be allocated on the stack and lead to better performance in certain scenarios than dynamically sizeable arrays. The trade-off is that the static arrays require more up-front compile time and after a certain point (length in the 50-100 element range) it usually isn’t worth trying to use StaticArrays.

### 9.3.4. Avoid Type Instabilities

Type instabilities occur when the compiler cannot infer a single concrete type for a variable or function return value. These instabilities can significantly hinder Julia’s ability to generate optimized machine code, leading to performance degradation. When the compiler is not able to infer the types at compile-time (**compile time dispatch**), then while the program is running a lookup needs to be perfomed to find the most appropriate functions for the given type (**runtime dispatch**). Preferablly, when the types are known at compile-time, Julia is able to create machine code that will point directly to the desired function instead of needing to perform that lookup.

To avoid type instabilities, ensure that functions have consistent return types across all code paths. For example:

```
function multiple_return_types()
    if rand() > 0.5
        return 1.0 # Float
    else
        return 2   # Int
    end
end

function single_return_type()
    if rand() > 0.5
        return 1.0 # Float
    else
        return 2.0 # Float
    end
end

@btime sum(multiple_return_types() for _ in 1:1000)
@btime sum(single_return_type() for _ in 1:1000)
```

```
  5.153 μs (0 allocations: 0 bytes)
  828.424 ns (0 allocations: 0 bytes)
```

```
1493.0
```

Note that having heterogeneous types as above is not the same thing as **type instability**, which is when Julia cannot determine in advance what the data types will be. In the example above, the return type is not unstable: the compiler recongizes that the single parametrice type `Union{Float64,Int64}` will be returned., even though two different sitypes can be returned the When the types cannot be determined by the compiler, it leads to runtime dispatch.

The following function illustrates a common anti-pattern wherein an `Any` typed array is created and then elements are added to it. Because any type can be added to an `Any` array (we happen to just add floats to it) then Julia's not sure what types to expect inside the container and therefore has to determine it at runtime.

```julia
function unstable_function()
    values = [] # Implicitly Any[]
    push!(values, 1.0, 2.0)
    maximum(values)
end
```

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

Employ Julia's `@code_warntype` macro to identify type instabilities in your code. Code that cannot be inferred will be annotated with red text showing the point of the code that will return unstable types. In this Quarto document, the informative colors will not render with the way it does in a coding environment, but look for the `::Any` annotation which indicates the compiler could not identify the type to return.

```julia
@code_warntype unstable_function()
```

```
MethodInstance for unstable_function()
  from unstable_function() @ Main In[9]:1
Arguments
  #self#::Core.Const(Main.unstable_function)
Locals
  values::Vector{Any}
Body::Any
1 ─       (values = Base.vect())
│   %2  = Main.push!::Core.Const(push!)
│   %3  = values::Vector{Any}
│         (%2)(%3, 1.0, 2.0)
│   %5  = Main.maximum::Core.Const(maximum)
│   %6  = values::Vector{Any}
│   %7  = (%5)(%6)::Any
└──       return %7
```

More advanced exploration of type inference can be had via Cthulhu.jl[1], where you can interactively (via the REPL) navigate through different lines of code to explore inferred code.

When working with parametric types, look to avoid usage of generic type parameters (e.g. `Array{Any}`) whenever possible. For custom types, make use of parametric types to create type-stable abstractions. For example, the latter `struct Bond2` or `struct Bond3` will allow Julia to create distinct concrete types and methods as opposed to needing

[1]So-named for the "slow descent into madness" when descending into functions to follow the Julia compiler's type inference across many layers of function calls.

generic runtime dispatch due to the unpredicable potential types of the `struct` fields. The difference between `Bond2` and `Bond3` is that the fields in `Bond3` could be any type (such as `Float16` or `String`) as long as both of them equaled the type variable `T`.

```
struct Bond1
    par
    coupon
end

struct Bond2
    par::Float64
    coupon::Float64
end

struct Bond3{T}
    par::T
    coupon::T
end
```

### 9.3.5. Optimize for Branch Prediction

Modern CPUs use branch prediction to speculatively execute instructions before knowing the outcome of conditional statements. Optimizing your code for branch prediction can significantly improve performance, especially in tight loops or frequently executed code paths.

To optimize for branch prediction:

1. Structure your code to make branching patterns more predictable. For instance, in if-else statements, put the more likely condition first. This allows the CPU to more accurately predict the branch outcome.

2. Use loop unrolling to reduce the number of branches. This technique involves manually repeating loop body code to reduce the number of loop iterations and associated branch instructions. See Section 10.4 for more on what this means.

3. Consider using Julia's `@inbounds` macro to eliminate bounds checking in array operations when you're certain the accesses are safe. This reduces the number of conditional checks the CPU needs to perform.

4. For performance-critical sections with unpredictable branches, consider using branch-free algorithms or bitwise operations instead of conditional statements. This can help avoid the penalties associated with branch mispredictions.

5. In some cases, it may be beneficial to replace branches with arithmetic operations (e.g., using the ternary operator or multiplication by boolean values) to allow for better vectorization and reduce the impact of branch mispredictions.

Here's an example demonstrating the impact of branch prediction:

```julia
function sum_if_positive(arr)
    sum = 0.0
    for x in arr
        if isodd(x)
            sum += x
        else
            sum -= x
        end
    end
    sum
end

# Benchmark
arr = rand(Int, 1_000_000)
arr_mostly_odds = fill(3, 999_999)
push!(arr_mostly_odds, 2) # add one even to get to 1M elements
@btime sum_if_positive($arr)
@btime sum_if_positive($arr_mostly_odds);
```

```
  736.291 μs (0 allocations: 0 bytes)
  686.375 μs (0 allocations: 0 bytes)
```

In this example, having consistently seen odd numbers means that the CPU will predict that the branch that will be used is the `sum += x` branch of the `if` statement.

Remember that optimizing for branch prediction often involves trade-offs. The benefits can vary depending on the specific hardware and the nature of your data. If performance critical, profile your code to ensure that your optimizations are actually improving performance in your specific use case. Overoptimizing on one set of hardware (e.g. local computer) may not translate the same on another set of hardware (e.g. server deployment).

### 9.3.6. Further Reading

- What scientists must know about hardware to write fast code
- Optimizing Serial Code, SciML Book

# 10. Parallelization

Quantity has a quality all its own. - Attributed to Vladimir Lennin

## 10.1. In this section

Fundamentals of parallel workloads, different mechanisms to distribute work: vectorization, multi-threading, GPU, and multi-device workflows. Different programming models: map-reduce, arrays, and tasks.

## 10.2. Amdahl's Law and the Limits of Parallel Computing

An important ground-truth in computing is that there is an upper limit to how fast a workload can be sped up through distributing the workload among multiple processor units. For example, if there is a modeling workload wherein 90% of the work is independent (say policy or asset level calculations) and the remaining 10% of the workload is an aggregate (say company or portfolio level), then the theoretical maximum speedup of the process is 10x faster (1 / 90% parallelizable load). This is captured in a law known as **Amdahl's Law** and it reflects the *theoretical* maximum speedup a workload could see. In practice, the speedup is worse than this due to overhead of moving data around, scheduling the tasks, and aggregating results. This is why in many cases a good effort in sequential workloads (see @#sec-performance-single) is often a more fruitful effort than trying to parallelize some workloads.

That said, there are still many modeling use-cases for parallelization. Modern investment and insurance portfolios can easily contain 100's of thousands or millions of seriatim holdings. In many cases, these can be evaluated independently, though on the often times there is interaction with the total portfolio (contract dividends, non-guaranteed elements, profit sharing, etc.). Further, even if the holdings are not parallelizable across the holdings dimension, we are often interested in independent evaluations across economic scenarios which is amendable to parallelization.

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

Where:

- $S(n)$ is the theoretical speedup of the execution of the whole task
- $n$ is the number of processors
- $p$ is the proportion of the execution time that benefits from improved resources

We can visualize this for different combinations of $p$ and $n$ in Figure 10.1.

```
using CairoMakie

function amdahl_speedup(p, n)
    return 1 / ((1 - p) + p / n)
end

function main()
    fig = Figure(size=(800, 600))
    ax = Axis(fig[1, 1],
        title="Amdahl's Law",
        xlabel=L"Number of processors ($n$)",
        ylabel="Speedup",
        xscale=log2,
        xticks=2 .^ (0:16),
        xtickformat=x -> "2^" .* string.(Int.(log.(2, x))),
        yticks=0:2:20
    )

    n = 2 .^ (0:16)
    parallel_portions = [0.5, 0.75, 0.9, 0.95]
    linestyles = [:solid, :dash, :dashdot, :solid]

    for (i, p) in enumerate(parallel_portions)
        speedup = [amdahl_speedup(p, ni) for ni in n]
        lines!(ax, n, speedup, label="$(Int(p*100))%", linestyle=linestyles[i])
    end

    xlims!(ax, 1, 2^16)
    ylims!(ax, 0, 20)

    axislegend(ax, L"Parallel portion ($p$)", position=:lt)
    fig
end

main()
```

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

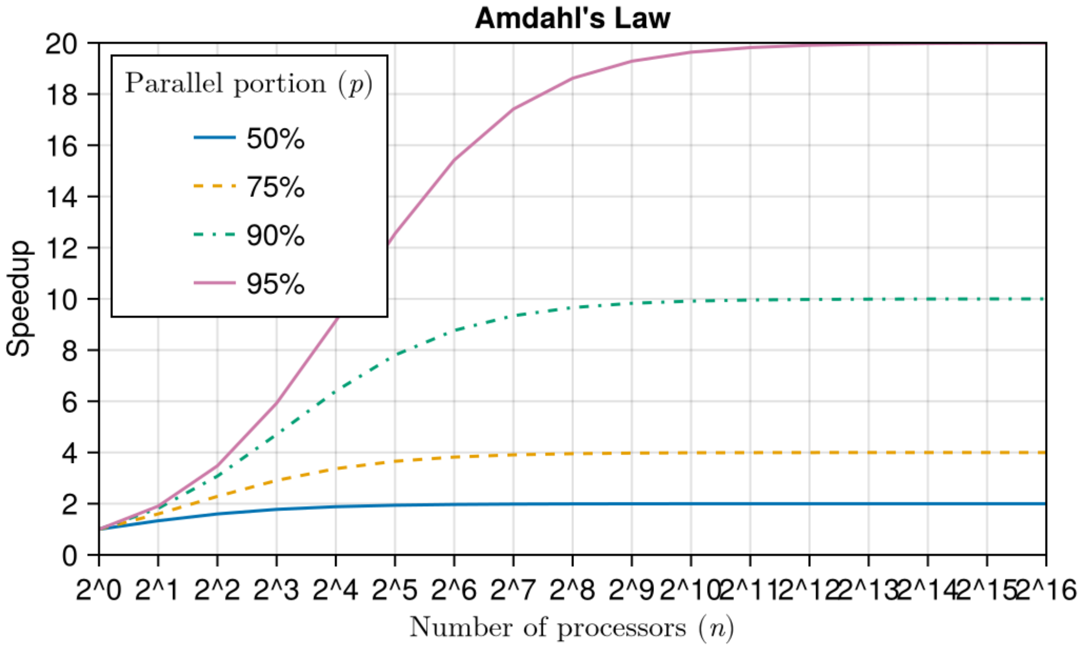



Figure 10.1.: Theoretical upper bound for speedup of a workload given the parallelizable portion $p$ and number of processors $n$.

With this understanding, we will be able to set expectations and analyze the benefit of parallelization.

## 10.3. Types of Parallelism

Parallel processing comes in different flavors and is related to the details of hardware as discussed in Chapter 8. We will necessarily extend the discussion of hardware here, as parallelization is (mostly) inextricably tied to hardware details (we will revisit this in Section 10.8).

Table 10.1.: Major types of computational parallelism highlighting their key characteristics, advantages, and potential drawbacks.

| Type | Description | Strengths | Weaknesses |
| --- | --- | --- | --- |
| Vectorization (SIMD) | Performs same operation on multiple data points simultaneously | Efficient for data-parallel tasks, uses specialized CPU instructions | Limited to certain types of operations, data must be contiguous |
| Multi-Threading | Executes multiple threads concurrently on a single CPU | Good for task parallelism, utilizes multi-core processors effectively | Overhead from thread management, potential race conditions |
| GPU | Uses graphics processing units (GPUs) for parallel computations | Excellent for massively parallel tasks, high throughput | Specialized programming required, data transfer overhead |
| Multi-Device / Distributed | Spreads computation across multiple machines or devices | Scales to very large problems, can use heterogeneous hardware | Complex to implement and manage, network latency issues |

## 10.4. Vectorization

**Vectorization** in the context of parallel processing refers to special circuits within the CPU wherein the CPU will load multiple data units (e.g. 4 or 8 floating point numbers) in a contiguous block and perform the same instruction on them at the same time. This is also known as **SIMD, or Single-Instruction Multiple Data**.

The requirements for SIMD-able code are that:

- The intended section for SIMD is inside the inner-most loop.
- There are no branches (if-statements) inside the loop body.
  - Indexing an array is actually a possible branch, as two cases could arise: the index is inbounds or out-of-bounds. To avoid this, either use `for x in collection`, `for i in eachindex(collection)` or `for i in 1:n; @inbounds collection[i]` though the last of these is discouraged in favor of ther prior, safer options.
-

```
using BenchmarkTools

function prevent_simd(arr)
    sum = 0
    for x in arr
        if x > 0
            sum += x
        end
    end
    return sum
end

function allow_simd(arr)
    sum = 0
    for x in arr
        sum += max(x, 0)
    end
    return sum
end

let
    x = rand(10000)

    @btime prevent_simd($x)
    @btime allow_simd($x)
end
```

```
  32.000 μs (0 allocations: 0 bytes)
  44.333 μs (0 allocations: 0 bytes)
```

```
4987.843821085504
```

In testing the above code, the `allow_simd` version should be several times faster than the `prevent_simd` example. The reason is that `prevent_simd` has a branch (`if x > 0`) where the behavior of the code may change depending on the value in `arr`. Conversely, the behavior of `allow_simd` is always the same in each iteration, no matter the value of `x`. This allows the compiler to generate vectorized code automatically.

Note that the compiler is able to identify vectorizable code in many cases, though through some cases may benefit from a more manual hint to the compiler through macro annotations (see `?@simd` for details).

Other types of parallelism that we will discuss in this chapter have some risk of errors or data corruption if not used correctly. SIMD isn't prone to issues like this because if the code is not SIMD-able then the compiler will not auto-vectorize the code block.

### 10.4.1. Hardware

Vectorization is hardware dependent. If the CPU does not support vectorization you will not see speedups from it. Many consumer and professional chips have AVX2 (Advanced Vector Extensions, with the 2 signifying second-generation 256 bit width, allowing four simultaneous 64-bit operations). The next generation is AVX512, having twice the SIMD capacity as AVX2. However, as of 2025 most consumer chips do not yet have that and commercial chips may not actually be faster than the AVX2 due to thermal restrictions (SIMD uses more power and generates more heat).

### 10.4.2. Additional Packages

Some additional packages to be aware of include:

- LoopVectorization.jl which can enhance the vectorized loops even further, such as handling the "tail" of a vectorized loops more efficiently than the base compiler. The "tail" refers to situations like where you have a vector width of 8, but don't have a collection that's a nice multiple of 8 (say 1001 elements).
- Octavian.jl implements a linear algebra-like library, utilizing parallelism via vectorization to generate efficent code for the system it's running on.
- Tulio.jl is an einsum library, a domain-specific language for tensor (a specific subset of vectors) operations, common in machine learning and linear algebra.

## 10.5. Multi-Threading

### 10.5.1. Tasks

To understand mulithreading examples, we first need to discuss **Tasks**, which are chunks of computation that get performed together, but after which the computer is free to switch to a new task. Technically, there are some instructions within a task that will let the computer pause and come back to that task later (such as `sleep`). Tasks are do not, by themselves, allow for multiple computations to be performed in parallel. For example, one task might be loading a data file from persistent storage into RAM. After that task is complete, the computer continues on with another task in the queue (rendering a web page, playing a song, etc.). In this way even with a single processor and core, a computer could be "doing multiple things at once" (or "multi-tasking") even though nothing is running in parallel. The scheduling of the tasks is handled automatically by Julia and the operating system.

Here's an example of a couple of tasks where we write to an array. The second task actually writes to the array first, since we asked the first task to sleep (which allows the computer to yield to other tasks in the queue)[1].

```
let
   shared_array = zeros(5)

   task1 = @task begin
        sleep(1)
       shared_array[1] = 1

       println("Task 1: ", shared_array)
   end

   task2 = @task begin
       shared_array[2] = 2
       println("Task 2: ", shared_array)
   end

   schedule(task1); schedule(task2)
   wait(task1)
   wait(task2)

   println("Main: ", shared_array)
end
```

```
Task 2: [0.0, 2.0, 0.0, 0.0, 0.0]
Task 1: [1.0, 2.0, 0.0, 0.0, 0.0]
Main: [1.0, 2.0, 0.0, 0.0, 0.0]
```

#### 10.5.1.1. Channels

**Channels** are a way to communicate data in an ordered way between tasks. You specify a type of data that the buffer will contain and how many elements it can hold. It then stores items (via `put!`) in a first-in-first-out (FIFO) queue, which can be popped off the queue (via 'take!) by other tasks.

[1]Technically, it's possible that the second task doesn't write to the array first. This could happen if there's enough tasks (from our program or others on the computer) that saturate the CPU during the first task's `sleep` period such that the first task gets picked up again before the second one does.

Here's an example of a system which generates trades in the financial markets at random time intervals, and a monitoring tasks takes the results and tabulates running statistics:

```
let

    function trade_producer(channel,i)
            sleep(rand())                                                        ①
            profit = randn()                                                     ②
            put!(channel, profit)
            println("Producer: Trade Result #$i $(round(profit, digits=3))")
    end

    function portfolio_monitor(channel,n)
        sum = 0.0
        for _ in 1:n                                                             ③
            profit = take!(channel)
            sum += profit
            println("Monitor: Received $(round(profit, digits=3)), Cumulative profit: $(round(
        end
    end

    channel = Channel{Float64}(32)                                               ④

    # Start producer and consumer tasks
    @sync begin                                                                  ⑤
        for i in 1:5; @async trade_producer(channel,i); end                      ⑥
        @async portfolio_monitor(channel,5)
    end


    # Close the channel and wait for tasks to finish
    close(channel)
end
```

① Random sleep between 0 and 1 seconds
② Generate a random number from standard normal distribution to simulate profit or loss from a trade
③ In this teaching example, we've limited the system to produce just five "trades". In practice, this could be kept running indefinitely via, e.g., `while true`.
④ Create a channel with a buffer size of 32 floats (in this limited example, we could have gotten away with just 5 since that's how many the demonstration produces). In practice, you want this to be long enough that the consumer of the channel never

gets so far behind that the channel fills up. The channel is created outside of the `@sync` block so that `channel` is in scope when we `close` it.

⑤ `@sync` waits (like `wait(task)`) for all of the scheduled tasks within the block to complete before proceeding with the rest of the program.

⑥ `@async` does the combination of creating a task via `@task` and `schedule`-ing in one, simpler call.

```
Producer: Trade Result #5 1.781
Monitor: Received 1.781, Cumulative profit: 1.781
Producer: Trade Result #4 -1.727
Monitor: Received -1.727, Cumulative profit: 0.054
Producer: Trade Result #3 0.547
Monitor: Received 0.547, Cumulative profit: 0.6
Producer: Trade Result #2 0.935
Monitor: Received 0.935, Cumulative profit: 1.535
Producer: Trade Result #1 -0.779
Monitor: Received -0.779, Cumulative profit: 0.756
```

This is really useful for handling events that are "external" to our program. If we were just doing a modelign excersise using static data, then we could control the order of processing and not need to worry about monitoring a volatile source of data. Nonetheless, tasks can still be useful in some cases even if a model is not using "live" data: for example if one of the steps in a model is to load a very large dataset, it may be possible to perform some computations while chunked task requests are queued to load more data from the disk.

while Julia's garbage collector will eventually clean up unclosed channels, it's a good practice to explicitly `close` them to ensure proper resource management, clear signaling of completion, and to avoid potential blocking or termination issues in your programs.

An additional thing to be aware of: if the task never finishes properly inside the `@sync`, then your program may get stuck in an infite loop and hang. Such as if one of the tasks never has a termination conidtion such as an upper bound on a loop, or a clear way to `break` out of a `while true` loop. While not different than a normal loop, such issues become less obvious underneath the layer of task abstractions.

The key takeaway for tasks is that it's a way to chunk work into bundles that can be run in a concurrent fashion, even if nothing is technically being processed in parallel. The multi-threading and parallel programming paradigms sections build off of tasks so an understanding of tasks is helpful. However, some of the higher level libraries hide the task-based building blocks from you as the user/developer and so an intricate understanding of tasks is not required to be successful in parallelizing your Julia code.

### 10.5.2. Multi-Threading Overview

When a program starts on your computer, a **process** is created which is where the operating system allocates some overhead items (keeping track of the the code and memory allocations and layout) and block of memory in RAM that can be utilized by that process. Different processes do not have access to each other's allocated memory.

> **Note**
>
> Readers may be familar with starting Excel in different processes. When Microsoft Excel is opened multiple times, in different processes, then the workbooks in each respective process do not share memory and cannot create links or use full copy/paste functionality between them. It's only when workbooks are opened within the same process that the workbooks may seamslessley talk to each other.

Within each process, a main thread is created. That thread is where the running of the code occurs. For the level of the discussion here, you can mainly think of a process as a container with shared memory for threads, which do the real work (as illustrated in Figure 10.2). Besides the main thread, other threads can be created within the process and access the same shared memory.

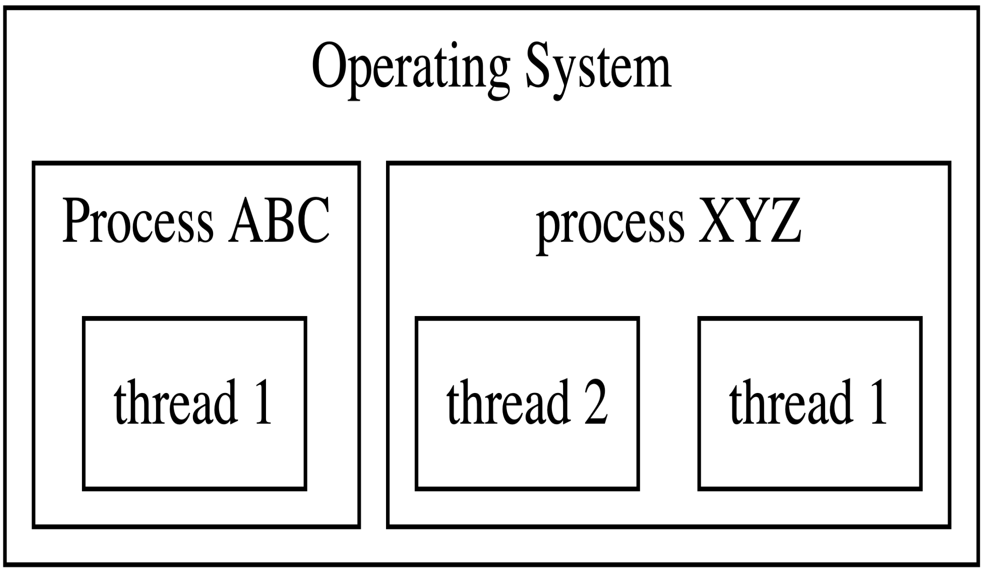



Figure 10.2.: When a program starts, the operating system creates a process for which multiple threads (a *main* thread plus optional additional threads) share memory.

The advantage of threads is that within a single physical processor, there may be multiple cores. Those cores can access the shared process memory and run tasks from different threads simultaneously. This is a technique that takes advantage of modern processor architecture wherein several (sometimes as many as 32 or more) cores exist on the same chip.

**Note**

Technically, there are different flavors of threading. While not critical for the understanding and modeling-focused dicsussion here, for completeness here is a bit more detail on different thread types.

- Recall that **Tasks** are chunks of computation that get performed together, but after which the computer is free to switch to a new task. For example, one task might be loading a data file from persistent storage into RAM. After that task is complete, the computer continues on with another task in the queue (rendering a web page, playing a song, etc.). In this way even with a single processor and core, a computer could be "doing multiple things at once" (or "multi-tasking") even though nothing is running in parallel.
- **Operating System Threads** or just **Threads** are managed (as the name implies) at the operating system level. The benefit to this is that operating system level threads have more power: the operating system can pause or limit throughput on running programs if the operating system needs the resources for something it deems higher priority. It's technically possible to use this power to force a higher priority for your own code, but Julia and many other languages do not offer creating of these types of threads in favor of the next type of threads. Operating system threads have a higher amount overhead (time and memory) involved in creating and destroying the threads.
- **Green threads**, **cooperative threads**, **fibers**, or **user-threads** are the type of threads that Julia provides. They are managed at the process (Julia) level and don't have as much overhead in their creation as operating system threads. Also in Julia, a thread is implemented via Tasks

Parallelism in modern computing comes in many flavors, occurs at many different levels (hardward, OS, software, network), and has many different implementations of similar concepts. The terminology of threading in practice and online documentation is confusing and prone to confusion. If you are having a discussion or asking a question, feel free to take the time to ask for clarification on the terminology being used at a given point in time.

> **!** Important
>
> To use multi-threading, Julia needs to be started with more than one thread. This can be done by either setting the environment variable `JULIA_NUM_THREADS` to either `auto` or specify a number like `4`. You can also specify how many threads to start julia with if given the `-t` command line argument (such as running `julia -t 4` to start Julia with four threads from the command line). For the examples in this book and how many threads are used, see **?@sec-colophon** for the system settings. Why doesn't it automatically start with more than one thread? Between "hyper-threading" (synthetic additional thread capacity), multi-core architectures, and the different types of threads it's actually difficult to predict how many threads will be optimal for a given system. Julia's current default is to take the more conservative approach and start single-threaded unless otherwise specified. The "auto" option is a best-guess but can, on certain systems and configurations, be very bad for performance. The authors recommend for most common systems to just use "auto".

#### 10.5.2.1. Multi-Threading Pitfalls

Different threads being able to access the same memory is a double-edged sword. It is useful because we do not need to create multiple copies of the data in RAM or in the cache[2] and can improve the overall throughput of our usually memory-bandwidth-limited machines. The downside is that if we are mutating the shared data for which our program relies upon, then our program may produce unintended results if the modification occurs carelessly. There are a couple of related issues to be aware of:

##### 10.5.2.1.1. Race Conditions

The first issue is known as a **race condition**, which occurs when a block of memory has been read from or written to in an unintended order. For example, if we have two threads which are accumulating a sub-total, each process may read the running sub-total before the other thread has finished it's update.

In the following example, we use the `Threads.@threads` to tell Julia to automatically distribute the work across threads.

```
function sum_bad(n)
    subtotal = 0
    Threads.@threads for i in 1:n
```

[2]There are some chips which do not have access to the same memory in a multi-threading context, and are known as non-uniform memory access (NUMA). These architectures work more like those in Section 10.7.

```
        subtotal += i
    end
    subtotal
end

sum_bad(100_000)
```

```
1562512500
```

The result of this should be that `subtotal` doesn't equal `5000050000` because the different threads pick up the subtotal before the other thread finishes adding it's current value. As a result, when one of the threads finishes it's work another may be starting from a place that ignores the updated value.

#### 10.5.2.2. Avoiding Multi-threading Pitfalls

We will cover several ways to manage multi-threading race conditions, but it is the recommendation of the authors to primarily utilize higher level library code, which will be demonstrated after covering some of the more basic, manual techniques.

##### 10.5.2.2.1. Chunking up work into single-threaded work

In this example, we can correct `sum_bad` by splitting the work into different threads, each of which is independent. Then, we can aggregate the results of each of the chunks.

```
function sum_single(a)
    s = 0
    for i in a
        s += i
    end
    s
end
@btime sum_single(1:100_000)
```

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

```
5000050000
```

Note that in the single-threaded case, Julia is able to identify this common pattern and use a shortcut, calculating the sum of the integers 1 through $n$ as $\frac{n(n+1)}{2}$.

```
function sum_chunker(a)

chunks = Iterators.partition(1:a, a ÷ Threads.nthreads())
           tasks = map(chunks) do chunk
               Threads.@spawn sum_single(chunk)
           end
           chunk_sums = fetch.(tasks)
           return sum_single(chunk_sums)

end

@btime sum_chunker(100_000)
```

```
  1.133 μs (34 allocations: 2.52 KiB)
```

```
5000050000
```

#### 10.5.2.2.2. Using Locks

**Locks** prevent memory from being accessed from more than one thread at a time.

```
function sum_with_lock(n)
    subtotal = 0
    lock = ReentrantLock()
    Threads.@threads for i in 1:n
        Base.@lock lock begin
            subtotal += i
        end
    end
    subtotal
end
@btime sum_with_lock(100_000)
```

```
  6.116 ms (199483 allocations: 3.05 MiB)
```

```
5000050000
```

#### 10.5.2.2.3. Using Atomics

**Atomics** are certain primitive values with a reduced set of operations for which Julia and the compiler can automatically create thread-safe code. This is often significantly faster than the context-switching overhead needed with locking and unlocking memory for threaded tasks. Compared with locks, atomics are simpler to implement and easier to reason about. The downside is that atomics are limited to the available primitive atomics types and methods.

```julia
function sum_with_atomic(n)
    subtotal = Threads.Atomic{Int}(0)
    Threads.@threads for i in 1:n
        Threads.atomic_add!(subtotal, i)
    end
    subtotal[]
end

@btime sum_with_atomic(100_000)
```

```
  480.541 μs (22 allocations: 2.16 KiB)
```

```
5000050000
```

## 10.6. GPU and TPUs

### 10.6.1. Hardware

Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs) are hardware accellerators for massively parallel computations. A TPU is very similar to a GPU but have special ability to handle data types and instructions that are more specialized for linear algebra operations; going forward we will simply refer to these types of accellerators as GPUs.

GPUs have similar components as the CPU as discussed in Chapter 8. They have RAM, caches for the cores, and cores that run the coded instructions on the data. The differences from a CPU are primarily:

- A GPU typically has thousands of cores while a CPU generally has single or double digit cores.
  - The cores typically operate at a *slower* clock speed than CPUs, relying on the sheer number of cores to perform computations faster.

- The GPU cores essentially have to be running the same set of instructions on all of the data, not unlike vectorization (Section 10.4).
  - GPU code is not suited for code with branching conditions (e.g. `if` statements) and so is more limited in the kinds of computations it can handle compared to the CPU.
- The RAM is typically much more constrained, typically less than a quarter of what primary RAM might be.
  - As a result, GPUs may need strategies to move chunks of data to and from the GPU memory for moderately large datasets. Further, it's actually fairly common to use a lower-precision datatype (e.g. `Float16` or `Float32`) to improve overall program throughput at the cost of some precision.
- The caches are similar in concept to CPU, but unlike most CPU caches, the there is locality to data wherein core #1 will have much quicker access to a different subset of data than, say, core #1024.
- A GPU is usually a secondary device of sorts: it physically and in device architecture is separate from the CPU which remains in charge of overal computer execution. To some extent, this is changing with some of the latest computer hardware. For example, the M-series of Apple processors have the CPU, GPU, and RAM in a single tightly integrated package for efficiency and computational power.
  - The implication of this (as with any movement of memory) is that there is overhead to moving data to and from the GPU. Your calculations will need to be in the singlke milliseconds range of time in order to start to see benefit from utilizing a GPU.

#### 10.6.1.1. Notable Vendors and Libraries

Like the difference between x86 and ARM architectures, GPU also have specific architectures which vary by the vendor. To make full use of the hardware, the vendors need to (1) provide device drivers which allow the CPU to talk to the GPU, and (2) provide the libraries (lower level application programming interfaces, or APIs) which allow developers to utilize different hardware features without needing to write machine code. As of the mid 2020s, here are the most important GPU vendors and the associated programming library for utilizing their specific hardware:

Table 10.2.: Important GPU and TPU vendors and the associated library/interface.

| Vendor | Hardware | API Library/Package |
|---|---|---|
| NVIDIA | Geforce, GTX/RTX, various Data Center focused hardware | CUDA |
| AMD | Radeon, various Data Center focused hardware | ROCm |
| Intel | Core, Xeon, Arc processors | OneAPI |
| Apple | M Series processors | Metal |
| Google | Tensor processors | TensorFlow |

### 10.6.2. Utilizing a GPU

With some of the key conceptual differences between CPUs and GPUs explained, let's explore how to incorporate these powerful hardware accelerators.

#### 10.6.2.1. Julia GPU Libraries

There's essentially two types of GPU programming we will discuss here:

1. Array-based programming can, in Julia, be fairly easily translated into code that will run on GPUs.
2. Generated kernels wherein higher level Julia code is written but is compiled into GPU-compatible code.

> ℹ Note
>
> **Kernels** is a term used to describe explicit instructions for operating on data, as opposed to general code where the compiler can translate higher level functions into explicit instructions.

It is possible to explicitly write a GPU kernel in Julia or vendor API library (CUDA, Metal, etc.) but this is more advanced usage which will not be covered here.

Julia has wonderful support for several of the primary vendors (at the time of writing, CUDA, Metal, OneAPI, and ROCm) via the JuliaGPU organization. Installation of the required dependencies is also very straightforward and the interfaces at the array and generated kernel levels are very similar. The differences are obvious at the lower level vendor-API wrappers (which is the lower-level technique that will not be covered here).

The benefit of the consistency of the higher level libraries we will use here is that examples written for one of the types of accelerators will be largely directly translate-able to another. This is especially true for array programming, a little less so for the kernel style, and the least true for for the low level vendor-API wrapper functionality.

This book will be rendered on a Mac and therefore the examples will use Metal in order to run computational cells, however we'll show a CUDA translation for some of the examples in order to show the straight-forward nature of translating higher level GPU code in Julia is.

| GPU API | GPU Array Type | Kernel Macro |
|---|---|---|
| CUDA | `CuArray` | `@cuda` |
| Metal | `MtlArray` | `@metal` |
| oneAPI | `oneArray` | `@oneAPI` |
| ROCm | `ROCArray` | `@roc` |

#### 10.6.2.2. Array Programming on the GPU

First described in Section 5.5, array programming eschews writing loops and and instead favors initializing blocks of heap-allocated memory and filling it with data to be operated on at a single point in time. While this is often not the most efficient way to utilize CPUs, it's essentially the required style of code to utilize GPUs.

For the example below, we will calculate the present value of a series of cashflows across a number of different scenarios. An explanation of the code is given below the example.

```
using Metal

function calculate_present_values!(present_values,cashflows, discount_matrices) ①
    # Perform element-wise multiplication and sum along the time dimension
    present_values .= sum(cashflows .* discount_matrices, dims=1)        ②
end

# Example usage using 100 time periods, 100k scenarios
num_scenarios = 10^5
pvs = zeros(Float32,1,num_scenarios)
cashflows = rand(Float32, 100)                                            ③
discount_matrices = rand(Float32, 100, num_scenarios)                     ④

# copy the data to the GPU
pvs_GPU = MtlArray(pvs)
cashflows_GPU = MtlArray(cashflows)                                       ⑤
```

```
discount_matrices_GPU = MtlArray(discount_matrices)

@btime calculate_present_values!($pvs,$cashflows, $discount_matrices)
@btime calculate_present_values!($pvs_GPU,$cashflows_GPU, $discount_matrices_GPU) ⑥
```

① The function `calculate_present_values!` is written the same way as if we were just writing CPU code. Note that we are also passing a pre-allocated vector, `present_values` to store the result. This will allow us to isolate the performance of the computation, rather than including any overhead of allocating the array for the result.
② The code is broadcasted across the first dimension so that the single set of cashflows is discounted for each scenario's discount vector.
③ Metal only supports 32 bit floating point (some CUDA hardware will support 64 bit floating point)
④ Using 100 thousand scenarios for this example.
⑤ `MtlArray(array)` will copy the array values to the GPU.
⑥ Note that the data still lives on the GPU and is of the `MtlMatrix` (a type alias for a 2-D `MtlArray`).

```
[ Info: Precompiling Metal [dde4c033-4e86-420c-a63e-0dd931031962]
Precompiling SpecialFunctionsExt
  ✓ Metal → SpecialFunctionsExt
  1 dependency successfully precompiled in 5 seconds. 66 already precompiled.
[ Info: Precompiling SpecialFunctionsExt [05d8ebbe-653a-54ed-ba56-24759129d732]
┌ Warning: Module Metal with build ID fafbfcfd-c218-0d55-0001-57797256ce66 is missing from the cac
│ This may mean Metal [dde4c033-4e86-420c-a63e-0dd931031962] does not support precompilation but i
└ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing SpecialFunctionsExt [05d8e
```

```
  2.963 ms (4 allocations: 38.53 MiB)
  118.083 μs (585 allocations: 16.66 KiB)
```

```
1×100000 MtlMatrix{Float32, Private}:
 25.4794  26.5342  26.8929  30.0138  …  29.756  27.1691  29.0747  26.0403
```

The testing suggests approximately 200 times faster computation when performed on the GPU. Note however, that does not include the overhead of (1) moving the data to the GPU (in the initial `MtlArray(cashflows)` call), or (2) returning the data to the CPU (since the return type for the GPU version is `MtlArray`). We can measure this overhead by wrapping the data transfer inside another function and benchmarking it:

```
function GPU_overhead_test(present_values, cashflows,discount_matrices)
    pvs_GPU = MtlArray(present_values)
    cashflows_GPU = MtlArray(cashflows)                                    ⑤
    discount_matrices_GPU = MtlArray(discount_matrices)

    calculate_present_values!(pvs_GPU,cashflows_GPU, discount_matrices_GPU)

    Array(pvs_GPU) # convert to CPU array
end

@btime GPU_overhead_test($pvs,$cashflows,$discount_matrices)
```

```
  8.820 ms (1024 allocations: 421.60 KiB)
```

```
1×100000 Matrix{Float32}:
 25.4794  26.5342  26.8929  30.0138  …  29.756  27.1691  29.0747  26.0403
```

With the additional overhead, the computation on the GPU takes more total time than if the work were done just on the CPU. This is a very simple example, and the balance tips heavily in favor of the GPU when:

1. The computational demands are significantly higher (e.g. we were to do more calculations than just a simple multiply/divide/sum).
2. The data size grows bigger.

**Note**

The previous example can be translated to CUDA by simply exchanging `MtlArray` for `CuArray`.

**Warning**

This example again underscores that hardware paralellization is not an automatic "win" for performance. A lot of uninformed discussion around modeling performance is to simply try to get things to run on the GPU and it is often *not* the case that the models will run faster. Further, as the modeling logic gets more complex, it does require greater care to keep in mind GPU constraints (acceptable data types, memory limitations, avoiding scalar operations, data transfer between CPU and GPU, etc.). A best practice is to contemplate sequential performance and memory usage before leveraging GPU accelerators.

#### 10.6.2.3. Kernel Programming on the GPU

Another approach to GPU programing is often referred to as kernel programming, or being much more explicit about *how* a computation is performed. This is as opposed to the declarative approach in the array-oriented style (Section 10.6.2.2) wherein we specified *what* we wanted the computation to be.

The key ideas here are that we need to manually specifiy several aspects which came 'free' in the array-oriented style. The tradeoff is that we can be more fine-tuned about how the computation leverages our hardware, potentially increasing performance.

The GPU libraries in Julia abstract much of the low level programming typically necessary for this style of programming, but we still need to explicitly look at:

1. How the GPU will iterate across different cores/threads threads.
2. How many threads to utilize, the optimal number depends on the shape of the computation (long vectors, multi-dimensional arrays), memory constraints, and hardware specifics.

- GPU threads: Individual units of execution within a kernel. Each thread runs the same kernel code but operates on a different portion of the data.

3. How to chunk (group) the data to distribute the data to the different GPU threads

Our strategy for the present values example will be to distribute the work such that different GPU threads are working on different scenarios. Within a scenario, the loop is a very familiar approach: initialize a subtotal to zero and then accumulate the calculated present values.

```
function calculate_present_values_kernel!(present_values,cashflows, discount_matrices)
    idx = thread_position_in_grid_1d()                                   ①

    pv = 0.0f0                                                           ②
    for t in 1:size(cashflows, 1)
        pv += cashflows[t] * discount_matrices[t, idx]                   ③
    end

    present_values[idx] = pv                                             ④
    return nothing                                                       ⑤
end
```

① As the work is distributed across threads, `thread_position_in_grid_1d()` will give the index of the current thread so that we can index data approporiately for the work as we decide to splitit up (we've split up thw work by scenario in this example).

② Recall that we are working with `Float32` on the GPU here, so the zero value is set via the `f0` notation indicating a 32-bit floating point number.
③ The loop is across timesteps within each thread, while the thread index is tracked with `idx`.
④ The result is written to the pre-allocated array of present values, and we avoid race conditions because the different threads are working on difference scenarios.
⑤ We don't explicitly have to `return nothing` here, but it makes it extra clear that the intention of the function is to mutate the `present_values` array given to it. This mutation intention is also signaled by the `!` convention in the function name.

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

The kernel above was fairly similar to how we might write code for CPU-threaded approaches, but we now need to specify the technicals of launching this on the GPU. The `threads` defines how many independent calculations to run at a given time, and the maximum will be dependent on the hardware used. The `groups` argument defines the number of threads that share memory and synchronize results together (meaning that group will wait for all threads to finish before moving onto the next chunck of data). The push-pull here is that threads that can share data avoid needing to create duplicate copies of that data in memory, but if there is variability in how long each calculation will take, then the waiting time for synchronizing results may slow the overall computation down.

Our task utilizes shared memory of the cashflows for each thread, so through some experimentation in advance, we find that a relatively large group size of ~512 is optimal.

We bring this all together through the use of the kernel macro `@metal`:

```julia
threads = 1024
groups = cld(num_scenarios, 512)

@btime @metal threads=$threads groups=$groups calculate_present_values_kernel!(
    $pvs_GPU,
    $cashflows_GPU,
    $discount_matrices_GPU
    )
```

```
  18.625 μs (171 allocations: 4.08 KiB)
```

```
Metal.HostKernel{typeof(calculate_present_values_kernel!), Tuple{MtlDeviceMatrix{Float32, 1}, M
```

This is approximately seven times faster than the array-oriented style above, meaning that the GPU kernel version's computation is over 1000 times faster than the CPU version. However, we saw previsouly that the cost of moving the data to the GPU memory and then back to the CPU memory was the biggest time sink of all - again we'd need to have more scale in the problem to make offloading to the GPU beneficial overall.

**ℹ Note**

The Metal GPUs are able to iterate threads across three different dimensions. In the prior example, we only used one dimension and thus used `thread_position_in_grid_1d()`. If we were distributing the threads across, say, three dimensions then we would use `thread_position_in_grid_2d()`.
How do you determine how many dimensions to use? A good approach is to mimic the data you are trying to parallelize. In the example of calculating a vector of present values across 100k scenarios, that was the primary 'axis of parallelization'. If instead of a one-dimensional set of cashflows (e.g. a single asset with fixed cashflows), we had a two-dimensional set of cashflows (e.g. a portfolio of many assets), then we may find the best balance of code simplicity and performance to iterate across two dimensions of threads (but we are still limited by the same number of total available threads).

**ℹ Note**

The above example would be translated to CUDA by changing just a few things:

- The thread indexing would be `idx = threadIdx().x` instead of `i = threadIdx().x`
- The GPU arrays should be created with `CuArray` instead of `MtlArray`.
- The kernel macro would be `@cuda threads=1024 calculate_present_values_kernel!(...)` instead of `@metal threads=threads groups=groups calculate_present_values_kernel(...)`. The memory sharing and synchronizing between threads is more manual than Metal.jl, but this is not strictly necessary for our example.

THIS CODE NEEDS TO BE TESTED ON A CUDA MACHINE:

```
function calculate_present_values_kernel!(present_values,cashflows, discount_matrices)
    idx = threadIdx().x

    pv = 0.0f0
    for t in 1:size(cashflows, 1)
        pv += cashflows[t] * discount_matrices[t, idx]
```

```
    end

    present_values[idx] = pv
    return nothing
end

groups = cld(num_scenarios, 512)

@cuda threads=threads calculate_present_values_kernel!(
    pvs_GPU,
    cashflows_GPU,
    discount_matrices_GPU
    )
```

## 10.7. Multi-Processing / Multi-Device

Mutiple device, or **multi-device** computer refers to using separate groups of memory/processor combinations to accomplish tasks in parallel. This can be as simple as multiple distinct cores on within a single desktop computer, or many separate computers networked across the internet, or many processesors within a high performance cluster or a computing-as-a-service provider like Amazon Web Services or JuliaHub.

Everything discussed previously related to hardware (Chapter 8, Section 10.5, Section 10.6) continues to apply. The additional complexity is attempting to syncronize the computation across multiple sets of the same (homogenous) or different (heterogenous) hardware.

As you might imagine, approaches to multi-device computing can vary widely. Julia's approach tries to strike the balance between capability and user-friendliness and uses a primary/worker model wherein one of the processors is the main coordinator while other processors are "workers". If only one processor is started, then the main processor is also a worker processor. This main/worker approach uses a "one-sided" approach to coordination. The main worker utilizes high level calls and the workers respond, with some of the communication and handoff handled by Julia transparently from the user's perspective.

A useful mental model is the asynchronous task-based concepts discussed in Section 10.5.1, as the main worker will effectively queue up work with the worker processors. Because there may be a delay associated with the computation or the communication between the processors, the worker runs asynchronously.

| Description | Task API | Distributed Analogue |
|---|---|---|
| Create a new task | `Task()` | `@spawnat` |
| Run task asynchronously | `@async` | `@spawnat` |
| Retrieve task result | `fetch` | `fetch` |
| Wait for task completion | `@sync` | `sync` |
| Communication between tasks | `Channel` | `RemoteChannel` |

Adapting the trade producer and monitor example from above to run on multiple processors (see #sec-channels to review the base model and algorithm), we make a few key changes:

- `using Distributed` loads the Distributed standard Julia library, providing the interface for multi-processing across different hardware.
- `addprocs(n)` will add `n` worker processors (the main processor is already counted as one worker). When adding local machine processors, the processors are part of the local machine. This starts new Julia processes (you can see this in the task manager of the machine) which inherit the package environment (i.e. `Project.toml` and environment variables) from the main process; this does not occur automatically if not part of the same local machine.
  - To add processors from other machines, see the Distributed Computing section of the Julia docs.
- `myid()` is the identification number of the given processor that's been spun up.
- We use a `RemoteChannel` instead of a `Channel` to facilitate communication across processors.
- Instead of `@async`, we use `@spawnat n` to create a task for processor number `n` (or `:any` will automatically assign a processor).

See

```julia
using Distributed
let

# Add worker processes if not already added
if nworkers() == 1
    addprocs(4)  # Add 4 worker processes
end

@everywhere function trade_producer(channel, i)
    sleep(rand())
    profit = randn()
    put!(channel, profit)
    println("Producer $(myid()): Trade Result #$i $(round(profit, digits=3))")
```

```
end

@everywhere function portfolio_monitor(channel, n)
    sum = 0.0
    for _ in 1:n                                                    ③
        profit = take!(channel)
        sum += profit
        println("Monitor $(myid()): Received $(round(profit, digits=3)), Cumulative profit: $(
    end
end

function run_distributed_simulation()
    channel = RemoteChannel(() -> Channel{Float64}(32))             ④

    # Start producer and consumer tasks
    @sync begin                                                     ⑤
        for i in 1:5
            @spawnat :any trade_producer(channel, i)                ⑥
        end
        @spawnat :any portfolio_monitor(channel, 5)
    end

    # Close the channel and wait for tasks to finish
    close(channel)
end

# Run the simulation
run_distributed_simulation()
end
```

```
      From worker 2:    Producer 2: Trade Result #5 2.208
      From worker 2:    Producer 2: Trade Result #1 -0.998
      From worker 4:    Producer 4: Trade Result #3 -0.918
      From worker 3:    Monitor 3: Received 2.208, Cumulative profit: 2.208
      From worker 3:    Monitor 3: Received -0.918, Cumulative profit: 1.29
      From worker 3:    Monitor 3: Received -0.998, Cumulative profit: 0.292
      From worker 3:    Producer 3: Trade Result #2 0.18
      From worker 3:    Monitor 3: Received 0.18, Cumulative profit: 0.472
      From worker 3:    Monitor 3: Received -0.683, Cumulative profit: -0.211
      From worker 5:    Producer 5: Trade Result #4 -0.683
```

Given the similarity to the single-process version, what's the motivation for this approach? A few differences:

- In this simplified example, we are simply starting additional Julia processes on the same machine. Like a threaded approach, the work will be split across the same multi-core processor. In this context, the main difference is that the processes do not share memory.
  - Communicating across processes generally has a little bit more overhead than communicating across threads.
- If distributing across machines, not sharing memory is advantageous if using multi-processing across machines that have their own memory stores which need not compete with the main process (such as distributing chunks of large datasets).
- The worker processors don't need to be the same architecture as the main processor, allowing usage of different machines or cloud computing that is communicating with a local main process.

## 10.8. Parallel Programming Models

The previous sections have explained the different parallel programming models and how to directly utilize them to harness additional computing power. Each approach (multi-threading, GPU, distributed processing, etc.) has unique considerations and tradeoffs. These approaches in Julia are generally much more accessable to beginning and intermediate users than other languages, but admittedly still requires a decent amount of thought and care.

It is possible, if you are willing to give up some fine-grained control, to utilize some higher level approaches which look to abstract away some of the particularities of the implementation.

### 10.8.1. Map-Reduce

Map-Reduce (Section 5.4.4) operations are inherently parallelizable and various libraries provide parallelized versions of the base `mapreduce`. This is the workhorse function of many 'big data' workloads and many statistical operations are versions of `mapreduce`.

#### 10.8.1.1. Multi-Threading

##### 10.8.1.1.1. OhMyThreads

ThreadsX.jl provides the threaded versions of essential functions such as `tmap`, `tmapreduce`,`tcollect`, and `tforeach` (see Table 5.1). In most cases, the chunking and data sharing is handled automatically for you.

```julia
import OhMyThreads
@btime OhMyThreads.tmapreduce(x -> x, +, 1:100_000)
```

```
[ Info: Precompiling OhMyThreads [67456a42-1dca-4109-a031-0a68de7e3ad5]
[ Info: Precompiling InverseFunctionsUnitfulExt [f5f6e0dd-5310-5802-bcb2-1cb72ad693d4]
Precompiling AccessorsIntervalSetsExt
  ✓ Accessors → AccessorsIntervalSetsExt
  ✓ Accessors → AccessorsStaticArraysExt
 2 dependencies successfully precompiled in 1 seconds. 17 already precompiled.
[ Info: Precompiling AccessorsIntervalSetsExt [727f68c9-d1d4-5b40-b284-36502e629768]
┌ Warning: Module Accessors with build ID fafbfcfd-1d37-8bcc-0001-578b5d1ad589 is missing from the
│ This may mean Accessors [7d9f7c33-5ae7-4f3b-8dc6-eff91059b697] does not support precompilation
└ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing AccessorsIntervalSetsExt [
Precompiling AccessorsStructArraysExt
  ✓ Accessors → AccessorsStructArraysExt
  1 dependency successfully precompiled in 0 seconds. 27 already precompiled.
[ Info: Precompiling AccessorsStructArraysExt [deedf894-762e-575a-ad8b-1df4bba63293]
┌ Warning: Module Accessors with build ID fafbfcfd-1d37-8bcc-0001-578b5d1ad589 is missing from the
│ This may mean Accessors [7d9f7c33-5ae7-4f3b-8dc6-eff91059b697] does not support precompilation
└ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing AccessorsStructArraysExt [
[ Info: Precompiling AccessorsStaticArraysExt [91548973-bbcf-5128-ac3c-c8b871e934a5]
┌ Warning: Module Accessors with build ID fafbfcfd-1d37-8bcc-0001-578b5d1ad589 is missing from the
│ This may mean Accessors [7d9f7c33-5ae7-4f3b-8dc6-eff91059b697] does not support precompilation
└ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing AccessorsStaticArraysExt [
Precompiling AccessorsUnitfulExt
  ✓ Accessors → AccessorsUnitfulExt
  1 dependency successfully precompiled in 1 seconds. 22 already precompiled.
[ Info: Precompiling AccessorsUnitfulExt [0f33c9ce-b40b-5f58-839e-64dee873ac84]
┌ Warning: Module Accessors with build ID fafbfcfd-1d37-8bcc-0001-578b5d1ad589 is missing from the
│ This may mean Accessors [7d9f7c33-5ae7-4f3b-8dc6-eff91059b697] does not support precompilation
└ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing AccessorsUnitfulExt [0f33c
Precompiling BangBangChainRulesCoreExt
  ✓ BangBang → BangBangTablesExt
  ✓ BangBang → BangBangChainRulesCoreExt
```

```
  ✓ BangBang → BangBangStaticArraysExt
  3 dependencies successfully precompiled in 1 seconds. 39 already precompiled.
[ Info: Precompiling BangBangChainRulesCoreExt [47e8a63d-7df8-5da4-81a4-8f5796ea640c]
┌ Warning: Module BangBang with build ID fafbfcfd-df2f-c9ea-0001-578b4c10b0e0 is missing from the
│ This may mean BangBang [198e06fe-97b7-11e9-32a5-e1d131e6ad66] does not support precompilation b
└ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing BangBangChainRulesCoreExt
[ Info: Precompiling BangBangStaticArraysExt [a9f1882a-14fa-573e-a12d-824431257a23]
┌ Warning: Module BangBang with build ID fafbfcfd-df2f-c9ea-0001-578b4c10b0e0 is missing from the
│ This may mean BangBang [198e06fe-97b7-11e9-32a5-e1d131e6ad66] does not support precompilation b
└ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing BangBangStaticArraysExt [a
Precompiling BangBangStructArraysExt
  ✓ BangBang → BangBangStructArraysExt
  1 dependency successfully precompiled in 1 seconds. 37 already precompiled.
[ Info: Precompiling BangBangStructArraysExt [d139770a-8b79-56c4-91f8-7273c836fd96]
┌ Warning: Module BangBang with build ID fafbfcfd-df2f-c9ea-0001-578b4c10b0e0 is missing from the
│ This may mean BangBang [198e06fe-97b7-11e9-32a5-e1d131e6ad66] does not support precompilation b
└ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing BangBangStructArraysExt [d
[ Info: Precompiling BangBangTablesExt [476361b5-ac10-5c09-8bec-30d098a22a5b]
┌ Warning: Module BangBang with build ID fafbfcfd-df2f-c9ea-0001-578b4c10b0e0 is missing from the
│ This may mean BangBang [198e06fe-97b7-11e9-32a5-e1d131e6ad66] does not support precompilation b
└ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing BangBangTablesExt [476361b

  7.804 μs (31 allocations: 2.42 KiB)

5000050000
```

#### 10.8.1.1.2. ThreadsX

ThreadsX.jl is built off of the wonderful Transducers.jl package, though the latter is a bit more advanced (more abstract, but as a result more composable and powerful). ThreadsX provides threaded versions of many popular base functions. It offers a wider set of readymade threaded functions, but has a much more complex codebase. For the vast majority of threading needs, OhMyThreads.jl should be sufficent and performant. See the documentation for all of the implemented functions, but for our illustrative example:

```julia
import ThreadsX
@btime ThreadsX.mapreduce(x -> x, +, 1:100_000)
```

```
Precompiling ThreadsX
  ✓ MicroCollections
  ✓ Transducers
  ✓ Transducers → TransducersReferenceablesExt
  ✓ Transducers → TransducersOnlineStatsBaseExt
  ✓ ThreadsX
 5 dependencies successfully precompiled in 3 seconds. 72 already precompiled.
[ Info: Precompiling ThreadsX [ac1d9e8a-700a-412c-b207-f0111f4b6c0d]
┌ Warning: Module BangBang with build ID fafbfcfd-df2f-c9ea-0001-578b4c10b0e0 is missing from the
│ This may mean BangBang [198e06fe-97b7-11e9-32a5-e1d131e6ad66] does not support precompilation b
└ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing ThreadsX [ac1d9e8a-700a-41
[ Info: Precompiling MicroCollections [128add7d-3638-4c79-886c-908ea0c25c34]
┌ Warning: Module Accessors with build ID fafbfcfd-1d37-8bcc-0001-578b5d1ad589 is missing from the
│ This may mean Accessors [7d9f7c33-5ae7-4f3b-8dc6-eff91059b697] does not support precompilation
└ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing MicroCollections [128add7c
[ Info: Precompiling Transducers [28d57a85-8fef-5791-bfe6-a80928e7c999]
┌ Warning: Module Accessors with build ID fafbfcfd-1d37-8bcc-0001-578b5d1ad589 is missing from the
│ This may mean Accessors [7d9f7c33-5ae7-4f3b-8dc6-eff91059b697] does not support precompilation
└ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing Transducers [28d57a85-8fef
[ Info: Precompiling TransducersReferenceablesExt [befac7fd-b390-5150-b72a-6269c65d7e1f]
┌ Warning: Module Transducers with build ID ffffffff-ffff-ffff-0001-57910717ef25 is missing from t
│ This may mean Transducers [28d57a85-8fef-5791-bfe6-a80928e7c999] does not support precompilatio
└ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing TransducersReferenceables
```

```
  13.166 μs (218 allocations: 16.48 KiB)
```

```
5000050000
```

#### 10.8.1.2. Multi-Processing

`reduce(op,pmap(f,collection))` will use a distributed map and reduce the resulting map on the main thread. This pattern works well if each application of `f` to elements of `collection` is costly.

`@distributed (op) for x in collection; f(x); end` is a way to write the loop with the reduction `op` for which the `f` need not be costly.

The difference between the two approaches is that with `pmap`, `collection` is made available to all workers. In the `@distributed` approach, the collection is partitioned and only a subset is sent to the designated workers.

Here's an example of both of these, calculating a simple example of counting coin flips:

```
# this is a example of poor utilization of pmap, since the operation is
# fast and the overhead of moving the whole collection dominates
@btime reduce(+,pmap(x -> rand((0,1)),1:10^3))
```

```
  93.408 ms (48571 allocations: 2.03 MiB)
```

```
513
```

```
function dist_demo()
    @distributed (+) for _ in 1:10^5
        rand((0,1))
    end
end

@btime dist_demo()
```

```
  239.166 μs (300 allocations: 12.86 KiB)
```

```
50227
```

### 10.8.2. Array-Based

Array based approaches will often utilize the parallelism of SIMD on the CPU or many cores on the GPU. It's as simple as using generic library calls which will often be optimized at the compiler level. Examples:

```
let
    x = rand(Float32,10^8)
    x_GPU = MtlArray(x)
    @btime sum($x)
    @btime sum($x_GPU)
end
```

```
  6.287 ms (0 allocations: 0 bytes)
  4.404 ms (878 allocations: 23.88 KiB)
```

```
4.99998f7
```

`sum(x)` compiles to SIMD instructions on the CPU, while using the GPU array type in `sum(x_GPU)` is enough to let the compiler dispatch on the GPU type and emit efficient, parallelized code for the GPU.

Distributed array types allow for large datasets to effectively be partitioned across multiple processors, and have implementations in the DistributedArrays.jl and Dagger.jl libraries.

### 10.8.3. Loop-Based

Loops which don't have race conditions can easily become multi-threaded. Here, we have three versions of updating a collection to square the contained values:

```
let v = collect(1:10000)

    for i in eachindex(v)
        v[i] = v[i]^2
    end
    v[1:3]
end
```

```
3-element Vector{Int64}:
 1
 4
 9
```

Using multi-threading

```
let v = collect(1:10000)
    Threads.@threads  for i in eachindex(v)
        v[i] = v[i]^2
    end
    v[1:3]
end
```

```
3-element Vector{Int64}:
 1
 4
 9
```

Using multi-processing:

```
let v = collect(1:10000)
    Distributed.@distributed for i in eachindex(v)
        v[i] = v[i]^2
    end
    v[1:3]
end
```

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

For more advanced usage, including handling shared memory see Section 10.7 and Section 10.5.

### 10.8.4. Task-Based

Task based approaches attempt to abstract the scheduling and distribution of work from the user. Instead of saying how the computation should be done, the user specifies the inteded operations and allows the library to handle the workflow. The main library for this in Julia is Dagger.jl.

Effectively, the library establishes a network topology (a map of how different processor nodes can communicate) and models the work as a directed, acyclic graph (a DAG, which is like a map of how the work is related). The library is then able to assign the work in the appropriate order to the available computation devices. The benefit of this is most apparent with complex workflows or network topologies where it would be difficult to manually assign, communicate, and schedule the workflow.

Here's a very simple example which demonstrates Dagger waiting for the two tasks which work in parallel (we already added multiple processors to this environment in Section 10.7):

```
import Dagger

# This runs first:
a = Dagger.@spawn rand(100, 100)

# These run in parallel:
b = Dagger.@spawn sum(a)
c = Dagger.@spawn prod(a)
```

```julia
# Finally, this runs:
wait(Dagger.@spawn println("b: ", b, ", c: ", c))
```

```
[ Info: Precompiling Dagger [d58978e5-989f-55fb-8d15-ea34adc7bf54]
Precompiling TransducersOnlineStatsBaseExt
  ✓ Transducers → TransducersOnlineStatsBaseExt
  1 dependency successfully precompiled in 1 seconds. 73 already precompiled.
[ Info: Precompiling TransducersOnlineStatsBaseExt [6f45943c-d98e-5e8a-8912-adbe5bfabdb6]
┌ Warning: Module Transducers with build ID ffffffff-ffff-ffff-0001-57910717ef25 is missing from t
│ This may mean Transducers [28d57a85-8fef-5791-bfe6-a80928e7c999] does not support precompilatio
└ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing TransducersOnlineStatsBas
Precompiling GraphVizSimpleExt
  ✓ Dagger → GraphVizSimpleExt
  1 dependency successfully precompiled in 16 seconds. 78 already precompiled.
[ Info: Precompiling GraphVizSimpleExt [8f367522-86d3-5221-bdf5-df974a7b0ff8]
┌ Warning: Module Dagger with build ID fafbfcfd-d20d-5008-0001-579d44158cca is missing from the ca
│ This may mean Dagger [d58978e5-989f-55fb-8d15-ea34adc7bf54] does not support precompilation but
└ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing GraphVizSimpleExt [8f3675

      From worker 3:    b: 5011.745990128438, c: 0.0
```

## 10.9. References

- https://book.sciml.ai/notes/06-The_Different_Flavors_of_Parallelism/
- https://docs.julialang.org/en/v1/manual/parallel-computing/
- https://enccs.github.io/julia-for-hpc/

# Part IV.

# Interdisciplinary Concepts and Applications

This section explores concepts from related fields that enhance financial modeling. We examine how ideas from computer science, statistics, and other disciplines intersect with and improve modeling practices. Through examples, we'll uncover the theoretical foundations supporting advanced techniques. This interdisciplinary approach aims to broaden your perspective and equip you with diverse tools for tackling complex financial problems.

# 11. Applying Software Engineering Principles

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

## 11.1. In this section

Modern software engineering practices such as version control, testing, documentation, and pipelines which makes modeling more robust and automated. Data practices and workflow advice.

## 11.2. Introduction

In addition to the core concepts from computer science described so far, there's also a similar set of ideas about the *practice* and *experience* of working with a code-based workflow that makes the approach more powerful than what the code itself can do.

That is, the majority of a professional financial modeler's time is often spent *doing things other than building models*, such as testing the model's results, writing documentation, collaborating with others on the design, and figuring out how to share the model with others inside the company. This chapter covers how a code-first workflow makes each one of those responsibilities easier or more effective.

Thre are three essential topics covered in this chapter:

- **Testing** is the practice of implementing automated checks for desired behavior and outputs in the system.
- **Documentation** is the practice of writing plain English (or your local language) to compliment the computer code for better human understanding.
- **Version Control** is the systematic practice of tracking changes and facilitating collaborative workflows on projects.

Additionally, some highly related topics are covered which help bridge some of the conceptual ideas into practical implementations for your own code and models.

This chapter is organized such that the conceptual topics are covered first, and towards the end of the chapter specific tips and setup related to Julia codebases is covered. The code examples in the earlier, conceptual parts have direct analogues in other languages.

## 11.3. Testing

Testing is a crucial aspect of software engineering that ensures the reliability and correctness of code. In financial modeling, where accuracy is paramount, implementing robust testing practices is essential, and in many cases now legally required by the regulatory body or financial reporting authority. But it's just good practice regardless of requirements.

Testing looks like this:

```
@test model_output = desired_output
```

If the expression evaluates to `true`, then the test passes. If the expression is anything else (`false`, or produces an error, or `nothing`, etc.) then the test will fail.

Here is an example of modeled behavior being tested. We have a function which will discount the given cashflows at a given annual effective interest rate. The cashflows are assumed to be equally spaced at the end of each period:

```
function present_value(discount_rate, cashflows)
    v = 1.0
    pv = 0.0
    for cf in cashflows
        v = v / (1 + discount_rate)
        pv = pv + v * cf
    end
    return pv
end
```

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

We might test the implementation like so:

```julia
using Test

@test present_value(0.05, [10]) ≈ 10 / 1.05
@test present_value(0.05, [10, 20]) ≈ 10 / 1.05 + 20 / 1.05^2
```

```
Test Passed
```

The above test passes because the expression is true. However, the following will fail because we have defined the function using a certain assumption about the compounding convention (compounded once per period) as opposed to the test outcome which presumes a continuous compounding convention. Additionally, the failing test will show the stacktrace of where the error occurred.

```julia
@test present_value(0.05, [10]) ≈ 10 * exp(-0.05 * 1)
```

> **Tip**
>
> When testing results of floating point math, it's a good idea to use the approximate comparison (≈, typed in a Julia editor with by entering `\approx<TAB>`). Recall that floating point math is a discrete, computer representation of continuous real numbers. As perfect precision is not efficient, very small differences can arise depending on the specific numbers involved or the order in which the operations are applied.
> In tests (as in the `isapprox/≈` function), you can also further specify a relative tolerance and an absolute tolerance:
>
> ```julia
> @test 1.02 ≈ 1.025 atol = 0.01
> @test 1.02 ≈ 1.025 rtol = 0.005
> ```
>
> ```
> Test Passed
> ```

The testing described in this section is sort of a 'sampling' based approach, wherein the modeler decides on some pre-determined set of outputs to test and determines the desired outcomes for that chosen set of inputs. For example, testing that `2 + 2 == 4` versus testing that a positive number plus a positive number will always equal another positive number. There are some more advanced techniques that cover testing of the second kind described in Section 12.5. In practice, the type of testing discussed in this section is much more common.

#### 11.3.0.1. Test Sets

Test sets are organizational tools to group related tests together, for example:

```
@testset "Scalar Discount" begin
    @test present_value(0.05,10) ≈ 10 / 1.05
    @test present_value(0.05,20) ≈ 20 / 1.05
end
@testset "Vector Discount" begin
    @test present_value(0.05,[10]) ≈ 10 / 1.05
    @test present_value(0.05,[10,20]) ≈ 10 / 1.05 + 20 / 1.05^2
end;
```

```
Test Summary:   | Pass  Total  Time
Scalar Discount |    2      2  0.1s
Test Summary:   | Pass  Total  Time
Vector Discount |    2      2  0.0s
```

There are many more related testing facilities described in the Julia Docs, such as combining `for` loops with test sets.

### 11.3.1. Test Driven Design

**Test Driven Design** (TDD) is a software development approach where tests are written before the actual code. The process typically follows these steps:

1. Write a test that defines a desired function or improvement.
2. Run the test, which should fail since the feature hasn't been implemented.
3. Write the minimum amount of code necessary to pass the test.
4. Run the test again. If it passes, the code meets the test criteria.
5. Refactor the code for optimization while ensuring the test still passes.

TDD can be particularly useful in financial modeling as it helps clarify (1) intended behavior of the system and (2) how you think the system should work.

For example, if we want to create a new function which calculates an interpolated value between two numbers, we might first define the test like this:

```
# interpolate between two points (0,5) and (2,10)
@test interp_linear(0,2,5,10,1) ≈ (10-5)/(2-0) * (1-0) + 5
```

We've defined how it should work for a value inside the bounding $x$ values, but writing the test has us wondering... should the function error if `x` is outside of the left and right $x$ bounds? Or should the function extrapolate outside the observed interval? The answer to that depends on exactly how we want our system to work. Sometimes the point of such a scheme is to extrapolate, other times extrapolating beyond known values can be dangerous. For now, let's say we would like to have the function extrapolate, so we can define our test targets to work like that:

```
@test interp_linear(0,2,5,10,-1) ≈ (10-5)/(2-0) * (-1 - 0) + 5
@test interp_linear(0,2,5,10,3) ≈ (10-5)/(2-0) * (3 - 0) + 5
```

By thinking through what the correct result for those different functions is, we have forced ourselves to think how the function should work generically:

```
function interp_linear(x1,x2,y1,y2,x)
    # slope times difference from x1 + y1
    return (y2 - y1) / (x2 - x1) * (x - x1) + y1
end
```

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

And we can see that our tests defined above would pass after writing the function.

```
@testset "Linear Interpolation" begin
    @test interp_linear(0,2,5,10,1) ≈ (10-5)/(2-0) * (1-0) + 5
    @test interp_linear(0,2,5,10,-1) ≈ (10-5)/(2-0) * (-1 - 0) + 5
    @test interp_linear(0,2,5,10,3) ≈ (10-5)/(2-0) * (3 - 0) + 5
end;
```

```
Test Summary:        | Pass  Total  Time
Linear Interpolation |    3      3  0.0s
```

### 11.3.2. Test Coverage

Testing is great, but what if some things aren't tested? For example, we might have a function that has a branching condition and only ever test one branch. Then when the other branch is encountered in practice it is more vulnerable to having bugs because its behavior was never double checked. Wouldn't it be great to tell whether or not we have tested all of our code?

The good news is that there is! **Test coverage** is a measurement related to how much of the codebase is covered by at least one associated test case. In the following example, code coverage would flag that the `... other logic` is not covered by tests and therefore encourage the developer to write tests covering that case:

```
function asset_value(strike,current_price)
    if current_price > strike
    # ... some logic
    else
    # ... other logic
    end
```

```
end

@test asset_value(10,11) ≈ 1.5
```

From this, it's possible to determine a score, of sorts, for how well a given set of code is tested. 100% coverage means every line of code has at least one test that double checked it's behavior.

**Warning**

Testing is only as good as the tests that are written. You could have 100% code coverage for a codebase with only a single rudimentary test covering each line. Or the test itself could be wrong! Testing is not a cure-all, but does encourage best practices.

Test coverage is also a great addition when making modification to code. It can be set up such that you receive reports on how the test coverage changes if you were to make a certain modification to a codebase. An example might look like this for a proposed change which added 13 lines of code, of which only 11 of those lines were tested. The coverage percent has therefore gone down because the proportion of new lines covered by tests is $11/13 = 84\%$, which is lower than the original coverage rate of 90%.

```
@@            Coverage Diff           @@
##          original    modif    +/-   ##
==========================================
- Coverage    90.00%   89.51%   -0.49%
==========================================
  Files            2        2
  Lines          130      143      +13
==========================================
+ Hits           117      128      +11
- Misses          13       15       +2
```

### 11.3.3. Types of Tests

Different tests can emphasize different aspects of model behavior. You could be testing a small bit of logic, or test that the whole model runs if hooked up to a database. The variety of this kind of testing has given rise to various 'named' types of testing, but it's somewhat arbitrary and the boundaries between the types is sort of fuzzy.

| Test Type | Description |
|---|---|
| Unit Testing | Verifies the functionality of individual components or functions in isolation. It ensures that each unit of code works as expected. |
| Integration Testing | Checks if different modules or services work together correctly. It verifies the interaction between various components of the system. |
| End-to-End Testing | Simulates real user scenarios to test the entire application flow from start to finish. It ensures the system works as a whole. |
| Functional Testing | Validates that the software meets specified functional requirements and behaves as expected from a user's perspective. |
| Regression Testing | Ensures that new changes or updates to the code haven't broken existing functionality. It involves re-running previously completed tests. |

There are other types of testing that can be performed on a model, such as performance testing, security testing, acceptance testing, etc., but these types of tests are outside of the scope of what we would evaluate with an `@test` check. It is possible to create more advanced, mathematical-type checks and tests, which is introduced in Section 12.5.3.

**Financial Modeling Pro-tip**

Test reports and test coverage are a wonderful way to demonstrate regular and robust testing for compliance

## 11.4. Documentation

The most important part of a code for maintenance is plainly written notes for humans, not the compiler. This includes comments, reference materials, and how-to pages, among other potential forms. Julia includes several ways to facilitate documentation of a project and associated codebase.

### 11.4.1. Comments

Comments are meant for the developer to aid in understanding a certain bit of code. A bit of time-tested wisdom is that after several weeks, months, or years away from a piece of code, something that seemed 'obvious' at the time tends to become perplexing at a later time. Writing comments is as much for yourself as it is your colleagues or successors.

Here's an example of documentation with single-line comments (indicated with the preceding `#`) and multi-line comments (enclosed by `#=` and `=#`):

```julia
function calculate_bond_price(face_value, coupon_rate, years_to_maturity, market_rate)
    # Convert annual rates to semi-annual
    semi_annual_coupon = (coupon_rate / 2) * face_value
    semi_annual_market_rate = market_rate / 2
    periods = years_to_maturity * 2

    # Calculate the present value of coupon payments
    pv_coupons = 0
    for t in 1:periods
        pv_coupons += semi_annual_coupon / (1 + semi_annual_market_rate)^t
    end

    # Calculate the present value of the face value
    pv_face_value = face_value / (1 + semi_annual_market_rate)^periods

    #=
    Sum up the components:
    1. Present value of all coupon payments
    2. Present value of the face value at maturity
    This gives us the total bond price
    =#
    bond_price = pv_coupons + pv_face_value

    return bond_price
end
```

### 11.4.2. Docstrings

Docstrings are intended to be a user-facing reference and help text. In Julia, docstrings are just strings placed in front of definitions. Markdown is available (and encouraged) to add formatting within the docstring.

Here's an example with some various features of documentation shown:

```julia
"""
    calculate_bond_price(face_value, coupon_rate, years_to_maturity, market_rate) # <1>

Calculate the price of a bond using discounted cash flow method.

Parameters:
- `face_value`: The bond's par value
- `coupon_rate`: Annual coupon rate as a decimal
- `years_to_maturity`: Number of years until the bond matures
```

```
- `market_rate`: Current market interest rate as a decimal

Returns:
- The calculated bond price

## Examples: # <2>

```julia-repl
julia> calculate_bond_price(1000, 0.05, 10, 0.06)
925.6126256977221
```

## Extended help: # <3>

This function uses the following steps to calculate the bond price:
1. Convert annual rates to semi-annual rates
2. Calculate the present value of all future coupon payments
3. Calculate the present value of the face value at maturity
4. Sum these components to get the total bond price

The calculation assumes semi-annual coupon payments, which is standard in many markets.
"""
function calculate_bond_price(face_value, coupon_rate, years_to_maturity, market_rate)
    # ... function defintion as above
end
```

① The typical docstring on a method includes the signature, indented so it's treated like code in the Markdown docstring.
② It's good practice to include a section that includes examples of appropriate usage of a function and the expected outcomes.
③ The *Extended help* section is a place to put additional detail that's available on generated docsites and in help tools like the REPL help mode.

The last point, the *Extended help* section is shown when using help mode in the REPL and including an extra `?`. For example, in the REPL, typing `?calculate_bond_price` will show the docstring up through the examples. Typing `??calculate_bond_price` will show the docstring in its entirety.

### 11.4.3. Docsites

**Docsites**, or documentation sites, are websites that are generated to host documentation related to a project. With modern tooling around programming projects, a really rich set of interactive documentation can be created while the developer/modeler focuses on simple documentation artifacts.

Specifically, modern docsite tooling generally takes in markdown text pages along with the in-code docstrings and generates a multi-page site that has navigation and search.

Typical contents of a docsite include:

- A *Quickstart* guide that introduces the project and provides an essential or common use case that the user can immediately run in their own environment. This helps to convey the scope and capability of a project and showcase how to model is intended to be used.
- *Tutorials* are typically worked examples that introduce basic aspects of a concept or package usage and work up to more complex use cases.
- *Developer documentation* is intended to be read by those who are interested in understanding, contributing, or modifying the codebase.
- *Reference documentation* describes concepts and available functionality. A subset of this is API, or **Application Programming Interface** documentation which is the detailed specification of the available functionality of a package, often consisting largely of docstrings like the previous example.

At a minimum, by writing docstrings, a docsite generator can create a searchable, hierarchical page with all of the content from the docstrings in a project. This alone is incredibly beneficial for current and potential users of a project. Creating the other types of documentation as described above is mechanically as simple as creating a new markdown file, though the hard part is learning how to write quality documentation. But at least the technical and workflow aspects have been made as easy as possible!

## 11.5. Version Control

**Version control systems** (VCS) refer to the tracking of changes to a codebase in a verifiable and coordinated way across project contributors. VCS underpins many aspects of automating the mundane parts of a modeler's job. Benefits of VCS include (either directly, or contribute significantly to):

- Access control and approval processes
- Versioning of releases
- Reproducibility across space and time of a model's logic
- Continuous testing and validation of results
- Minimization of manual overrides, intervention, and opportunity for user error
- Coordinating collaboration in parallel and in sequence between one or many contributors

Among several competing options, Git is the predominant choice as of this book's writing and therefore we will focus on Git concepts and workflows.

### 11.5.1. Git Overview

Git is a free and open source software that tracks changes in files using a series of snapshots. Each commit creates a new snapshot of the entire project, storing references to the files that have changed[1]. All of this is stored in a `.git` subfolder of a project, which is automatically created upon initializing a repository.

Under the hood, Git objects (which include things such as blobs, trees, commits, and tags) form a directed acyclic graph representing the project's history. Blobs store file contents, trees represent directories, commits point to trees and parent commits, and tags provide human-readable names for specific commits. This structure allows Git to efficiently handle branching, merging, and maintaining project history.

Hashes indicate a verifiable snapshot of a codebase. For example, a full commit ID `40f141303cec3d58879c493988b71c4e56d79b90` will *always* refer to a certain snapshot of the code, and if there is a mismatch in the git history between the contents of the repository and the commit's hash then the git repository is corrupted and it will not function. A corrupted repository usually doesn't happen in practice, of course! You might see hashes shortened to the last several characters (e.g. `6d79b90`) and in the following examples we'll shorten the hypothetical hashes to just three characters (e.g. `b90`).

A codebase can be **branched**, meaning that two different version of the same codebase can be tracked and switched between. Git lends itself to many different workflows, but a common one is to designate a primary branch (call it `main`) and make modifications in a new, separate branch. This allows for non-linear and controlled changes to occur without potentially tainting the `main` branch. It's common practice to always have the `main` branch be a 'working' version of the code that can always 'run' for users, while branches contain works-in-progress or piecemeal changes that temporarily make the project unable to run.

Here's an example of what a workflow looks like in the case of trying to fix an erroneous function `present_value` which was written as part of a hypothetical `FileXYZ.jl` file. This example is trivial, but in a larger project where a 'fix' or 'enhancement' may span many files and take several days or weeks to implement this type of workflow becomes like a superpower compared to traditional, manual version control approaches. It sure beats an approach where you end up with filenames like `FileXYZ v23 July 2022 Final.jl`!

[1]Git uses a content-addressable filesystem, meaning it stores data as key-value pairs. The key is a hash of the content, ensuring data integrity and allowing efficient storage of identical content.

Table 11.2.: A workflow demonstrating branching, staging, committing, and merging in order to fix an incorrect function definition for a present value (`pv`) function. The branch name/commit ID shows which version you would see on your own computer/filesystem.

| Branch `main`, `FileXYZ.jl` file | Branch `fix_function`, `FileXYZ.jl` file | Action | Active Branch (Commit ID) |
|---|---|---|---|
| `function pv(rate,amount,time)`<br>`    amount / (1+rate)`<br>`end` | Does not yet exist | Write original function which forgets to take into account the `time`. | `main` (`...58b`) |
| ″ | `function pv(rate,amount,time)`<br>`    amount / (1+rate)`<br>`end` | Create a new branch `fix_function`.<br>**Git command:** `git branch fix_function` | `main` (`...58b`) |
| ″ | ″ | Checkout the new branch and make it active for editing.<br><br>**Git command:** `git checkout fix_function` | `fix_function` (`...58b`) |
| ″ | `function pv(rate,amount,time)`<br>`    amount / (1+rate)^time`<br>`end` | Edit and save the changed file. | `fix_function` (`...58b`) |
| ″ | ″ | "Stage" the modified file, telling git that you are ready to record ("commit") a new snapshot of the project.<br><br>**Git command:** `git add FileXYZ.jl` | `fix_function` (`...58b`) |
| ″ | ″ | Commit a new snapshot of the project by committing with a note to collaborators saying `fix: present value logic`<br>**Git command:** `git commit -m 'fix: present value logic` | `fix_function` (`...6ac`) |
| ″ | ″ | Switch back to the primary branch<br><br>**Git command:** `git checkout main` | `main` (`...58b`) |

| Branch `main` , `FileXYZ.jl` file | Branch `fix_function` , `FileXYZ.jl` file | Action | Active Branch (Commit ID) |
|---|---|---|---|
| `function pv(rate,amount,time)`<br>`    amount / (1+rate)^time`<br>`end` | ″ | Merge changes from other branch into the `main` branch, incorporating the corrected version of the code.<br><br>**Git command:** `git merge fix_function` | `main`<br>(`...b90`) |

A visual representation of the git repository and commits for the actions described in Table 11.2 might be as follows, where the `...XXX` is the shortened version of the hash associated with that commit.

```
main branch         :  ...58b      →   ...b90
                          ↓           ↗
fix_function branch :  ...58b → ...6ac
```

#### 11.5.1.1. Git Tooling

Git is traditionally a command-line based tool, however we will not focus on the command line usage as a more beginner friendly and intuitive interface is available from different available software. Git, containing a graph structure (in the sense of graph theory, not data visualization) lends itself well to better understanding when using a graphical interface for new and intermediate users.

> **i Note**
>
> Some recommend Git tools with a **graphical user interface** (GUI) :
>
> - *Github Desktop* interfaces nicely with Github and provides a GUI for common operations.
> - *Visual Studio Code* has an integrated Git pane, providing a powerful GUI for common operations.
> - *GitKraken* is free for open source repositories but requires payment for usage in enterprise or private repository environments. GitKraken provides intuitive interfaces for handling more advanced operations, conflicts, or issues that might arise when using Git.

### 11.5.2. Collaborative Workflows

Git is a distributed VCS, meaning that copies of the repository and all it's history and content can live in multiple locations, such as on two colleauges' computer as well as a server. In this distributed model, what happens if you make a change locally?

Git maintains a local repository on each user's machine, containing the full project history. This local repository includes the working directory (current state of files), staging area (changes prepared for commit), and the .git directory (metadata and object database). When collaborating, users push and pull changes to and from remote repositories. Git uses a branching model, allowing multiple lines of development to coexist and merge when ready.

**Note**

By convention, the primary branch of a project is named the `main` branch. Historically, this was called the `master` branch (after the Master-Slave technology terms) but due to connotations of human slavery most software will now default to the `main` name. However, many projects continue to use or have never renamed their original `master` branch.

#### 11.5.2.1. Pull Requests

Layered onto core Git functionality, services like Github provide interfaces which enhance the utility of VCS. A major example of this is **Pull Requests** (or PRs), which is a process of merging git branches in a way that allows for additional automation and governance.

The following is an example of a PR on a repository adding a small bit of documentation to help future users. We'll walk through several elements to describe what's going on:

Referencing Figure 11.1, several elements are worth highlighting. In this pull request, the user (and author of this change), `alecloudenback`, is proposing to modify the `QuartoNotebookRunner` repository, which is not a repository for which the user `alecloudenback` has any direct rights to modify. After having made a copy of the repository, creating a new branch, making modifications, and committing those changes... a pull request has been made to modify the primary repository.

- All changes are recorded using Git, keeping track of authorship, timestamps, and history.
- At the top of the screenhsot, the title "Note that Quarto..." allows the author to summarize what is changed in the branch to be merged.
- The "Conversation" tab allows for additional details about the change to be discussed.
- In the top right, a `+1 -1 ■■□□□` is an indication of what's changed. In this case, a single line of code (documenation, really) was removed (`-1`) and replaced with another (`+1`)
- Not shown in the Figure 11.1, the "Files Changed" tab shows a file-by-file and line-by-line comparison of what has changed (see Figure 11.2).
- The reviewer (user `MichaelHatherly`) is a collaborator with rights to modify the destination repository has been assigned to review this change before merging it into the `main` branch.
- "CodeCoverage" was discussed above in testing, and in this case tests were automatically run when the PR was created, and the coverage indicates that there was no added bit of code that was untested.

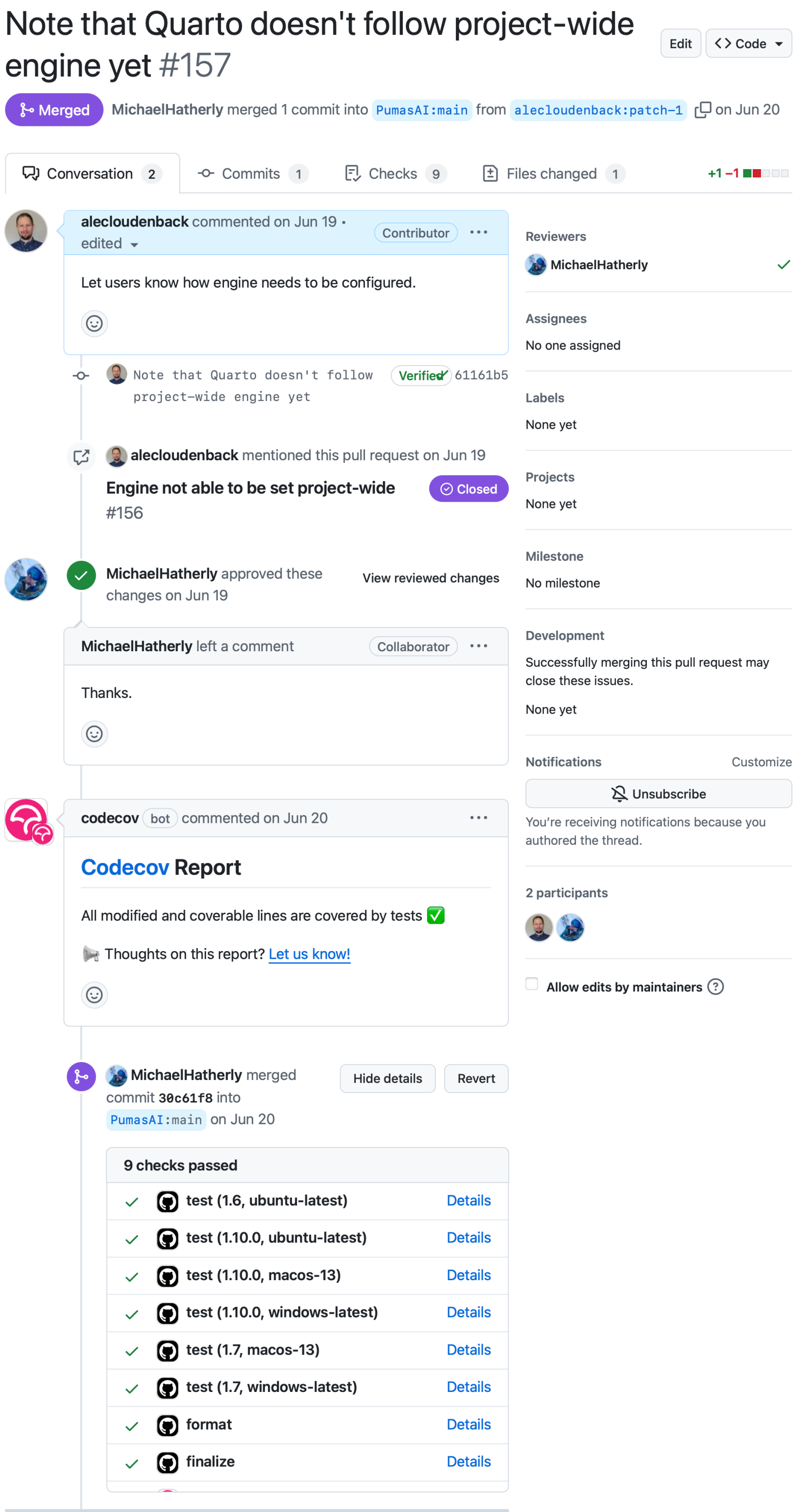



Figure 11.1.: A Pull Request on Github, demonstrating several utilities which enhance change management, automation, and governance.

- A section of "9 checks" that pass, which validated that tests within the repository still passed, on different combinations of operating systems and Julia versions. Additionally, the repository was checked to ensure that formatting conventions were followed.
- Additionally, there are a number of project management features not really showcased here. A few to note:
  - The cross-referencing to another related issue in a different repository (the reference to issue #156).
  - Assignees (assigned doers), labels, milestones, and projects are all functionality to help keep track of codebase development.

The features described here can take modelers many manhours of time - testing, review of changes, sign-off tracking, etc. This is such a powerful paradigm that should be adopted within the financial industry and especially amongst modelers.

### 11.5.3. Data Version Control

Git is not well suited for large files that change regularly (images, videos, datasets). Instead, the approach is to combine git with a **data version control** tool. These tool essentially replace the data (often called binary data or a blob) with a content hash in the git snapshots. The actual content referred to by the hash is then located elsewhere (e.g. a hosted server).

::: {#note-content-hash callout-note} ## Content Hashes

**Content hashes** are the output of a function that transforms arbitrary data into a string of data. Hashes are very common in cryptography and in areas where security/certainty is important. Eliding the details of how they work exactly, what's important to understand for our purposes is that a content hash function will take arbitrary bits and output the same set of data each time the function is called on the original bits.

For example, it might look something like this:

```julia
data = "A data file contents 123"
Base.hash(data)
```

```
0x7a1f7a6e6f61fa64
```

If it's run again on the same inputs, you get the same output:

```julia
Base.hash(data)
```

```
0x7a1f7a6e6f61fa64
```

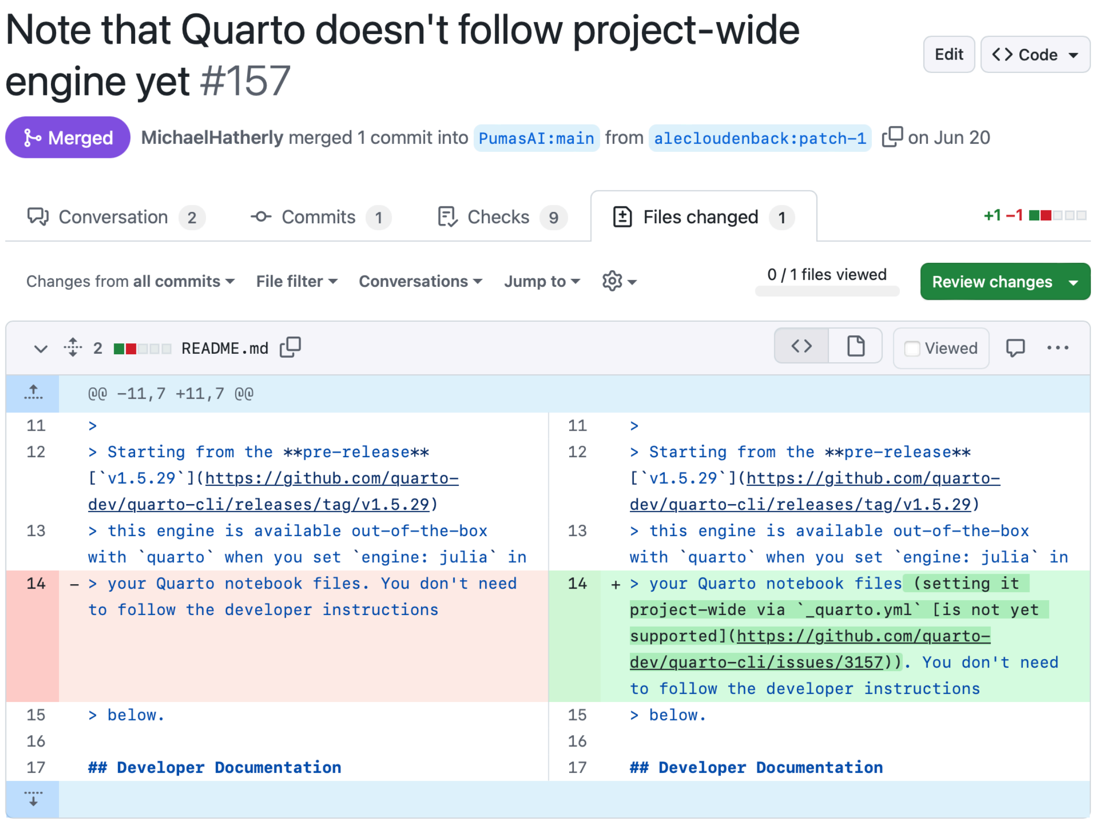



Figure 11.2.: A diff shows a file-by-file and line-by-line comparison of what has changed between commits in a codebase. Red inidicates something was removed or changed, and green shows what replaced it. Note that even within a line, there's extra green highlighting to show the newly added text while the unchanged text remains a lighter shade.

And if the data is changed even slightly, then the output is markedly different:

```
data2 = "A data file contents 124"
Base.hash(data2)
```

```
0xb0b70b13a7d93dcb
```

This is used in content addressed systems like Data Version Control (Section 11.5.3) and Artifacts (sec (**artifacts?**)) to ask for, and confirm the accuracy of, data instead of trying to address the data by it's location. That is, instead of trying to ask to get data from a given URL (e.g. `http://adatasource.com/data.csv`) you can set up a system which keeps track of available locations for the data that matches the content hash. Something like (in Julia-ish psuedocode):

```
storage_locations = Dict(
    0x4d7e8e449afd1c48 => [
        "http://datasets.com/1231234",
        "http://companyintranet.com/admin.csv",
        "C:/Users/your_name/Documents/Data/admin.csv"
    ]
)

function get_data(hash,locations)
    for location in locations[hash]
        if is_available(location)
            return get(location)
        end
    end

    # if loop didn't return data
    return nothing
end
```

:::

## 11.6. Distributing the Package

Ones you have creates something, the next best feeling after having it working is having someone else also use the tool. Julia has a robust way to distribute and manage dependencies. This section will cover essential and related topics to distributing your project publicly or with an internal team.

### 11.6.1. Registries

**Registries** are a way to keep track of packages that are available to install, which is more complex than it might seem at first. The registry needs to keep track of:

- What dependencies your package has.
- What versions of the dependencies your package is compatible with.
- Metadata about the package (name, unique identifier, authors).
- Versions of your package that your have made available and the Git hash associated with the that version for provenance and tracking
- The location where your package's repository lives so that the user can grab the code from there.

The Julia General Registry ("General") is the default registry that comes loaded as the default registry when installing Julia. From a capability standpoint, there's nothing that separates General from other registries, including ones that you can create yourself. At it's core, the registry is can be seen as a Git repository where each new commit just adds information associated with the newly registered package or version of a package.

For distributing packages in a private, or smaller public group see Section 22.9.2.1.

#### 11.6.1.1. General Registry and other Hosted Registries

At it's core, General is essentially the same as a local registry described in the prior section. However, there's some additional infrastructure supporting General. Registered packages get backed up, cached for speed, and multiple servers across the globe are set up to respond to Pkg requests for redundancy and latency. Nothing would stop you from doing the same for your own hosted registry if it got popular enough!

> Tip
>
> A local registry is a great way to set up internal sharing of packages within an organization. Services do exist for "managed" package sharing, adding enterprise features like whitelisted dependencies, documentation hosting, a 'hub' of searchable packages.

### 11.6.2. Versioning

Versioning is an important part of managing a complex web of dependencies. Versioning is used to let both users and the computer (e.g. Pkg) understand which bits of code are compatible with others. For this, consider your model/program's **Application Programming Interface** (API). The API is essentially defined by the outputs produced by your code given the same inputs. If the same inputs are provided, then for the same API version the same output should be provided. However, this isn't the only thing that matters. Another is the documentation associated with the functionality. If the documentation said that a function would work a certain way, then your program should follow that (or fix the documentation)!

Another case to consider is what if new functionality was *added*? Old code need not have changed, but if there's new functionality, how to communicate that as an API change? This is where **Semantic Versioning** (SemVer) comes in: *semantic* means that your version something is intended to convey some sort of meaning over and above simply incrementing from `v1` to `v2` to `v3`, etc.

#### 11.6.2.1. Semantic Versioning

Semantic Versioning (SemVer) is one of the most popular approaches to software versioning. It's not perfect but has emerged as one of the most practical ways since it get's a lot about version numbering right. Here's how SemVer is defined[2]:

```
Given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make incompatible API changes
- MINOR version when you add functionality in a backward compatible manner
- PATCH version when you make backward compatible bug fixes
```

So here are some examples of SemVer, if our package's functionality for `v1.0.0` is like this:

```
""" my_add(x,y)

Add the numbers x and y together
"""
my_add(x,y) = x - y
```

**Patch change (v1.0.1):** Fix the bug in the implementation:

```
""" my_add(x,y)
Add the numbers x and y together
"""
my_add(x,y) = x + y
```

[2]You can read more at SemVer.org.

This is a patch change because it fixes a bug without changing the API. The function still takes two arguments and returns their sum, as originally documented.

**Minor change (v1.1.0):** Add new functionality in a backward-compatible manner:

```
""" my_add(x,y)
Add the numbers x and y together

my_add(x,y,z)
Add the numbers x, y, and z together
"""
my_add(x,y) = x + y
my_add(x,y,z) = x + y + z
```

This is a minor change because it adds new functionality (the ability to add three numbers) while maintaining backward compatibility with the existing two-argument version.

**Major change (v2.0.0):** Make incompatible API changes:

```
""" add_numbers(numbers...)
Add any number of input arguments together
"""
add_numbers(numbers...) = sum(numbers)
```

This is a major change because it fundamentally alters the API. The function name has changed, and it now accepts any number of arguments instead of specifically two or three. This change is not backward compatible with code using the previous versions, so a major version increment is necessary.

> **Note**
>
> Numbers need not roll over to the next digit when they hit 10. That is, it's perfectly valid to go from `v1.09.0` to `v1.10.0` in SemVer.

> **Tip**
>
> Sometimes you'll see a package with a version that starts with zero, such as `v0.23.1`. **We recommend that as soon as you register a package, to make it a `v1`**. `v1` need not indicate the package is "complete" (what software is?), so don't hold back on calling it `v1`. You're letting users install it easily, so you might as well call it the first version and move on!
> According to SemVer's rules, there are no patch versions when the major version is zero. This means that you have one less meaningful digit to communicate to users what's going on with a change. Most packages put an upper bound on com-

> patibility so that major or minor changes in upstream packages are less likely to cause issues in their own packages. This can be somewhat painful to depend on a package which has a `v0` and is iterating through 'fixes' but is incrementing the minor version. You have to assume it's making backward incompatible changes and should have skepticism of just upgrading to the new version of the dependency. It takes work on the downstream dependencies to decide if they should upgrade, adding mental and time loads to other authors and users.

### 11.6.3. Artifacts

Artifacts are a way to distribute content-addressed ((**note-content-hash?**)) data and other dependencies. An example use case is if you want to distribute some demonstration datasets with a package. When a package in added or updated, the associated data is pulled and un-archived by Pkg instead of the author of the package needing to manually handle data dependencies. Aside from this convenience, it means that different packages could load the same data without duplicating the data download or storage (since the data is content-addressed). The use-case is not real-time data, as the content-hash can only be updated per package version.

For example, the MortalityTable.jl package redistributes various publicly available, industry mortality tables. Inside the repository, there's an `Artifacts.toml` file specified like:

```
#/Artifacts.toml

["mort.soa.org"]
git-tree-sha1 = "6164a6026f108fe95828b689fcd3b992acbff7c3"                    ①

    [["mort.soa.org".download]]
    sha256 = "6f5eb4909564b55a3397ccf4f4c74290e002f7e2e2474cebeb224bb23a9a2606" ②
    url = "https://github.com/JuliaActuary/Artifacts/raw/v2020-02-15/mort.soa.org/2020-02-15.t
```

① The sha1 hash of the un-archived data once downloaded, used to verify that extraction was successful.
② The sha256 hash of the archived (compressed) data to ensure that the data downloaded was as intended.

Then, within the package the data artifact can be referenced and handled by the artifact system rather than needing to manually handle it. That is, the data is reference-able like this:

```
table_dir = artifact"mort.soa.org"                                               ①
```

① artifact"..." is a string macro, which is a special syntax for macros that interact with strings. `md"..."` is another example, specifiying that the content of the string is Markdown content.

As opposed to something like this:

```
# psuedo Julia code
table_dir = if is_first_package_run

    data_path = download("url_of_data.tar.gz") # download to a temp location
    mv(data_path, "/somwhere/to/keep/data.tar.gz") # move to a 'permanent' location
    extract("/somwhere/to/keep/data.tar.gz") # extract contents
    "/somwhere/to/keep/data/" # return data path
else
    "/somwhere/to/keep/data/" # return pre-processed path

end
```

> **Note**
>
> Utility Packages such as ArtifactUtils.jl can assist in creating correct entries for `Artifact.toml` files.

> **Note**
>
> Artifacts support `.tar` (uncompressed) and `.tar.gz` because that compression format enjoys more universal support and features than the `.zip` format most common on Windows systems.

## 11.7. Example Repository

The is a good example of a repository which shows the organization of files and code, setting up testing, and documentation.

## 11.8. Example Repository Structure

A well-structured Julia package demonstrates key software engineering principles in action. The JuliaTemplateRepo repository demonstrates best practices for: - Logical file organization and code structure - Comprehensive test coverage and continuous integration - Clear, accessible documentation with examples - Standard tooling configuration for package development

This open-source template serves as a reference implementation.

The PkgTemplates.jl package will allow you to create an empty repository with all of the testing, documentation, Git, and continuos integration scaffolding already in place.

# 12. 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)

## 12.1. In this section

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

## 12.2. 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).

## 12.3. Algorithms & Complexity

**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 mathematical function or process. Further, we should indicate that a process needs to be specified in sufficient detail to be able to call itself an algorithm versus a heuristic which does not indicate with enough detail how the process would unfold.

### 12.3.1. 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.

#### 12.3.1.1. 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.

#### 12.3.1.2. 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.

#### 12.3.1.3. 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.

#### 12.3.1.4. 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.

#### 12.3.1.5. Comparison

##### 12.3.1.5.1. Big-O Notation

We can categorize the above implementations using a convention called **Big-O Notation**[1] 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 12.1 shows for the examples constructed above what the description, order, and order of magnitude complexity is.

Table 12.1.: 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 12.2 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 12.2.: Different Big-O Orders of Complexity

| Big-O Order | Description | $n = 10$ | $n = 1,000$ | $n = 1,000,000$ |
|---|---|---|---|---|
| $O(1)$ | Constant Time | 1 | 1 | 1 |
| $O(n)$ | Linear Time | 10 | 1,000 | 1,000,000 |
| $O(n^2)$ | Quadratic Time | 100 | 1,000,000 | 10^12 |
| $O(log(n))$ | Logarithmic Time | 3 | 7 | 14 |

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

| Big-O Order | Description | $n = 10$ | $n = 1,000$ | $n = 1,000,000$ |
|---|---|---|---|---|
| $O(n \times log(n))$ | Linearithmic Time | 30 | 7,000 | 14,000,000 |
| $O(2^n)$ | Exponential Time | 1,024 | ~10^300 | ~10^301029 |
| $O(n!)$ | Factorial Time | 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.

#### 12.3.1.5.2. Empirical Results

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

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

```
50005000
```

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

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

```
50005000
```

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

```
  2.398 μs (0 allocations: 0 bytes)
```

```
50005000
```

The preceding examples of constant, linear, and exponential times are *conceptually* correct but if we try to run them in practice we see that the description doesn't seem to hold at all for the linear time version, as it runs as quickly as the constant time version.

What happened was that the compiler was able to understand and optimize the linear version such that it effectively transformed it into the constant time version and avoid the iterative summation that we had written. For examples that are simple enough to use as a teaching problem, the compiler can often optimize different written code down to the same efficient machine code (this is the same Triangular Number optimization we saw in Section 4.4.3.4).

### 12.3.2. 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.

### 12.3.3. 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).

## 12.4. Data Structures

**Data structures** is the art and science of how to represent data in discrete objects. There are many common kinds and many specialized sub-kinds, and we will describe some of the most common ones here. Julia has many data structures available in the Base library, but an extensive collection of other data structures can be found in the DataStructures.jl package.

### 12.4.1. Arrays

An **array** is a contiguous block of memory containing elements of the same type, accessed via integer indices. Arrays have fast random access and are the fastest data structure for linear/iterated access of data.

In Julia, an array is a very common data structure and is implmemented with a simple declaration, such as:

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

In memory, the integers are stored as consecutive bits representing the integer values of `1`, `2`, and `3`, and would look like this (with the different integers shown on new lines for clarity):

```
0000000000000000000000000000000000000000000000000000000000000001
0000000000000000000000000000000000000000000000000000000000000010
0000000000000000000000000000000000000000000000000000000000000011
```

This is great for accessing the values one-by-one or in consecutive groups, but it's not efficient if values need to be inserted in between. For example, if we wanted to insert `0` between the `1` and `2` in `x`, then we'd need to overwrite the second position in the array, ask the operating system to allocate more memory[2], and re-write the bytes that come after our new value. Inserting values at the end (`push!(array, value)`) is usually fast unless more memory needs to be allocated.

### 12.4.2. Linked Lists

A **linked list** is a chain of nodes where each node contains a value and a pointer to the next node. Linked lists allow for efficient insertion and deletion but slower random access compared to arrays.

In Julia, a simple linked list node could be implemented as:

```
mutable struct Node
    value::Any
    next::Union{Node, Nothing}                                    ①
end

z = Node(3,Nothing)
```

[2]In practice, the operating system may have already allocated space for an array that's larger than what the program is actually using so far, so this step may be 'quick' at times, while other times the operating system may actually need to extend the block of memory allocated to the array.

```
y = Node(2,z)
x = Node(1,y)
```

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

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

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

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

### 12.4.3. Records/Structs

An aggregate of named fields, typically of fixed size and sequence. Records group related data together. We've encountered structs in Section 4.4.7, but here we'll add that simple `structs` with primitive fields can themselves be represented without creating pointers to the data stored:

```
struct SimpleBond
    id::Int
    par::Float64
end

struct LessSimpleBond
    id::String
    par::Float64
end

a = SimpleBond(1, 100.0)
b = LessSimpleBond("1", 100.0)
isbits(a), isbits(b)
```

```
(true, false)
```

Because `a` is comprised of simple elements, it can be represented as a contiguous set of bits in memory. It would look something like this in memory:

```
0000000000000000000000000000000000000000000000000000000000000001 # <1>
0100000001011001000000000000000000000000000000000000000000000000 # <2>
```

① The bits of `1`
② The bits of `100.0`

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

```
.... a pointer ... # <1>
0100000001011001000000000000000000000000000000000000000000000000 # <2>
```

① a pointer/reference to the array of characters that comprise the string ID
② The bits of `100.0`

In performance critical code, having data that is represented with simple bits instead of references/pointers can be much faster (see Chapter 24 for an example).

> **Note**
>
> For many mutable types, there are immutable, bits-types alternatives. For example:
>
> - Arrays have a `StaticArray` counterpart (from the StaticArrays.jl package).
> - Strings have `InlineStrings` (from the InlineStrings.jl package) which use fixed-width representations of strings.
>
> The downsides to the immutable alternatives (other than the loss of potentially desired flexibly that mutability provides) are that they can be harder on the compiler (more upfront compilation cost) to handle the specialized cases involved.

### 12.4.4. Dictionaries (Hash Tables)

#### 12.4.4.1. Hashes and Hash Functions.

**Hashes** are the result of a **hash function** that maps arbitrary data to a fixed size value. It's sort of a "one way" mapping to a simpler value which has the benefits of:

1. One way so that if someone knows the hashed value, it's *very* difficult to guess what the original value was. This is most useful in cryptographic and security applications.
2. Creating (probabilistically) unique IDs for a given set of data.

For example, we can calculate a type of has called an SHA hash on any data:

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

```
b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9
61391c8623dadb01976ad1623375b514785a43591e6b3c6b906b8729c572293f
```

We can easily verify that the `sha256` hash of `"hello world"` is the same each time, but it's virtually impossible to guess `"hello world"` if we are just given the resulting hash. This is the premise of trying to "crack" a password when the stored password hash is stolen.

One way to check if two set of data are the same is to compute the hash and see if the resulting hashes are equal. For example, maybe you want to see if two data files with different names contain the same data - comparing the hashes is a sure way to determine if they contain the same data.

#### 12.4.4.2. Dictionaries

Dictionaries map a *key* to a *value*. More specifically, they use the *hash of a key* to store a reference to the *value*.

Dictionaries offer constant-time average case access but must handle potential collisions of keys (generally, the more robust the collision handling means higher fixed cost for access).

### 12.4.5. Graphs

A **graph** is a collection of nodes (also called vertices) connected by edges to represent relationships or connections between entities. Graphs are versatile data structures that can model various real-world scenarios such as social networks, transportation systems, or computer networks.

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

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

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

g = Graph(Dict())
add_edge!(g, 1, 2)
add_edge!(g, 2, 3)
add_edge!(g, 1, 3)
```

This implementation represents an undirected graph. For a directed graph, you would only add the edge in one direction.

Graphs can be traversed using various algorithms such as depth-first search (DFS) or breadth-first search (BFS). These traversals are useful for finding paths, detecting cycles, or exploring connected components.

For more advanced graph operations, the Graphs.jl package provides a comprehensive set of tools for working with graphs in Julia.

### 12.4.6. Trees

A tree is a hierarchical data structure with a root node and child subtrees. Each node in a tree can have zero or more child nodes, and every node (except the root) has exactly one parent node. Trees are widely used for representing hierarchical relationships, organizing data for efficient searching and sorting, and in various algorithms.

A simple binary tree node in Julia could be implemented as:

```
mutable struct TreeNode
    value::Any
    left::Union{TreeNode, Nothing}
    right::Union{TreeNode, Nothing}
end

# Creating a simple binary tree
root = TreeNode(1,
    TreeNode(2,
        TreeNode(4, nothing, nothing),
        TreeNode(5, nothing, nothing)
```

```
    ),
    TreeNode(3,
        nothing,
        TreeNode(6, nothing, nothing)
    )
)
```

Trees have various specialized forms, each with its own properties and use cases:

- Binary Search Trees (BST): Each node has at most two children, with all left descendants less than the current node, and all right descendants greater.
- AVL Trees: Self-balancing binary search trees, ensuring that the heights of the two child subtrees of any node differ by at most one.
- B-trees: Generalization of binary search trees, allowing nodes to have more than two children. Commonly used in databases and file systems.
- Trie (Prefix Tree): Used for efficient retrieval of keys in a dataset of strings. Each node represents a common prefix of some keys.

Trees support efficient operations like insertion, deletion, and searching, often with $O(logn)$ time complexity for balanced trees. They are fundamental in many algorithms and data structures, including heaps, syntax trees in compilers, and decision trees in machine learning.

### 12.4.7. Data Structures Conclusion

Data structures have strengths and weakness depending on whether you want to priortize computational efficiency, memory (space) efficiency, code simplicity, and/or mutability. Due to the complexity of real world modeling needs, it can be the case that different representations of the data are more natural or more efficient for the use case at hand.

## 12.5. Formal Verification

Formal verification is a technique used to prove or disprove the correctness of algorithms with respect to a certain formal specification or property. In essence, it's a mathematical approach to ensuring that a system behaves exactly as intended under all possible conditions.

### 12.5.1. Basic Concept

In formal verification, we use mathematical methods to:

1. Create a formal model of the system
2. Specify the desired properties or behaviors
3. Prove that the model satisfies these properties

This process can be automated using specialized software tools called theorem provers or model checkers.

### 12.5.2. Formal Verification in Practice

It sounds like the perfect risk management and regulatory technique: prove that the system works exactly as intented. However, there has been very limited deployment of formal verification in industry. This is for serveral reasons:

1. Incomplete Coverage: It's often impractical to formally verify entire large-scale financial systems. Verification, if at all, is typically limited to critical components.
2. Incomplete Specification: Actually reasoning through how the system should behave in all scenarios requires actually contemplating mathematically complete and rigorous possiblilities that could occur.
3. Model-Reality Gap: The formal model may not perfectly represent the real-world system, especially in finance where market behavior can be unpredictable.
4. Changing Requirements: Financial regulations and market conditions change rapidly, potentially outdating formal verifications.
5. Performance Trade-offs: Systems designed for easy formal verification might sacrifice performance or flexibility.
6. Cost: The process can be expensive in terms of time and specialized labor.

### 12.5.3. Related Topics

#### 12.5.3.1. Property Based Testing

Testing will be discussed in more detail in Chapter 11, but an intermediate concept between Formal Verification and typical software testing is **property-based** testing, which tests for general rules instead of specific examples.

For example, a function which is associative ($(a+b)+c = a+(b+c)$) or commutative ($a + b = b + c$) can be tested with simple examples like:

```
using Test

myadd(a,b) = a + b

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

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

#### 12.5.3.2. Fuzzing

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

```
@testset for i in 1:10000
    a = rand()
    b = rand()

    @test myadd(a,b) == myadd(b,a)
end
```

This is a good advancement over the simple `@test myadd(1,2) == myadd(2,1)`, in terms of checking the correctness of `myadd`, but it comes at the cost of more computational time and non-deterministic tests.

# 13. Statistical Inference and Information Theory

"My greatest concern was what to call [the amount of unpredictability in a random outcome]. I thought of calling it 'information,' but the word was overly used, so I decided to call it 'uncertainty.'

When I discussed it with John von Neumann, he had a better idea. Von Neumann told me, 'You should call it entropy, for two reasons. In the first place, your uncertainty function has been used in statistical mechanics under that name, so it already has a name. In the second place, and more important, no one really knows what entropy really is, so in a debate you will always have the advantage.' "- Claude Shannon (1971)

## 13.1. In This Chapter

A brief introduction to information theory and its foundational role in statistics. Entropy and probability distributions. Bayes' rule and model selection comparison via likelihoods. A brief tour of modern Bayesian statistics.

## 13.2. Introduction

Statistics has an invaluable role in any data-driven modeling enterprise. As financial professionals dealing inherently with risk and uncertainty - we use probability and statistics to understand, model, and communicate these aspects.

Statistics curricula and practice is undergoing a significant transformation, with a larger focus on information theory and Bayesian methods (as opposed to the common Frequentist methods that have dominated the statistics field for mor than a century). Why the change? In short, these methods work better in a wider range of situations and convey more meaningful information about model performance and uncertainty.

In our experience, it is rare that a financial professional has had exposure to non-Frequentist theory and methods. Given how central probability and statistics is to this endeavor, we have drafted this chapter as a introduction - a proper treatment is beyond the scope of this book but armed with this knowledge, some of the terminology and tools should be more accessible, and possibly included in new financial models.

## 13.3. Information Theory

Probability, statistics, machine learning, signal processing, and even physics have a foundational link in **information theory** which is the description and analysis of how much useful data is contained within something.

Let's consider the following situation: we are studying a poorly made copy of a financial statement. Amongst many associated exhibits, we are interested in how many assets of a particular kind were held. Unfortunately, for one reason or another one of the digits is completely indecipherable. Here's what you can read, with the _ indicating that the digit is not visible:

```
32,000,_00
```

It is likely that you quickly formed an opinion on what the missing number is, but let us make that intuition more formal and quantitative.

Given that we know 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 13.1. 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[1]:

$$h(x_i) = \log_2 \frac{1}{p(x_i)} \tag{13.1}$$

Looking at Table 13.1, we can see that the information content of an outcome is *lower* when that outcome has a higher probability than the other potential outcomes. Specifically, If the digit was indeed `0`, we have gained less information relative to our expectation than if the missing digit were anything other than `0` .

> **Tip**
>
> The information content is sometimes referred to as a measure of *surprise* that one would have when observing a realized outcome. In our missing digit example

[1]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.

> (@tbl-digit-information-human), we would not be surposeid at a ll to find out that the missing digit were `0`. In contrast, we would be more surprised to find out the digit were an `8`.

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

$$H(X) = \sum p(x_i) \log_2 \frac{1}{p(x_i)} \tag{13.2}$$

The entropy $H(X)$ of the presumed outcomes in Table 13.1 distribution of outcomes is $0.722$bits.

Table 13.1.: Probability distribution of missing digit, knowing the human inclination to prefer round numbers when estimating.

| $x_i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $p(x_i)$ | .91 | .01 | .01 | .01 | .01 | .01 | .01 | .01 | .01 | .01 |
| $h(x_i)$ | 0.136 | 6.644 | 6.644 | 6.644 | 6.644 | 6.644 | 6.644 | 6.644 | 6.644 | 6.644 |

To be clear, we have take a non-uniform view on the probability distribution for the missing digit, and we'll refer to this as the **prior assumption** (or just **prior**). This is unashamedly an opinionated assumption, just like your intuition when you encountered `32,000,_00`! All we are doing is giving a quantitative basis for describing this assumption. Taking a view on a prior distribution is a quantitative way to describe incorporating previously encountered data and professional judgment. Having a prior assumption like this is completely compatible with information theory.

Our professional judgment notwithstanding: what if we had another colleague who believed humans are completely rational and without bias for certain numbers? They would then be arguing for a prior assumed distribution consistent with Table 13.2.

With the uniform prior assumption, $H(X) = 3.322$bits and $h(x_i)$ is also uniform. Note that $H$ is higher for the uniform prior than the prior in Table 13.1. We will not prove it here, but a uniform probability over a set of outcomes is the highest entropy distribution that can be assumed for this problem. A higher entropy prior distribution can typically be viewed as a less biased prior assumption than a lower entropy prior.

Table 13.2.: Probability distribution of missing digit with uniform, maximal entropy for the assumed probability distribution.

| $x_i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $p(x_i)$ | .10 | .10 | .10 | .10 | .10 | .10 | .10 | .10 | .10 | .10 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $h(x_i)$ | 3.322 | 3.322 | 3.322 | 3.322 | 3.322 | 3.322 | 3.322 | 3.322 | 3.322 | 3.322 |

The choice of prior assumption can significantly impact the interpretation and analysis of the missing information. If we have strong reasons to believe that the human bias prior is more appropriate given the context (e.g., knowing that the number is an estimate), then we would expect the missing digit to be '0' with high probability. However, if we have no specific knowledge about the nature of the number and prefer to make a more conservative assumption, the uniform prior may be more suitable.

In real-world scenarios, the choice of prior assumptions often depends on domain knowledge, available data, and the specific problem at hand. It is important to carefully consider and justify the prior assumptions used in information-theoretic and statistical analyses.

### 13.3.1. Example: Classificaiton

In this example, we will determine the optimal splits for a decision tree[2] based on the information gained at each node in the tree.

```
using DataFrames

employed = [true, false, true, true, true, false, false, true]
good_credit = [true, true, false, true, false, false, false, true]
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 13.2:

```
H0 = let
    p_default = sum(default_data.default) / nrow(default_data)
    p_good = 1 - p_default
    p_default * log2(1 / p_default) + p_good * log(1 / p_good)
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. Whichver

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

Table 13.3.: 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 |

critieria gains us the most information is the preferred attribute to create a decision split.

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. The name for this is the **information gain**, $IG(inputs, attributes)$:

$$IG(T, a) = H(T) - H(T|a)$$

In words, the the information gain is simply the difference in entropy before and after learning the value of an outcome $a$. We will develop an example illustrating this

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
    p_default = sum(df_employed.default) / nrow(df_employed)
    p_good = 1 - p_default
    # p_default * log2(1 / p_default) + p_good * log(1 / p_good)
    p_default * log2(1 / p_default) + 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
    p_default = sum(df_unemployed.default) / nrow(df_unemployed)
    p_good = 1 - p_default
    p_default * log2(1 / p_default) + p_good * log(1 / p_good)
end
```

```
0.7986309056458281
```

To balance these two results, we weight them according to the amount of data (number of observations) that would fall into each leaf:

```
H1_employment = let
    p_emp = nrow(df_employed) / nrow(default_data)
    p_unemp = 1 - p_emp

    p_emp * H_employed + p_unemp * H_unemployed
end
```

```
0.29948658961718555
```

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 13.3 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.

### 13.3.2. 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[3].

Table 13.4.: Maximum Entropy Distributions and the conditions under which they are applicable. For example, if you know that a probability must be continuous and have a positive mean (and can't be normalized), then the MED is the Exponential Distribution.

| Constraint | Discrete Distribution | Continuous Distribution |
|---|---|---|
| Bounded range | Uniform (discrete) | Uniform (continuous) |

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

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

The distributions in Table 13.4 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.

#### 13.3.2.1. 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 13.5.: Underlying processes create typical probability distributions. That there is significant overlap with the distributions in Section 13.3.2 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 |
| 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 an unusual dataset, you don't need to comb the depths of Wikipedia to find the perfect probability distribution for that situation.

#### 13.3.2.2. Additive and Multiplicative Processes

Table 13.5 describes some examples, let us discuss further what it means to have a process that arises via an additive vs multiplicative effect[4].

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.

Tip 1: Logarithms

The logarithm of a geometric process transforms the outcomes into "log-space". The information is the same, but is often a more convenient form for the analysis. That is, if:

[4] 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.

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

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.

In the context of computational thinking:
*First,* we should think about how to transform data or modeling outcomes into a more convenient format. The log transform doesn't eliminate any information but may map the information into a shape that is easier for an optimizer or Monte Carlo simulation to explore.
*Second*, per Chapter 4, 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.

## 13.4. Bayes' Rule

The minister and statistician Thomas Bayes derived a relationship of conditional probabilities that we today know as **Bayes' Rule**[5], 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 $D$.

[5]Laplace actually deserves most of the credit, as it was he who formalized the modern notion of Bayes' rule and cemented the mathematical formulation. Bayes just described it first, in a way that actually had almost no direct impact on math or science. See "The Theory That Would Not Die".

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 13.5. First, let's look at combining information theory and Bayes' rule in an applied example.

### 13.4.1. 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:

$$P(H_1 \| D) = (P(D \| H_1) \times P(H_1))/P(D)$$

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

Where:

- $P(H_1|D)$ and $P(H_2|D)$ are the posterior probabilities of models $H_1$ and $H_2$, respectively, given the data $D$.
- $P(D|H_1)$ and $P(D|H_2)$ are the likelihoods of the data $D$ under models $H_1$ and $H_2$, respectively.
- $P(H_1)$ and $P(H_2)$ are the prior probabilities of models $H_1$ and $H_2$, respectively.
- $P(D)$ is the marginal likelihood of the data, which serves as a normalizing constant.

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

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

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

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

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

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

The interpretation of the Bayes factor is as follows:

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

In practice, the likelihoods $P(D|H_1)$ and $P(D|H_2)$ are often calculated using the probability density or mass functions of the models, evaluated at the observed data points. The prior probabilities $P(H_1)$ and $P(H_2)$ can be assigned based on prior knowledge or assumptions about the models. By comparing the likelihoods of the models using the Bayes factor, we can quantify the relative support for each model given the observed data, while taking into account the prior probabilities of the models.

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

⚠ Null 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! There is subjectivity in the null hypothesis, data collection methodologies, study design, handling of missing data, choice of data *not* to include, which statistical tests to perform, and interpretation of relationships.
We might as well call the null hypothesis a prior and stop trying to disprove it absolutely. Instead: focus on model comparison, model structure, and posterior probabilities of the competing theories.
Over 100 statistical tests have been developed in service of NHST Lewis (2013), but it's widely viewed now that a focus on NHST has led to *worse* science due to a multitude of factors, such as:

- "P-hacking" or trying to find subsets of data which can (often only by chance) support rejecting some null.
- Cognitive anchoring to the importance of a p-value of 0.05 or less – why choose that number versus 0.01 or 0.001 or 0.49?
- Bias in research processes where one may stop data collection or experimentation after achieving a favorable test result.
- Inappropriate application of the myriad of statistical tests.
- Focus on p-values rather than effects that simply matter more or have greater effect.
  - For example, which is of more interest to doctors? A study indicating a 1 in a billion chance of serious side effect , with p-value of `0.0001` or a study indicating a 1 in 3 chance with p-value `0.06`? Many journals would only publish the former study, but the latter study intuitively suggests a potentially more risky drug.
- Difficulty to determine *causal* relationships.

The authors of this book recommend against basic NHST and memorization of statistical tests in favor of principled Bayesian approaches. For the actuarial readers, NHST is analogous to traditional credibility methods (of which the authors also prefer more modern statistical approaches).

#### 13.4.1.1. Example: Rainfall Risk Model Comparison

The example we'll look at relates to the annual rainfall totals for a specific location in California[6], 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[7] instead of a scientifically accurate weather model, we narrow the problem to finding a

[6]https://data.ca.gov/dataset/annual-precipitation-data-for-northern-california-1944-current
[7]See @sec-predictive-vs-explanatory.

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 13.6 shows three competing models based on thinking about the real-world outcome we are trying to model. These three are chosen for the increasingly sophisticated thought process that might lead the modeler to recommend them - but which is supportable by the statistics?

Table 13.6.: Three alternative hypothesis about the distribution of annual rainfall totals.

| Hypothesis | Process | Possible Rationale |
|---|---|---|
| $H_1$ | A Normal (Gaussian) distribution | The sum of independent rainstorms creates annual rainfall totals that are normally distributed |
| $H_2$ | A LogNormal distribution | Since it's normal-ish, but skewed and can't be negative |
| $H_3$ | A Gamma Distribution | Since rainfall totals would be the sum of exponentially-distributed independent rainfall events |

**Note**

In the literature for rainfall modeling, $H_3$ (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.
We are able to avoid learning and memorizing specialty distributions and statistical tests, which are so common in Frequentist approaches. First-principles reasoning on the probabilistic processes can get one to a reasonable hypothesis, comparable to 'specialist' knowledge one would encounter in the literature for a particular applied field.

Here's the data:

```
rain = [
    39.51, 42.65, 44.09, 41.92, 28.42, 58.65, 30.18, 64.4, 29.02,
    37.00, 32.17, 36.37, 47.55, 27.71, 58.26, 36.55, 49.57, 39.84,
    82.22, 47.58, 51.18, 32.28, 52.48, 65.24, 51.12, 25.03, 23.27,
    26.11, 47.3, 31.8, 61.45, 94.95, 34.8, 49.53, 28.65, 35.3, 34.8,
    27.45, 20.7, 36.99, 60.54, 22.5, 64.85, 43.1, 37.55, 82.05, 27.9,
    36.55, 28.7, 29.25, 42.32, 31.93, 41.8, 55.9, 20.65, 29.28, 18.4,
    39.31, 20.36, 22.73, 12.75, 23.35, 29.59, 44.47, 20.06, 46.48,
    13.46, 9.34, 16.51, 48.24
];
```

Plotted, we see some of the characteristics that align with our prior assumptions and knowledge about the system itself, such as: the data being constrained to positive values and a skew towards having some extreme weather years with lots of rainfall.

```
using CairoMakie
hist(rain)
```

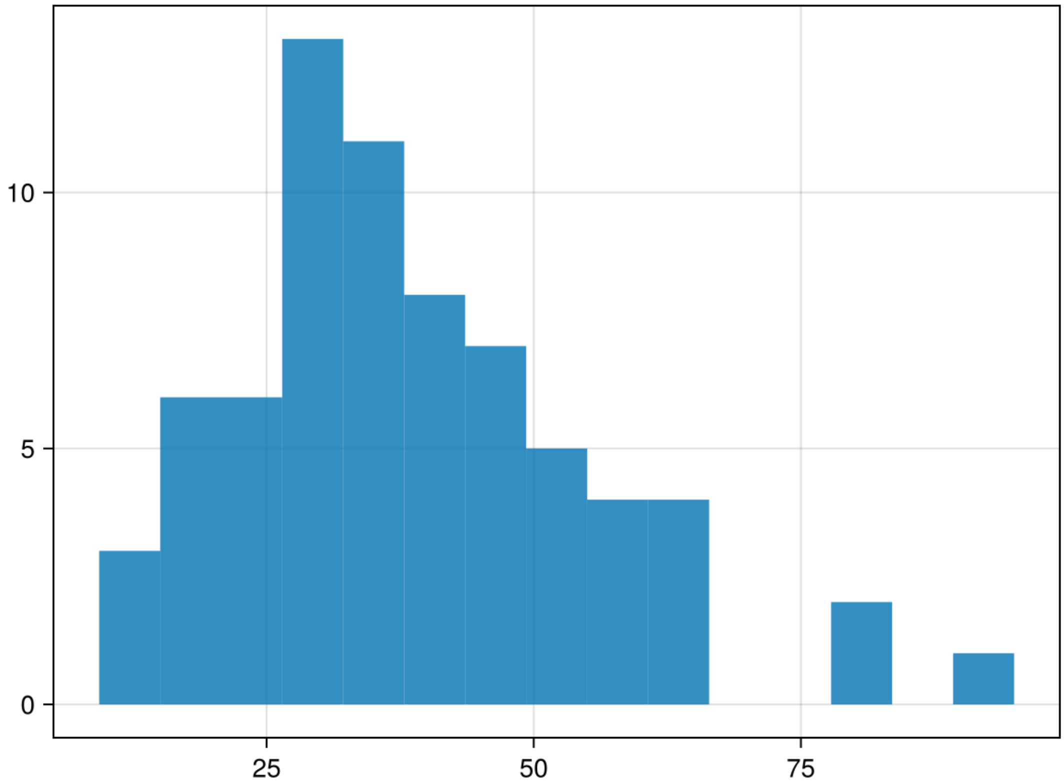



Figure 13.1.: Annual rainfall totals for a specific location in California.

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)
ln = fit_mle(Normal, log.(rain))
```

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

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

Let's look at the likelihoods. For the practical reasons described in Tip 1, we will compare the the log-likelihoods to maintain convention with what you'd likely see or deal with in practice. Taking the log of the likelihood does not change the ranking of the likelihoods.

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

    @show n_lik
    @show ln_lik
    @show lg_lik
end;
```

```
n_lik = -296.16751566478115
ln_lik = -42.09272021737913
lg_lik = -43.79151806348801
```

The results indicate that the LogNormal and the Gamma model for rainfall distribution are very superior to the Normal model, consistent with the visual inspection of the quantiles in Figure 13.2. We reach that conclusion by noting how much more likely the latter two are, as the likelihoods of $-42$ and $-44$ is much greater than $-296$[8].

```
let x = rain

    range = 1:0.1:100
    fig, ax, _ = lines(range, cdf.(n, range), label="Normal", axis=(xgridvisible=false, ygridv
    lines!(ax, range, cdf.(ln, log.(range)), label="LogNormal")
    lines!(range, cdf.(lg, log.(range)), label="LogGamma")
```

[8]The values are negative because we are taking the logarithm of a number less than 1. The likelihoods are less than 1 because the likelihood is the joint (multiplicative) probability of observing each of the individual outcomes.

```
    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
```

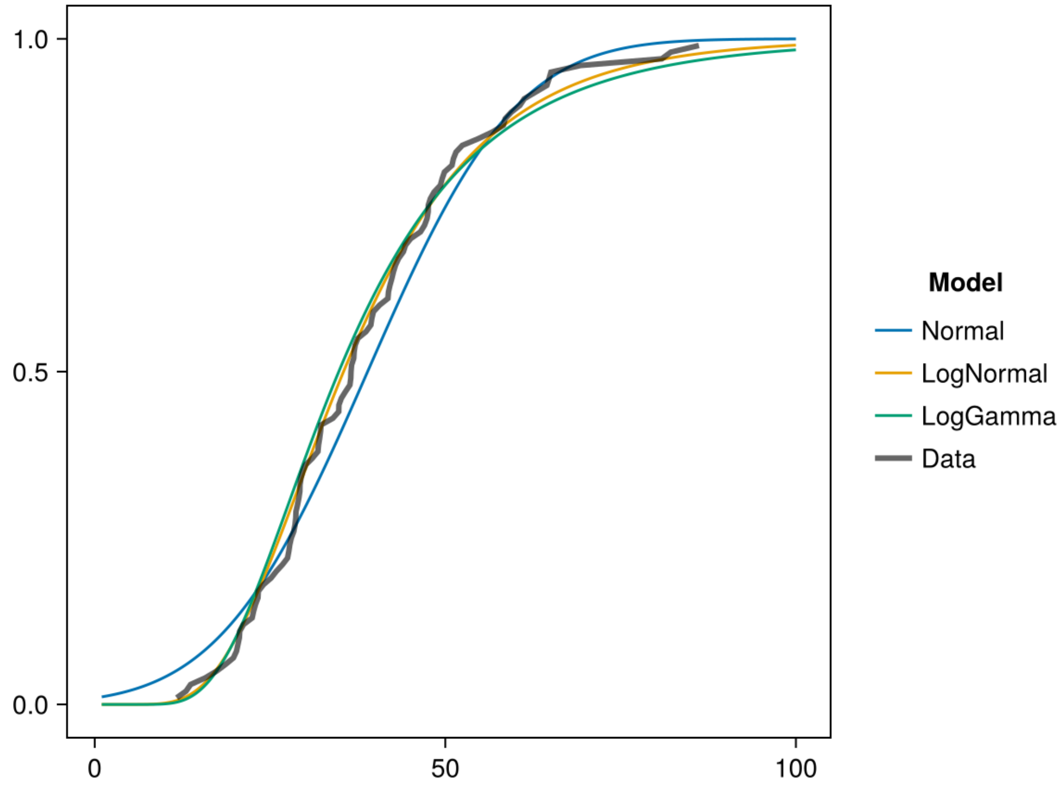



Figure 13.2.

We evaluated the likelihood at a single point estimate of the parameters, but a true posterior probability of the parameters of the distributions will be represented by a *distribution* rather than a point. The rest of chapter will describe how to express the posterior probabiliteis of the the parameters for $H_1$, $H_2$, and $H_3$ using Bayesian statistical methods.

# 13.5. Modern Bayesian Statistics

## 13.5.1. Background

Bayesian statistics is generally *not* taught in undergraduate statistics. Bayes' rule is introduced, basic probability exercises are assigned, and then statistics moves on to a curriculum of regression and NHSTs (of the Frequentist school). Why is the applied practice of statistics then gravitating towards Bayesian approaches? There are both philosophical and practical reasons why.

### 13.5.1.1. Philosophical Motivations

*Philosophically*, one of the main reasons why Bayesian thinking is appealing is its ability to provide straightforward interpretations of statistical conclusions.

For example, when estimating an unknown quantity, a Bayesian probability interval can be directly understood as having a high probability of containing that quantity. In contrast, a Frequentist confidence interval is typically interpreted only in the context of a series of similar inferences that could be made in repeated practice. In recent years, there has been a growing emphasis on interval estimation rather than hypothesis testing in applied statistics. This shift has strengthened the Bayesian perspective since it is likely that many users of standard confidence intervals intuitively interpret them in a manner consistent with Bayesian thinking.

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 — 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.

#### 13.5.1.2. Practical Motivations

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

For most real-world problems, deriving the posterior distribution is analytically intractable and computational methods must be used. Advances in raw computing power only in the 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 financial companies trying to manage risk!

The availability of open-source libraries, such as Turing.jl, PyMC3, and Stan provide access to the core routines in an accessible interface. To get the most out of these tools requires the mindset of computational thinking described in this book - understanding model complexity, model transformations and structure, data types and program organization, etc.

#### 13.5.1.3. Advantages of the Bayesian Approach

The main advantages of this approach over traditional actuarial techniques are:

1. **Focus on distributions rather than point estimates of the posterior's mean or mode.** We are often interested in the distribution of the parameters and a focus on a single parameter estimate will 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.

#### 13.5.1.4. Challenges with the Bayesian Approach

With the Bayesian approach, there are a handful of things that are challenging. Many of the listed items are not unique to the Bayesian approach, but there are different facets of the issues that arise.

1. **Model Construction**. One must be thoughtful about the model and how variables interact. However, with the flexibility of modeling, you can apply (actuarial) science to 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.

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.

#### 13.5.1.5. Subjectivity as a Feature

**A Bayesian approach to defining a statistical model is an approach that allows for explicitly incorporating professional judgment.** Encoding assumptions into a Bayesian model forces the actuary to be explicit about otherwise fuzzy predilections. The explicit assumption is also more amenable to productive debate about its merits and biases than an implicit judgmental override.

#### 13.5.1.6. 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[9] and therefore should be similarly reconsidered.

### 13.5.2. Implications for Financial Modeling

Like Bayes' Formula itself, another aspect of financial literature that is taught but often glossed over in practice is the difference between process risk (volatility), parameter

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

risk, and model formulation risk. When performing analysis that relies on stochastic results, in practice typically only process/volatility risk is assessed.

Bayesian statistics provides the tools to help financial modelers address parameter risk and model formulation. The posterior distribution of parameters derived is consistent with the observed data and modeled relationships. This posterior distribution of parameters can then be run as an additional dimension to the risk analysis.

Additionally, best practices include skepticism of the model construction itself, and testing different formulation of the modeled relationships and variable combinations to identify models which are best fit for purpose. Tools such as Information Criterion, posterior predictive checks, Bayes factors, and other statistical diagnostics can inform the actuary about trade-offs between different choices of model.

**i Bayesian Versus Machine Learning**

Machine learning (ML) is *fully compatible* with Bayesian analysis - one can derive posterior distributions for the ML parameters like any other statistical model and the combination of approaches may be fruitful in practice.
However, to the extent that actuaries have leaned on ML approaches due to the shortcomings of traditional actuarial approaches, Bayesian modeling may provide an attractive alternative without resorting to notoriously finicky and difficult-to-explain ML models. The Bayesian framework provides an explainable model and offers several analytic extensions beyond the scope of this introductory chapter:

- Causal Modeling: Identifying not just correlated relationships, but causal ones, in contexts where a traditional designed experiment is unavailable.
- Bayes Action: Optimizing a parameter for, e.g., a CTE95 level instead of a parameter mean.
- Information Criterion: Principled techniques to compare model fit and complexity.
- Missing data: Mechanisms to handle the different kinds of missing data.
- Model averaging: Posteriors can be combined from different models to synthesize different approaches.

### 13.5.3. Basics of Bayesian Modeling

A Bayesian statistical model has four main components to focus on:

1. **Prior** encoding assumptions about the random variables related to the problem at hand, before conditioning on the data.
2. A **Model** that defines how the random variables give rise to the observed outcome.
3. **Data** that we use to update our prior assumptions.

4. **Posterior** distributions of our random variables, conditioned on the observed data and our model

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.

### 13.5.4. Markov-Chain Monte Carlo

While computing the posterior distribution for most model parameters is analytically intractable, we can probabilistically sample from the posterior distribution and achieve an approximation of the posterior distribution. MCMC samplers, as they are called, do this by moving through the parameter space and travel to different points in proportion to the posterior probability. It is a Markov-Chain because the probability of the next point's location is influenced by the prior sampling point's location.

#### 13.5.4.1. Example: MCMC from Scratch

Here is a simple example demonstrated with one of the oldest MCMC algorithm, called Metropolis-Hastings. The general idea is this:

1. Start at an arbitrary point and make that the `current_state`.
2. Propose a new point which is the `current_state` plus some movement that comes from a random distribution, `proposal_dist`.
3. Calculate the likelihood ratio of the proposed versus current point (`acceptance_ratio` below).
4. Draw a random number - if that random number is less than the `acceptance_ratio`, then move to that new point. Otherwise do not move.
5. Repeat steps 2-4 until the distribution of points converges to a stable posterior distribution.

This gets us what we desire because the resulting distribution of samples has frequency that's proportional to the posterior distribution.

We will try to find the posterior of an arbitrary set of normally distributed asset returns. We set the true, (unobserved in reality) values for $\mu$ and $\sigma$ and then draw 250 observations:

```
# In reality, we don't observe the parameters
# we are interested in determining the values for.
σ = 0.15
μ = 0.1
```

```
n_observations = 250
return_dist = Normal(μ,σ)
returns = rand(return_dist,n_observations)

# plot the distribution of returns
μ_range = LinRange(-0.5, 0.5, 400)
σ_range = LinRange(0.0, 3.0, 400)

f = Figure()
ax1 = Axis(f[1,1],title="True Distribution of Returns")
ax2 = Axis(f[2,1],title="Simulated Outcomes",xlabel="Return")
plot!(ax1,return_dist)
vlines!(ax1,[μ],color=(:black,0.7))
text!(ax1,μ,0;text="mean ($μ)",rotation=pi/2)
hist!(ax2,returns)
vlines!(ax2,[mean(returns)],color=(:black,0.7))
text!(ax2,mean(returns),0;text="mean ($(round(mean(returns);digits=3)))",rotation=pi/2)

linkxaxes!(ax1,ax2)

f
```

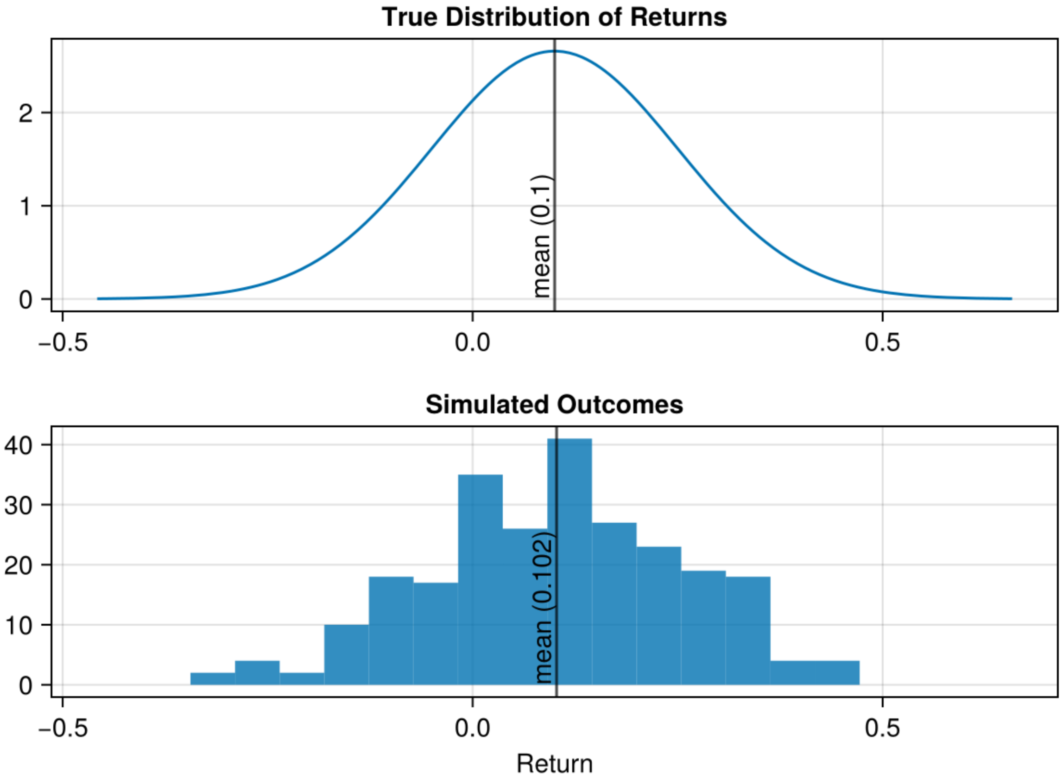



Figure 13.3.: The target probability densities which we will attempt to infer via MCMC.

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

```
# Define the proposal step distribution
proposal_std = 0.05
proposal_dist = Normal(0,proposal_std)
```

```
Normal{Float64}(μ=0.0, σ=0.05)
```

We next define how many steps we want the chain to sample for, and implement the algorithm's main loop containing the logic in the steps above.

```
# MCMC parameters
num_samples = 5000
burn_in = 500
```

```
# Define priors
μ_prior = Normal(0, 0.25)
σ_prior = Gamma(0.5)

# Initialize the Markov chain
μ_current, σ_current =  0.0, 0.25
current_prob = sum(logpdf(Normal(μ_current, σ_current), r) for r in returns) +
               logpdf(μ_prior, μ_current) +
               logpdf(σ_prior, σ_current)

chain = zeros(num_samples, 2)

count = 0
# MCMC sampling loop
while count < num_samples
    # Generate a new proposal
    μ̇, σ̇ = μ_current + rand(proposal_dist), σ_current + rand(proposal_dist)
    if σ̇ > 0

        # Calculate the acceptance ratio

        proposal_prob = sum(logpdf(Normal(μ̇, σ̇), r) for r in returns) +
                           logpdf(μ_prior, μ̇) + logpdf(σ_prior, σ̇)
        log_acceptance_ratio = proposal_prob - current_prob

        # Accept or reject the proposal
        if log(rand()) < log_acceptance_ratio
            μ_current, σ_current = μ̇, σ̇
            current_prob = proposal_prob
        end

        # Store the current state as a sample
        count += 1
        chain[count, :] .= μ_current, σ_current
    else
        # skip because σ can't be negative
    end
end

chain
```

```
5000×2 Matrix{Float64}:
 0.0        0.25
```

```
0.07303    0.173391
0.07303    0.173391
0.0967572  0.160889
0.0967572  0.160889
0.0967572  0.160889
0.0967572  0.160889
0.0967572  0.160889
0.0967572  0.160889
0.0967572  0.160889
0.0967572  0.160889
0.0967572  0.160889
0.0967572  0.160889
⋮
0.101172   0.14465
0.101172   0.14465
0.101172   0.14465
0.101172   0.14465
0.101172   0.14465
0.101172   0.14465
0.101172   0.14465
0.101172   0.14465
0.101172   0.14465
0.101172   0.14465
0.101172   0.14465
0.111746   0.150169
```

The resulting chain contains a list of points that the algorithm has moved along during the sampling process. Note that there is a `burn-in` parameter. This is because we want the chain iterate long enough to be effectively independent of both (1) the starting point for the sample, and (2) so that different chains are effectively independent.

After having performed the sampling, we can now visualize the chain versus the `target_distribution`. A few things to note:

1. The red line indicates the "warm up" or "burn-in" phase and we do not consider that as part of the sampled chain because those values are too correlated with the arbitrary starting point.
2. The blue line indicates the path traveled by the Metropolis-Hasting algorithm. Long asides into low-probability regions are possible, but in general the path will traverse areas in proportion to the probability of interest.

```
# Plot the chain
let
    f = Figure()
```

```
    # μ lines
    ax1 = Axis(f[1, 1], ylabel="σ", xticklabelsvisible=false)

    # burn in lines
    scatterlines!(ax1,chain[1:burn_in,1],
                  chain[1:burn_in,2],
                  color=(:red,0.1),
                  markercolor=(:red,0.1))

    # sampled lines
    scatterlines!(ax1,chain[burn_in+1:end,1],
                  chain[burn_in+1:end,2],
                  color=(:blue,0.1),
                  markercolor=(:blue,0.1))

    # μ histogram
    ax2 = Axis(f[2,1],xlabel="μ", yticklabelsvisible=false)
    hist!(ax2,chain[burn_in+1:end,1],color=:blue)
    linkxaxes!(ax1, ax2)

    # σ histogram
    ax3 = Axis(f[1,2],xticklabelsvisible=false)
    hist!(ax3,chain[burn_in+1:end,2],color=:blue, direction=:x)
    linkyaxes!(ax1, ax3)

    f
end
```

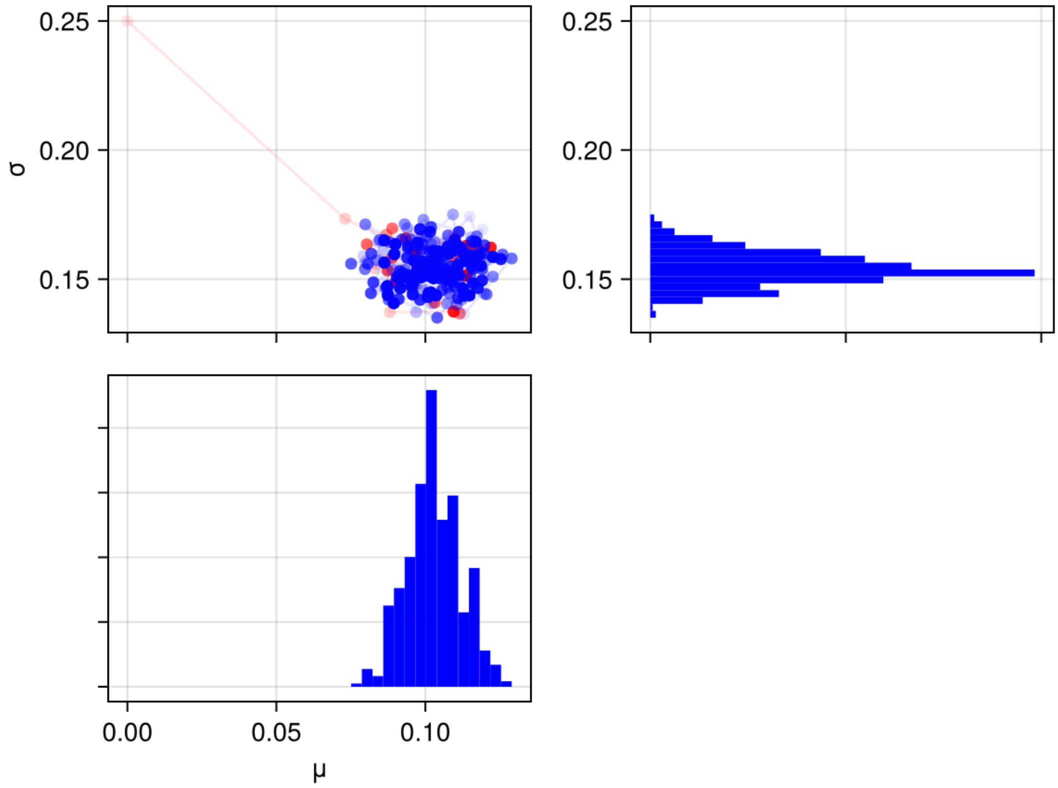



Figure 13.4.: The blue lines of the MCMC chain explore the posterior density of interest (after discarding the burn-in samples in red). Note that locaitons where the sampler remained longer (rejected more proposals) show up as darker points.

In this example, $\mu$ and $\sigma$ are independent, but if there were a correlation (such as when $mu$ were higher, $sigma$ were also higher) then the sampler would pick up on this, and we would see a skew in the plotted chain.

The point of this short introduction to MCMC is that the technique is not magic, just computationally intensive. More advanced libraries will perform the sampling for you with more advanced algorithms than Metropolis-Hastings, but by demonstrating that we could do-it-from-scratch in a small amount of code.

### 13.5.5. MCMC Algorithms

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

1. Often times the algorithm will back-track (take a "U-Turn"), wasting steps in regions already explored.
2. The algorithm can have a very high rate of rejecting proposals if the proposal mechanism generates steps that would move the current state into a low-probability regions.
3. The choice of proposal distribution and parameters can greatly influence the speed of convergence. Too large of movement and key regions can be entirely skipped over, while small movements can take much longer than necessary to explore the space.
4. As the number of parameters grows, the dimensionality of the parameter space to explore also grows making posterior exploration much harder.
5. The shape of the posterior space can be more ore less difficult to explore. Complex models may have regions of density that are not nicely "round" - regions may be curved, donut shaped, or disjointed.

The problems above mean that MCMC sampling is very computationally expensive for more complex examples. Compared with Metropolis-Hastings, modern algorithms (such as the No-U-Turn (NUTS)) algorithm explore the posterior distribution more efficiently by avoiding back-tracking to already explored regions and dynamically adjusting the proposals to adaptively fit the posterior. Many of them take direct influence from particle physics, with the algorithm keeping track of the energy of the current state as it explores the posterior space.

Algortihms have only brought so much relief to the modeler with finite resources and compute. There is still a lot of responsibility for modeler to design models that are computationally efficient, transformed to eliminate oddly-shaped density regions, or find the right simplifications to the analysis in order to make the problem tractable.

**Note**

What does it mean to transform the parameter space?
An example will be shown in Chapter 29 where we want to ensure that a binomial variable is constrained to the region $[0, 1]$ but the underlying factors are allowed to vary across the entire real numbers. We use a logit (or inverse logit, a.k.a. logistic) to transform the parameters to the required probability range for the binomial outcome.
Another common transform is "Normalizing" the data to center the data around zero and to scale the outcomes such that the sample standard deviation is equal to one.

### 13.5.6. Rainfall Example (Continued)

We will construct a Bayesian model using the Turing.jl library. Using a battle-tested library allows us to step back from the intricacies of defining our own sampler and routine and focus on the models and analysis. The goal is to fit the parameters of one of the competing models from above in order to demonstrate an MCMC analysis workflow and essential concepts.

The first thing that we will do is use Turing's `@model` macro to define a model. This has a few components:

1. The "model" is really just a Julia function that takes in data and relates the data to the statistical outcomes modeled.
2. The `~` is the syntax to either relate a parameter to a prior assumptions.
3. A loop (or broadcasted `.~`) that ties specific data observations to the random process.

Think of the `@model` block really as a model *constructor*. It isn't until we pass data to the model that you get a fully instantiated `Model` type[10].

Here's what defining the LogNormal model looks like in Turing. We have to specificy prior distirubtions for LogNormal parameters.

```julia
using Turing

@model function rainLogNormal(logdata)                                   ①

    # Prior Assumptions for the (Log) Normal Parameters
    μ ~ Normal(4,1)                                                      ②
    σ ~ Exponential(0.5)                                                 ③

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

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

① Defining the model uses the `@model` macro from Turing.
② We know that there will be positive rainfall and 96% of mean annual rainfall will be between $exp(2)$ and $exp(6)$, or 7 and 403 inches.
③ In a LogNormal model, 0.5 deviations covers a lot of variation in outcomes.

[10]Specifically: a `DynamicPPL.Model` type (PPL = Probabilistic Programming Language).

#### 13.5.6.1. Setting Priors

In the example above, we used "weakly informative" priors. We constrained the prior probability to plausible ranges, knowing enough about the system of study (rainfall) that it would be completely implausible for there to be a `Uniform(0,Inf)` distribution of mean log-rainfall total, knowing that rain can't fall in infinite quantities.

Admittedly, we haven't confirmed with a meteorologist that $exp(20)$ (485 million) inches of rain per year is impossible. But such is the beauty of the transparency of Bayesian analysis that the prior assumption is right there! Front and center and ready to be debated by other modelers!

"Strongly informative" priors would be something where we want to encode a stronger assumption about the plausible range of outcomes, such as if we knew enough about the problem domain that we could tell given the location of the rainfall, we'd expect 95% of the rainfall to be between, say, 10 and 30 inches per year.

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

#### 13.5.6.2. Sampling

Analysis should begin by evaluating the prior assumptions for reasonability and coverage over possible outcomes of the process we are trying to model. The top plot in Figure 13.8 shows the modeled rainfall outcomes taking on a wide range of possible outcomes. If we had more knowledge of the system we could enforce a stronger (narrower) prior assumption to constrain the model to a smaller set of values.

The object returned is an MCMCChains structure containing the samples as well as diagnostic information. Summary information gets printed below.

```
chain_prior = sample(m, Prior(), 1000)
```

Assessment of samples from the prior should include:

- Confirming that the model's behavior is reasonable that
- Confirming that the model covers the range of possible data that might be observed.

The sample outcomes from the modeled prior are shown in Figure 13.8.

Next, we sample the posterior by using the No-U-Turns (NUTS) algorithm and drawing 1000 samples (not including the warm-up phase). This is the primary result we will analyze further.

```
chain_posterior = sample(m,NUTS(),1000)
```

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

Iterations        = 1:1:1000
Number of chains  = 1
Samples per chain = 1000
Wall duration     = 0.21 seconds
Compute duration  = 0.21 seconds
parameters        = μ, σ
internals         = lp

Summary Statistics
  parameters      mean       std      mcse    ess_bulk   ess_tail      rhat   ⋯
      Symbol   Float64   Float64   Float64     Float64    Float64   Float64   ⋯

           μ    4.0280    1.0128    0.0309   1078.3787   984.4339    0.9998   ⋯
           σ    0.4917    0.4778    0.0162    829.5487   747.0542    0.9993   ⋯
                                                                1 column omitted

Quantiles
  parameters      2.5%     25.0%     50.0%     75.0%     97.5%
      Symbol   Float64   Float64   Float64   Float64   Float64

           μ    2.0994    3.3553    4.0316    4.7003    5.9478
           σ    0.0131    0.1524    0.3667    0.6742    1.6939
```

Figure 13.5.: Model output for the sampled prior. This isn't running an MCMC algorithm, it's simply taking draws from the defined prior assumptions.

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

Iterations        = 501:1:1500
Number of chains  = 1
Samples per chain = 1000
Wall duration     = 2.24 seconds
Compute duration  = 2.24 seconds
parameters        = μ, σ
internals      = lp, n_steps, is_accept, acceptance_rate, log_density, hamiltonian_energy, hamil

Summary Statistics
  parameters     mean      std     mcse    ess_bulk   ess_tail     rhat   ⋯
      Symbol  Float64  Float64  Float64     Float64    Float64  Float64   ⋯

           μ   3.5693   0.0557   0.0017   1121.6880   699.2394   1.0021   ⋯
           σ   0.4501   0.0382   0.0013    963.2925   560.7130   0.9999   ⋯
                                                            1 column omitted

Quantiles
  parameters      2.5%     25.0%     50.0%     75.0%     97.5%
      Symbol   Float64   Float64   Float64   Float64   Float64

           μ    3.4592    3.5352    3.5711    3.6064    3.6816
           σ    0.3834    0.4228    0.4492    0.4752    0.5337
```

Figure 13.6.: Model output for the sampled posterior.

#### 13.5.6.3. Diagnostics

Before analyzing the result itself, we should check a few things to ensure the model and sampler were well behaved. MCMC techniques are fundamentally stochastic and randomness can cause an errant sampling path. Or a model may be mis-specified such that the parameter space to explore is incompatible with the current algorithm (or any known so far).

A few things we can check:

First, the `ess` or **effective sample size** which adjusts the number of samples for the degree of autocorrelation in the chain.Ideally, we would be able to draw independent samples from the posterior, but due to the Markov-Chain approach the samples can have autocorrelation between neighboring samples. We collect less information about the posterior in the presence of positive autocorrelation.

An `ess` greater than our sample indicates that there was less (negative) autocorrelation than we would have expected for the chain. An `ess` much less than the number of samples indicates that the chain isn't sampling very efficiently but, aside from needing to run more samples, isn't necessarily a problem.

```
ess(chain_posterior)
```

```
ESS
  parameters         ess   ess_per_sec
      Symbol     Float64       Float64

           μ   1121.6880      500.7536
           σ    963.2925      430.0413
```

Second, the `rhat` ($\widehat{R}$) is the Gelman-Rubin convergence diagnostic and it's value should be very close to `1.0` for a chain that has converged properly. Even a value of 1.01 may indicate an issue and quickly gets worse for higher values.

```
rhat(chain_posterior)
```

```
R-hat
  parameters      rhat
      Symbol   Float64

           μ    1.0021
           σ    0.9999
```

Next, we can look at the “trace” plots for the parameters being sampled (Figure 13.7). These are sometimes called “hairy caterpillar” plots because in a healthy chain sample, we should see a series without autocorrelation and that the values bounce around randomly between individual samples.

```
let
    f = Figure()
    ax1 = Axis(f[1,1],ylabel="μ")
    lines!(ax1,vec(get(chain_posterior,:μ).μ.data))
    ax2 = Axis(f[2,1],ylabel="σ")
    lines!(ax2,vec(get(chain_posterior,:σ).σ.data))
    f
end
```

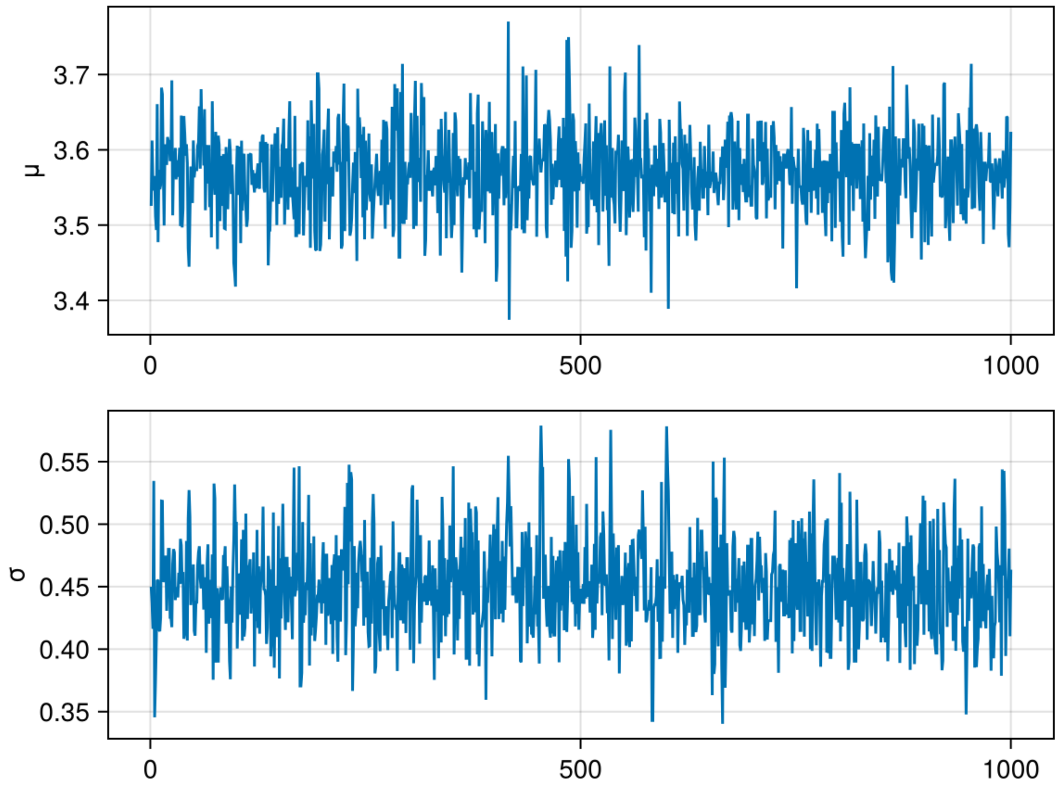



Figure 13.7.: The trace plots indicate low autocorrelation which is desirable for an MCMC sample.

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

#### 13.5.6.4. Analysis

Let's see how it looks compared to the data first. Figure 13.8 shows 200 samples from the prior and posterior. The prior (top) shows how wide the range of possible rainfall outcomes could be using our weakly informative prior assumptions. The bottom shows that after having learned from the data, the posterior probability of rainfall has narrowed considerably.

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

    # plot the actual data
    percentiles= 0.01:0.01:0.99
    lines!(axis,quantile.(Ref(rain),percentiles),percentiles,linewidth=3)
end

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

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

    linkxaxes!(ax1, ax2)

    f
end
```

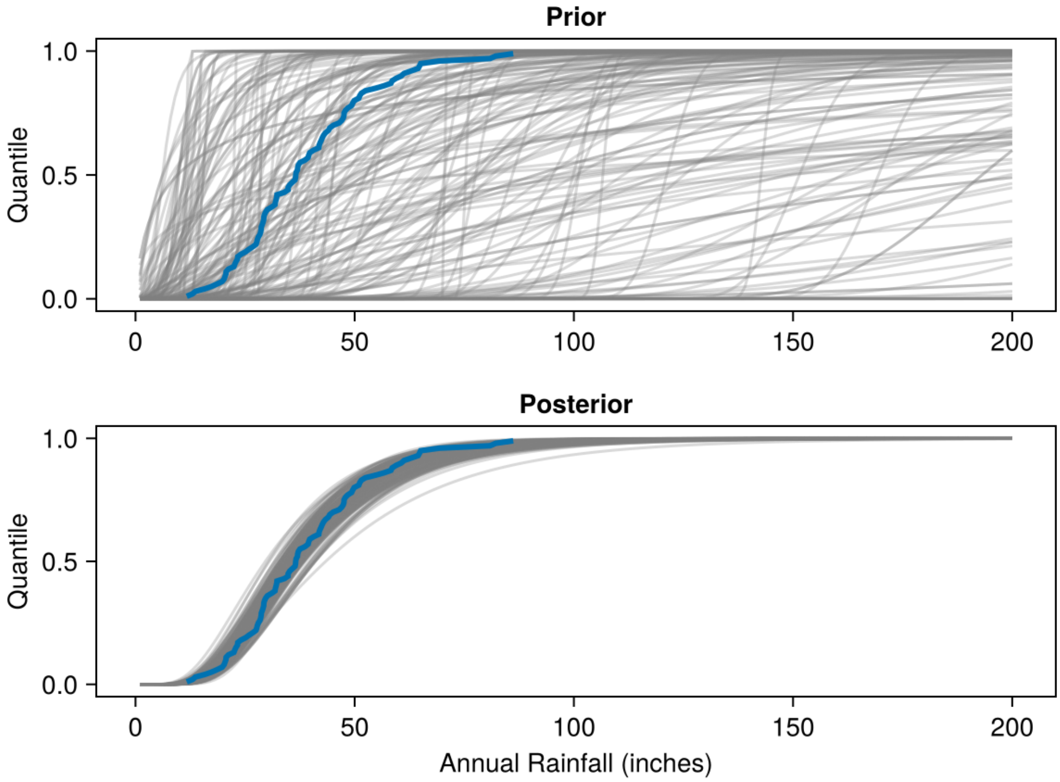



Figure 13.8.: The prior model show a wide range of possible outcomes, and the shape of the distribution is reasonable: there's a nice 'S' shape to the CDF, indicating a dense region where most outcomes would fall in the PDF. The fitted posterior model (bottom) has good coverage of the observed data (shown in blue).

Comparing to the maximum likelihood analysis from before by plotting the MLE point estimate onto the marginal densities in Figure 13.9. The peak of the the posterior is referred to as the **maximum a posteriori** (MAP) and would be the point estimate proposed by this Bayesian analysis. However, the Bayesian way of thinking about distributions of outcomes rather than point estimates is one of the main aspects we encourage for financial modelers. Using the posterior distribution of the parameters, we can assess parameter uncertainty directly instead of ignoring it as we tend to do with point estimates.

```
let
    # get the parameters from the earlier MLE approach
    p = params(ln)
```

```
    f = Figure()

    # plot μ posterior
    ax1 = Axis(f[1,1],title="μ posterior",xgridvisible=false)
    hidedecorations!(ax1)
    d = density!(ax1,vec(get(chain_posterior,:μ).μ.data))
    l = vlines!(ax1,[p[1]],color=:red)


    # plot σ posterior
    ax2 = Axis(f[2,1],title="σ posterior", xgridvisible=false)
    hidedecorations!(ax2)
    density!(ax2,vec(get(chain_posterior,:σ).σ.data))
    vlines!(ax2,[p[2]],color=:red)

    Legend(f[1,2],[d,l],["Posterior Density", "MLE Estimate"])

    f
end
```

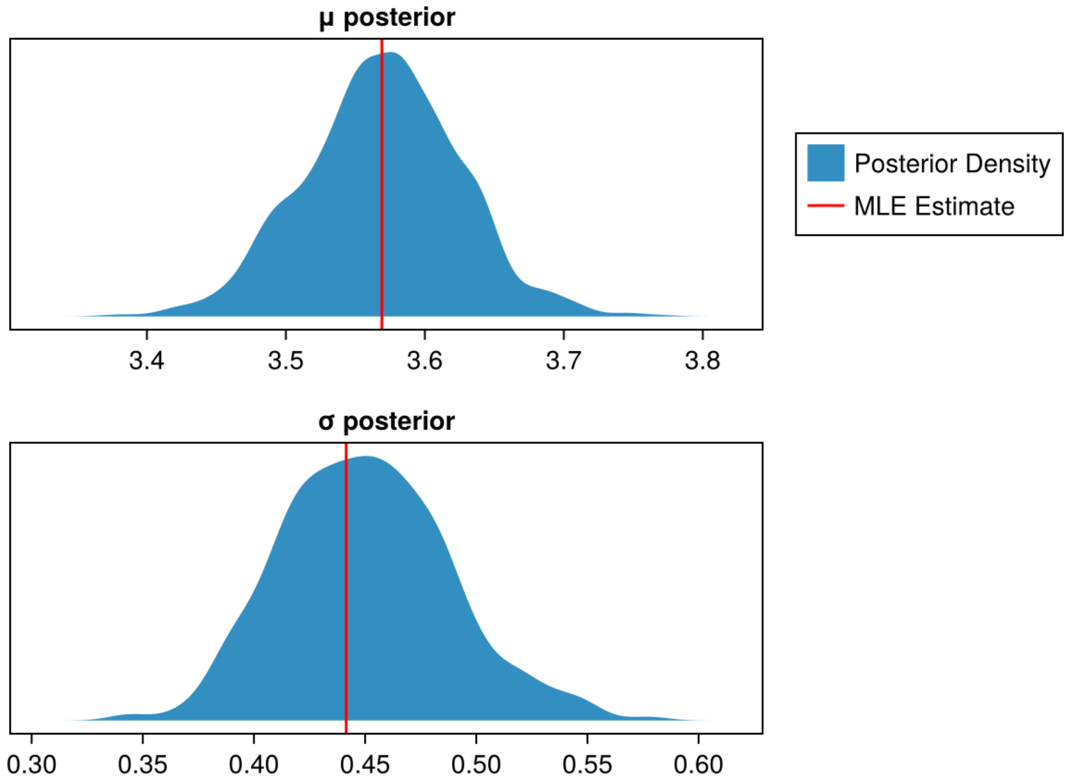



Figure 13.9.: The MLE point estimate need not necessarily align with the peak or center of posterior densities (e.g. in the case of a bimodal distribution).

#### 13.5.6.5. Model Limitations

We have built and assessed a simple statistical model that could be used in the estimation of risk for a particular location. Nowhere in our model did we define a mechanism to capture a more sophisticated view of the world. There is no parameter for changes over time due to climate change, or inter-annual seasonality for El Niño or La Niña cycles, or any of a multitude of other real-world factors that can influence the forecasting. All we've defined is that there is a LogNormal process generating rainfall in a particular location. This may or may not be sufficient to capture the dynamics of the problem at hand.

Part of the benefits of the Bayesian approach is that it allows us to extend the statistical model to be arbitrarily complex in order to capture our intended dynamics. We are limited by the availability of data, computational power and time, and our own expertise in the modeling. Regardless of the complexity of the model, the same fundamental techniques and idea apply in the Bayesian approach.

#### 13.5.6.6. Continuing the Analysis

Like any good model, you can often continue the analysis in any number of directions, such as: collecting more data, evaluating different models, creating different visualizations, making predictions about future events, creating a multi-level model that predicts rainfall for multiple related locations simultaneously, among many other

Earlier we discussed model comparison. To compute a real Bayes Factor in comparing the different models, we would take the average likelihood across the posterior samples instead of just comparing the maximum likelihood points as we did earlier. There are more sophisticated tools for estimating out of sample performance of the model, or measures that evaluate a model for over-fitting by penalizing the diagnostic statistic for the model having too many free parameters. See LOO (leave-one-out) cross-validation and various "information criteria" in the resources listed in Section 13.5.8.

### 13.5.7. Conclusion

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

### 13.5.8. Further Reading

Bayesian approaches to statistical problems are rapidly changing the professional statistical field. To the extent that the actuarial profession incorporates statistical procedures, financial professionals should consider adopting the same practices. The benefits of this are a better understanding of the distribution of risk and return, results that are more interpretable and explainable, and techniques that can be applied to a wider range of problems. The combination of these things would serve to enhance best practices related to understanding and communicating about financial quantities.

Textbooks recommended by the author are:

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

Chi Feng has an interactive demonstration of different MCMC samplers available at: https://chi-feng.github.io/mcmc-demo/.

# 14. Automatic Differentiation

## 14.1. In This Chapter

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

## 14.2. Motivation for (Automatic) Derivatives

Derivatives are one of the most useful analytical tools we have. Determining the rate of change with respect to an input is effectively sensitivity testing. Knowing the derivative lets you optimize things faster (see **?@sec-optimization**). You can test properties and implications (monotonicy, maxima/minima).

## 14.3. Finite Differentiation

Finite differentiation is evaluating a function $f(x)$ at a value $x$ and then at a nearby value $x + \epsilon$. The line drawn through these two points effectively estimates the line that is tangent to the function $f$ at $x$ - effectively the derivative has been found by approximation. That is, we are looking to approximate the derivative using the property:

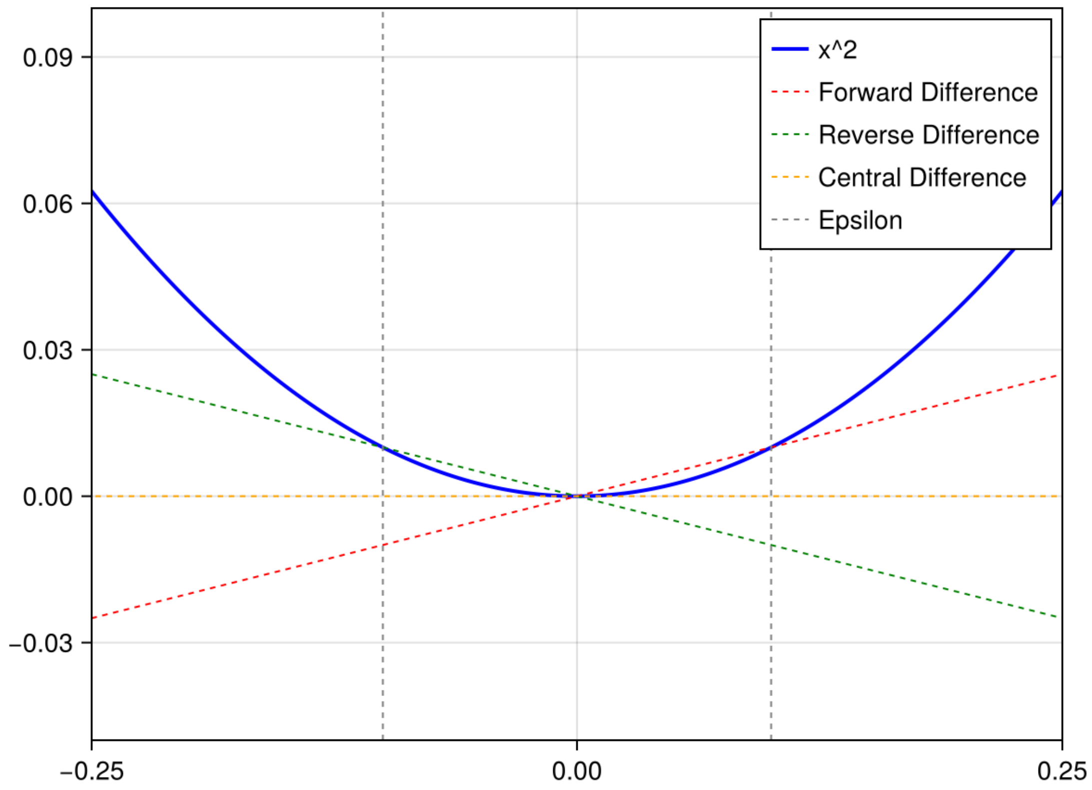
$$f'(x) = \lim_{\epsilon \to 0} \frac{f(x_0 + \epsilon) - f(x_0)}{\epsilon}$$

We can approximate the result by simply choosing a small $\epsilon$.

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

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

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 guessing an initial point, evaluating the function at that point, and determining what the derivative of the function is at that point. Both items are used to update the guess to one that's closer to the solution. This approach is used in many optimization algorithms such as Newton's Method.

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

There are additional challenges with the finite differentiation method. In practice, we are often interested in much more complex functions than $x^2$. For example, we may actually be interested in the sum of a series that is many elements long or contains more complex operations than basic algebra. In the prior example, the $\epsilon$ is set unusually wide for demonstration purposes. As $\epsilon$ grow smaller generally, the accuracy of all three finite different methods increases. However, that's not always the case due to both the complexity of the function that you may be trying to differentiate or due to numerical inaccuracies of floating point math.

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

$$f(x) = exp(x)$$

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

```
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.

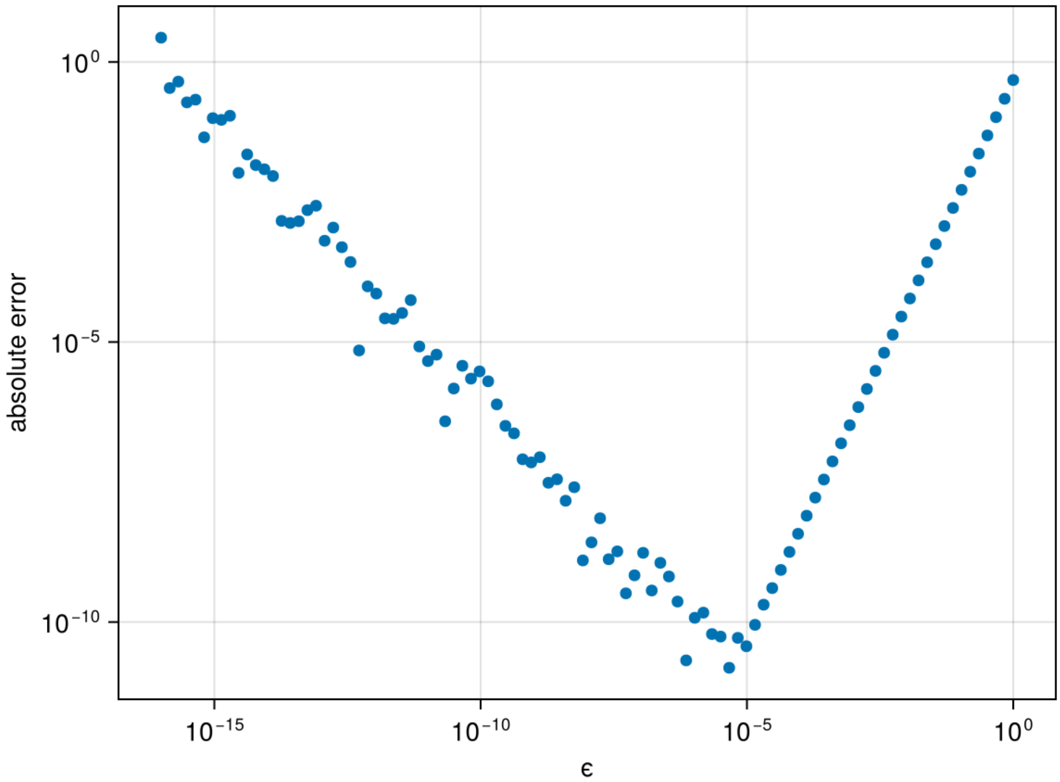



Figure 14.1.: A log-log plot showing the absolute error of the finite differences. Further to the left, roundoff error dominates while further to the right, truncation error dominates.

> **Note**
>
> The `@.` in the code example above is a macro that applies broadcasting each function to its right. `@. (f(x0 + ϵ) - f(x0 - ϵ)) / 2ϵ` is the same as `(f.(x0 .+ ϵ) .- f.(x0 .- ϵ)) ./ (2 .* ϵ)`

A few observations:

1. At virtually every value of `ϵ` we observe some error from the true derivative.
2. That error is the sum of two parts: **truncation error** is inherent in that we are using a given value for `ϵ` and not determining the limiting analytic value as $\epsilon \to 0$. 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 depending on the function and the point we are attempting to evaluate. This presents a number of practical difficulties in various algorithms.

Additionally,when computing the finite difference we must evaluate the function multiple times to determine a single estimate of the derivative. When performing something like optimization the process typically involves iteratively making many guesses — plus the number of guesses required to find the right answer can depend on the ability to accurately determine the derivative at a point!

Admittedly, despite the accuracy and computational overhead, finite differences can be very useful in many circumstances. However, a more appealing alternative approach will be covered next.

## 14.4. Automatic Differentiation

**Automatic differentiation** ("autodiff" or "AD" for short) is essentially the practice of defining algorithmically what the derivatives of 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.

### 14.4.1. Dual Numbers

To understand where we are going, let's remind ourselves about complex numbers. Complex numbers are of the form which has an real part ($r$) and an imaginary part ($iq$):

$$r + iq$$

By definition we say that $i^2 = -1$. This is useful because it allows us to perform certain types of operations (e.g. finding a square root of a negative number) that is otherwise unsolvable with just the real numbers[1]. After defining how the normal algebraic operations (addition, multiplication, etc.) work for the imaginary number, we are able to utilize the imaginary numbers for a variety of practical mathematical tasks.

What is meant by extending the algebraic operations for imaginary numbers? For example, stating how addition should work for imaginary numbers:

$$(r + iq) + (s + iu) = (r + s) + i(q + u)$$

In a similar fashion as extending the Real ($\mathbb{R}$) numbers with an *imaginary* part, for automatic differentiation we will extend them with a *dual* part. A **dual number** is one of the form:

$$a + \epsilon b$$

Where $\epsilon^2 = 0$ and $\epsilon \neq 0$ by defintion. For our purposes here, one can think of $b$ as the derivative of the function evaluated at the same point as $a$. An intial example should make this clearer. First let's define a `DualNumber`:

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

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 compared to object oriented single dispatch!):

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

```
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 * e.a, d.b * e.a + d.a * e.b)
Base.:*(x, d::DualNumber) = DualNumber(d.a * x, d.b * x)
Base.:*(d::DualNumber, x) = x * d
```

Now what if we evaluate this function:

```
f2(x) = 5 + 3x

f2(DualNumber(10, 1))
```

```
DualNumber{Int64, Int64}(35, 3)
```

We have found that the second component is `3`, which is indeed the derivative of $5 + 3x$ with respect to $x$. And in the first part we have the value of `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:

```julia
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)
```

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

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

```julia
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:

```julia
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:

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

## 14.5. 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())
```

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

```
5.192809434785093
```

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

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

```
DualNumber{Float64, Int64}(6.009544668700171, 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 effectively 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.

## 14.6. Automatic Differentiation in Practice

We have, of course, not defined an exhaustive list of operations, covering only `+`, `*`, `exp`, `sin`, and `cos`. There are only a few more arithmetic (`-`, `/`) and trancendental (`log`, more trigonometric functions, etc.) before we would have a very robust set of algebraic operations defined for our `DualNumber`. In fact, it's possible to go even further and to define the behavior through conditional expressions and iterations to differentiate fairly complex functions or to extend the mechanism to partial derivatives and higher-order derivatives as well.

```
import Distributions
import ForwardDiff

N(x) = Distributions.cdf(Distributions.Normal(), x)

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

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

"""
    eurocall(parameters)

Calculate the Black-Scholes implied option price for a european call where `parameters` is a v

- `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)
```

```
0.024933768194037 28
```

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
```

### 14.6.1. Performance

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

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

```
  26.674 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.

```
  302.477 ns (3 allocations: 704 bytes)
```

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

## 14.7. Forward Mode and Reverse Mode

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

## 14.8. Practical tips for Automatic Differentiation

Here are a few practical tips to keep in mind.

### 14.8.1. Choosing between Reverse Mode and Forward Mode

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

### 14.8.2. Mutation

Auto-differentiation works through most code, but a particularly tricky part to get right is when values within arrays are mutated (changed). It's possible to do so but may require a little bit more boilerplate to setup. As of 2024, Enzyme.jl has the best support for functions with mutation inside of them.

### 14.8.3. Custom Rules

Custom rules for new or 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.

### 14.8.4. Available Libraries

- **ForwardDiff.jl** provides robust forward-mode AD.
- **Zygote.jl** is a reverse-mode package with the innovations of being able to differentiate `structs` in addition to arrays and scalars.
- **Enzyme.jl** is a newer package which allows for both forward and reverse mode, but has the advantage of supporting array mutation. Additionally, Enzyme works at the level of LLVM code (an intermediate level between high level Julia code and machine code) which allows for different, sometimes better, optimizations.

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

## 14.9. References


- https://book.sciml.ai/notes/08-Forward-Mode_Automatic_Differentiation_(AD)_via_High_Dimension
- https://blog.esciencecenter.nl/automatic-differentiation-from-scratch-23d50c699555

# 15. Optimization

## 15.1. In This Chapter

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

## 15.2. Differentiable programming

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

Elements in differentiable programming

- Differentiable functions: Functions are defined using operations that are differentiable. These operations include basic arithmetic operations (addition, subtraction, multiplication, division), as well as more complex operations like exponentials, logarithms, trigonometric functions, etc.
- Automatic differentiation (AD): Automatic differentiation is used to compute derivatives of functions with respect to their inputs or parameters. AD exploits the fact that every computer program, no matter how complex, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division), and elementary functions (exponentials, logarithms, trigonometric functions). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program. Refer to Chapter 14 on automatic differentiation.
- Optimization and machine learning: Differentiable programming is particularly useful in optimization problems, where gradients or higher-order derivatives are required to find the minimum or maximum of a function. It's also widely used in

machine learning, where optimization algorithms like gradient descent are used to train models by adjusting their parameters to minimize a loss function.

- Gradient calculation: The gradient plays a crucial role in optimization problems primarily because it provides the direction of the steepest ascent of a function. Optimization algorithms often iteratively update parameters in the direction opposite to the gradient (for minimization problems), which tends to converge towards a local minimum (or maximum for maximization problems). Besides, computing the gradient is often computationally feasible and relatively inexpensive compared to other methods for determining function behavior, such as higher-order derivatives or function evaluations at different points. Beyond just the direction, the magnitude (or norm) of the gradient also indicates how steep the function change is in that direction. This information is used to adjust step sizes in optimization algorithms, balancing between convergence speed and stability.
- Local and global optimization: A local optimal value refers to a solution where the objective function (or cost function) has the best possible value in a neighborhood surrounding that solution. A global optimal value, on the other hand, is the best possible value of the objective function across the entire feasible domain. For smooth and convex functions, the gradient points towards the global minimum (or maximum), making it extremely efficient for finding the optimal solution. Even for non-convex functions, the gradient provides valuable information about the direction to move towards improving the objective function value locally.

Here shows the value and the derivative of a simple function at a certain point:

```
using Zygote

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

println("Value of f(x) at x=", f(x))
println("Gradient of f(x) at x=", gradient(f, x))
```

```
Value of f(x) at x=17.0
Gradient of f(x) at x=(14.0,)
```

## 15.3. Gradient-Free Optimization

This category includes algorithms that do not rely on gradients or derivative information. They often explore the objective function using heuristics or other types of probes to guide the search.

### 15.3.1. Linear optimization

Linear optimization, also known as linear programming (LP), is a mathematical method for finding the best outcome in a mathematical model with linear relationships. It involves optimizing a linear objective function subject to a set of linear equality and inequality constraints. Linear programming has a wide range of applications across various fields, including operations research, economics, engineering, and logistics.

We will use linear optimization to solve the following problem, with $n$ the number of elements in $b$:

$$\max_{x} \quad c \cdot x \text{Subject to} x \geq 0$$
$$A_i \cdot x \leq b_i \quad \forall i \in n$$

```
using JuMP, GLPK, LinearAlgebra

# Define the objective coefficients
c = [1.0, 2.0, 3.0]
# Define the constraint matrix (A) and right-hand side (b)
A = [1.0 1.0 0.0;
    0.0 1.0 1.0]
b = [10.0, 20.0]
# Create a JuMP model
linear_model = Model(GLPK.Optimizer)
# Define decision variables
@variable(linear_model, x[1:3] >= 0)
# Define objective function
@objective(linear_model, Max, dot(c, x))
# Add constraints
@constraint(linear_model, constr[i=1:2], dot(A[i, :], x) <= b[i])
# Solve the optimization problem
optimize!(linear_model)

# Print results
println("Objective value: ", objective_value(linear_model))
println("Optimal solution:")
for i in 1:3
    println("\tx[$i] = ", value(x[i]))
end
```

```
Objective value: 70.0
Optimal solution:
    x[1] = 10.0
    x[2] = 0.0
    x[3] = 20.0
```

### 15.3.2. Integer programming

Integer Programming (IP) is a type of optimization problem where some or all of the variables are restricted to be integers. Although the problem definition seems similar to an LP, the complexity of solving an IP hugely increases as the solution space is not continuous but discrete.

Let us use IP to solve this problem. A factory produces two types of products x□ and x□ with the following details:

$$\max_x 40 \cdot x_1 + 50 \cdot x_2 \text{Subject to} x_1 and x_2 are integers$$
$$4 \cdot x_1 + 3 \cdot x_2 \leq 200 (labor) x_1 + 2 \cdot x_2 \leq 40 (material)$$

```
# Import necessary packages
using JuMP, GLPK

# Create a model with the GLPK solver
model = Model(GLPK.Optimizer)

# Define decision variables (x₁ and x₂ are integers)
@variable(model, x₁ >= 0, Int)
@variable(model, x₂ >= 0, Int)

# Define the objective function (maximize profit)
@objective(model, Max, 40 * x₁ + 50 * x₂)

# Add constraints
@constraint(model, 4x₁ + 3x₂ <= 200)  # Labor constraint
@constraint(model, x₁ + 4x₂ <= 40)   # Material constraint

# Solve the model
optimize!(model)

# Check the solution status
if termination_status(model) == MOI.OPTIMAL
```

```
    println("Optimal solution found!")
    println("x₁ (Product x₁ units): ", value(x₁))
    println("x₂ (Product x₂ units): ", value(x₂))
    println("Maximum Profit: ", objective_value(model))
else
    println("No optimal solution found.")
end
```

```
Optimal solution found!
x₁ (Product x₁ units): 40.0
x₂ (Product x₂ units): 0.0
Maximum Profit: 1600.0
```

### 15.3.3. Nelder-Mead simplex method

The Nelder-Mead simplex method is a popular optimization algorithm used for minimizing (or maximizing) nonlinear functions that are not necessarily differentiable. It's particularly useful when gradient-based methods cannot be applied. It is often used in low-dimensional problems due to its simplicity and robustness.

```
using Optim, Plots

# Define the Rosenbrock function
function rosenbrock(v)
    x, y = v[1], v[2]
    return (1 - x)^2 + 100 * (y - x^2)^2
end

# Initial guess for (x, y)
initial_guess = [-1.5, 2.0]

# Perform optimization using the Nelder-Mead method
result = optimize(rosenbrock, initial_guess, NelderMead())

# Extract results
optimal_point = Optim.minimizer(result)
minimum_value = Optim.minimum(result)

println("Optimal Point: ", optimal_point)
println("Minimum Value: ", minimum_value)
```

```
Optimal Point: [0.9999913430783984, 0.9999847519696222]
Minimum Value: 5.016695917763382e-10
```

### 15.3.4. Simulated annealing

Simulated Annealing (SA) is a probabilistic optimization technique inspired by the annealing process in metallurgy. It is used to find near-optimal solutions to optimization problems, particularly in cases where traditional gradient-based methods may get stuck in local minima/maxima. SA accepts worse solutions with a certain probability, allowing it to explore the search space more broadly initially and then gradually narrow down towards better solutions as it progresses.

```
using Random

Random.seed!(1234)

# Parameters
max_iterations = 1000            # Number of iterations
initial_temperature = 100.0      # Starting temperature
cooling_rate = 0.99              # Cooling rate (temperature multiplier)
bounds = (-5.12, 5.12)           # Bounds for the search space
dimension = 5                    # Number of dimensions in the search space

# Objective function: Rastrigin function
function rastrigin(x)
    A = 10
    n = length(x)
    return A * n + sum(xi^2 - A * cos(2 * π * xi) for xi in x)
end

# Random initialization within bounds
function initialize_solution()
    return rand(bounds[1]:0.01:bounds[2], dimension)
end

# Random perturbation within bounds
function perturb_solution(solution)
    perturbed = copy(solution)
    index = rand(1:dimension)
    perturb_amount = rand(-0.1:0.01:0.1)  # Small random change
    perturbed[index] += perturb_amount
    # Ensure perturbed solution is within bounds
    perturbed[index] = clamp(perturbed[index], bounds[1], bounds[2])
    return perturbed
end
```

```
# Simulated Annealing main function
function simulated_annealing()
    current_solution = initialize_solution()
    current_value = rastrigin(current_solution)
    best_solution = copy(current_solution)
    best_value = current_value
    temperature = initial_temperature

    for iteration in 1:max_iterations
        # Generate new candidate solution by perturbation
        candidate_solution = perturb_solution(current_solution)
        candidate_value = rastrigin(candidate_solution)

        # Acceptance probability (Metropolis criterion)
        ΔE = candidate_value - current_value
        if ΔE < 0 || rand() < exp(-ΔE / temperature)
            current_solution = candidate_solution
            current_value = candidate_value
        end

        # Update best solution found so far
        if current_value < best_value
            best_solution = copy(current_solution)
            best_value = current_value
        end

        # Decrease temperature
        temperature *= cooling_rate
        println("Iteration $iteration: Best Value = $best_value, Temperature = $temperature")
    end

    return best_solution, best_value
end

# Run the simulated annealing algorithm
best_solution, best_value = simulated_annealing()
println("Best Solution: ", best_solution)
println("Best Value (Minimum): ", best_value)
```

```
Iteration 1: Best Value = 79.6456061966745, Temperature = 99.0
Iteration 2: Best Value = 79.6456061966745, Temperature = 98.01
Iteration 3: Best Value = 79.6456061966745, Temperature = 97.0299
Iteration 4: Best Value = 79.6456061966745, Temperature = 96.059601
```

```
Iteration 5: Best Value = 79.6456061966745, Temperature = 95.09900499
Iteration 6: Best Value = 79.6456061966745, Temperature = 94.1480149401
Iteration 7: Best Value = 79.6456061966745, Temperature = 93.206534790699
Iteration 8: Best Value = 79.6456061966745, Temperature = 92.27446944279201
Iteration 9: Best Value = 79.6456061966745, Temperature = 91.35172474836409
Iteration 10: Best Value = 79.6456061966745, Temperature = 90.43820750088045
Iteration 11: Best Value = 79.6456061966745, Temperature = 89.53382542587164
Iteration 12: Best Value = 79.6456061966745, Temperature = 88.63848717161292
Iteration 13: Best Value = 79.6456061966745, Temperature = 87.75210229989679
Iteration 14: Best Value = 79.6456061966745, Temperature = 86.87458127689783
Iteration 15: Best Value = 79.6456061966745, Temperature = 86.00583546412885
Iteration 16: Best Value = 79.6456061966745, Temperature = 85.14577710948755
Iteration 17: Best Value = 79.6456061966745, Temperature = 84.29431933839268
Iteration 18: Best Value = 79.6456061966745, Temperature = 83.45137614500875
Iteration 19: Best Value = 79.6456061966745, Temperature = 82.61686238355867
Iteration 20: Best Value = 79.6456061966745, Temperature = 81.79069375972308
Iteration 21: Best Value = 79.6456061966745, Temperature = 80.97278682212585
Iteration 22: Best Value = 79.6456061966745, Temperature = 80.1630589539046
Iteration 23: Best Value = 79.6456061966745, Temperature = 79.36142836436555
Iteration 24: Best Value = 79.6456061966745, Temperature = 78.56781408072189
Iteration 25: Best Value = 79.6456061966745, Temperature = 77.78213593991467
Iteration 26: Best Value = 79.6456061966745, Temperature = 77.00431458051553
Iteration 27: Best Value = 79.6456061966745, Temperature = 76.23427143471037
Iteration 28: Best Value = 79.6456061966745, Temperature = 75.47192872036327
Iteration 29: Best Value = 79.6456061966745, Temperature = 74.71720943315964
Iteration 30: Best Value = 79.6456061966745, Temperature = 73.97003733882804
Iteration 31: Best Value = 79.6456061966745, Temperature = 73.23033696543976
Iteration 32: Best Value = 79.6456061966745, Temperature = 72.49803359578536
Iteration 33: Best Value = 79.6456061966745, Temperature = 71.7730532598275
Iteration 34: Best Value = 79.6456061966745, Temperature = 71.05532272722922
Iteration 35: Best Value = 79.6456061966745, Temperature = 70.34476949995693
Iteration 36: Best Value = 79.6456061966745, Temperature = 69.64132180495736
Iteration 37: Best Value = 79.6456061966745, Temperature = 68.94490858690779
Iteration 38: Best Value = 79.6456061966745, Temperature = 68.25545950103871
Iteration 39: Best Value = 79.6456061966745, Temperature = 67.57290490602833
Iteration 40: Best Value = 79.6456061966745, Temperature = 66.89717585696805
Iteration 41: Best Value = 79.6456061966745, Temperature = 66.22820409839836
Iteration 42: Best Value = 79.6456061966745, Temperature = 65.56592205741438
Iteration 43: Best Value = 79.6456061966745, Temperature = 64.91026283684023
Iteration 44: Best Value = 79.6456061966745, Temperature = 64.26116020847184
Iteration 45: Best Value = 79.6456061966745, Temperature = 63.618548606387115
Iteration 46: Best Value = 79.6456061966745, Temperature = 62.98236312032324
Iteration 47: Best Value = 79.6456061966745, Temperature = 62.35253948912001
```

```
Iteration 48: Best Value = 79.6456061966745, Temperature = 61.72901409422881
Iteration 49: Best Value = 79.6456061966745, Temperature = 61.11172395328652
Iteration 50: Best Value = 79.6456061966745, Temperature = 60.500606713753655
Iteration 51: Best Value = 79.6456061966745, Temperature = 59.89560064661612
Iteration 52: Best Value = 79.6456061966745, Temperature = 59.29664464014996
Iteration 53: Best Value = 79.6456061966745, Temperature = 58.70367819374846
Iteration 54: Best Value = 79.6456061966745, Temperature = 58.116641411810974
Iteration 55: Best Value = 79.6456061966745, Temperature = 57.535474997692866
Iteration 56: Best Value = 79.6456061966745, Temperature = 56.960120247715935
Iteration 57: Best Value = 79.6456061966745, Temperature = 56.390519045238776
Iteration 58: Best Value = 79.6456061966745, Temperature = 55.82661385478639
Iteration 59: Best Value = 79.6456061966745, Temperature = 55.26834771623852
Iteration 60: Best Value = 79.6456061966745, Temperature = 54.71566423907614
Iteration 61: Best Value = 79.6456061966745, Temperature = 54.16850759668538
Iteration 62: Best Value = 79.6456061966745, Temperature = 53.62682252071853
Iteration 63: Best Value = 79.6456061966745, Temperature = 53.09055429551134
Iteration 64: Best Value = 79.6456061966745, Temperature = 52.55964875255623
Iteration 65: Best Value = 79.6456061966745, Temperature = 52.03405226503067
Iteration 66: Best Value = 79.6456061966745, Temperature = 51.51371174238036
Iteration 67: Best Value = 79.6456061966745, Temperature = 50.998574624956554
Iteration 68: Best Value = 79.6456061966745, Temperature = 50.488588878706985
Iteration 69: Best Value = 79.6456061966745, Temperature = 49.98370298991991
Iteration 70: Best Value = 79.6456061966745, Temperature = 49.483865960020715
Iteration 71: Best Value = 79.6456061966745, Temperature = 48.98902730042051
Iteration 72: Best Value = 79.6456061966745, Temperature = 48.4991370274163
Iteration 73: Best Value = 79.6456061966745, Temperature = 48.01414565714214
Iteration 74: Best Value = 79.6456061966745, Temperature = 47.53400420057071
Iteration 75: Best Value = 79.6456061966745, Temperature = 47.05866415856501
Iteration 76: Best Value = 79.6456061966745, Temperature = 46.58807751697936
Iteration 77: Best Value = 79.6456061966745, Temperature = 46.12219674180957
Iteration 78: Best Value = 79.6456061966745, Temperature = 45.66097477439147
Iteration 79: Best Value = 79.6456061966745, Temperature = 45.20436502664755
Iteration 80: Best Value = 79.6456061966745, Temperature = 44.752321376381076
Iteration 81: Best Value = 79.6456061966745, Temperature = 44.30479816261727
Iteration 82: Best Value = 79.6456061966745, Temperature = 43.861750180991095
Iteration 83: Best Value = 79.6456061966745, Temperature = 43.42313267918119
Iteration 84: Best Value = 79.6456061966745, Temperature = 42.98890135238938
Iteration 85: Best Value = 79.6456061966745, Temperature = 42.559012338865486
Iteration 86: Best Value = 79.6456061966745, Temperature = 42.133422215476834
Iteration 87: Best Value = 79.6456061966745, Temperature = 41.712087993322065
Iteration 88: Best Value = 79.6456061966745, Temperature = 41.29496711338884
Iteration 89: Best Value = 79.6456061966745, Temperature = 40.882017442254956
Iteration 90: Best Value = 79.6456061966745, Temperature = 40.473197267832404
```

```
Iteration 91: Best Value = 79.6456061966745, Temperature = 40.06846529515408
Iteration 92: Best Value = 79.6456061966745, Temperature = 39.66778064220254
Iteration 93: Best Value = 79.6456061966745, Temperature = 39.27110283578051
Iteration 94: Best Value = 79.6456061966745, Temperature = 38.8783918074227
Iteration 95: Best Value = 79.6456061966745, Temperature = 38.48960788934848
Iteration 96: Best Value = 79.6456061966745, Temperature = 38.104711810454994
Iteration 97: Best Value = 79.6456061966745, Temperature = 37.72366469235045
Iteration 98: Best Value = 79.6456061966745, Temperature = 37.34642804542694
Iteration 99: Best Value = 79.6456061966745, Temperature = 36.97296376497267
Iteration 100: Best Value = 79.6456061966745, Temperature = 36.60323412732294
Iteration 101: Best Value = 79.6456061966745, Temperature = 36.23720178604971
Iteration 102: Best Value = 79.6456061966745, Temperature = 35.874829768189215
Iteration 103: Best Value = 79.6456061966745, Temperature = 35.516081470507324
Iteration 104: Best Value = 79.6456061966745, Temperature = 35.16092065580225
Iteration 105: Best Value = 79.6456061966745, Temperature = 34.80931144924423
Iteration 106: Best Value = 79.6456061966745, Temperature = 34.461218334751784
Iteration 107: Best Value = 79.6456061966745, Temperature = 34.11660615140427
Iteration 108: Best Value = 79.6456061966745, Temperature = 33.775440089890225
Iteration 109: Best Value = 79.6456061966745, Temperature = 33.43768568899132
Iteration 110: Best Value = 78.46759797911986, Temperature = 33.10330883210141
Iteration 111: Best Value = 78.46759797911986, Temperature = 32.7722757437804
Iteration 112: Best Value = 75.84419432799184, Temperature = 32.44455298634259
Iteration 113: Best Value = 75.37247911306463, Temperature = 32.12010745647917
Iteration 114: Best Value = 75.37247911306463, Temperature = 31.798906381914374
Iteration 115: Best Value = 75.37247911306463, Temperature = 31.48091731809523
Iteration 116: Best Value = 75.37247911306463, Temperature = 31.16610814491428
Iteration 117: Best Value = 75.37247911306463, Temperature = 30.854447063465138
Iteration 118: Best Value = 75.37247911306463, Temperature = 30.545902592830487
Iteration 119: Best Value = 75.37247911306463, Temperature = 30.240443566902183
Iteration 120: Best Value = 75.37247911306463, Temperature = 29.93803913123316
Iteration 121: Best Value = 74.43645822840338, Temperature = 29.638658739920828
Iteration 122: Best Value = 70.58544589244562, Temperature = 29.34227215252162
Iteration 123: Best Value = 70.58544589244562, Temperature = 29.048849430996402
Iteration 124: Best Value = 70.58544589244562, Temperature = 28.758360936686437
Iteration 125: Best Value = 70.58544589244562, Temperature = 28.47077732731957
Iteration 126: Best Value = 70.58544589244562, Temperature = 28.186069554046373
Iteration 127: Best Value = 70.58544589244562, Temperature = 27.90420885850591
Iteration 128: Best Value = 70.58544589244562, Temperature = 27.625166769920853
Iteration 129: Best Value = 70.58544589244562, Temperature = 27.348915102221643
Iteration 130: Best Value = 70.58544589244562, Temperature = 27.075425951199424
Iteration 131: Best Value = 70.58544589244562, Temperature = 26.804671691687428
Iteration 132: Best Value = 70.58544589244562, Temperature = 26.536624974770554
Iteration 133: Best Value = 70.58544589244562, Temperature = 26.271258725022847
```

```
Iteration 134: Best Value = 70.58544589244562, Temperature = 26.00854613777262
Iteration 135: Best Value = 70.58544589244562, Temperature = 25.748460676394892
Iteration 136: Best Value = 70.58544589244562, Temperature = 25.490976069630943
Iteration 137: Best Value = 70.58544589244562, Temperature = 25.23606630893463
Iteration 138: Best Value = 70.58544589244562, Temperature = 24.983705645845284
Iteration 139: Best Value = 70.58544589244562, Temperature = 24.73386858938683
Iteration 140: Best Value = 70.58544589244562, Temperature = 24.48652990349296
Iteration 141: Best Value = 70.58544589244562, Temperature = 24.24166460445803
Iteration 142: Best Value = 70.58544589244562, Temperature = 23.99924795841345
Iteration 143: Best Value = 70.58544589244562, Temperature = 23.759255478829314
Iteration 144: Best Value = 70.58544589244562, Temperature = 23.52166292404102
Iteration 145: Best Value = 69.1062579592097, Temperature = 23.28644629480061
Iteration 146: Best Value = 69.1062579592097, Temperature = 23.053581831852604
Iteration 147: Best Value = 69.1062579592097, Temperature = 22.82304601353408
Iteration 148: Best Value = 69.1062579592097, Temperature = 22.59481555339874
Iteration 149: Best Value = 69.1062579592097, Temperature = 22.36886739786475
Iteration 150: Best Value = 69.1062579592097, Temperature = 22.145178723886104
Iteration 151: Best Value = 69.1062579592097, Temperature = 21.923726936647242
Iteration 152: Best Value = 69.1062579592097, Temperature = 21.704489667280768
Iteration 153: Best Value = 69.1062579592097, Temperature = 21.48744477060796
Iteration 154: Best Value = 69.1062579592097, Temperature = 21.27257032290188
Iteration 155: Best Value = 69.1062579592097, Temperature = 21.059844619672862
Iteration 156: Best Value = 69.1062579592097, Temperature = 20.849246173476132
Iteration 157: Best Value = 69.1062579592097, Temperature = 20.64075371174137
Iteration 158: Best Value = 69.1062579592097, Temperature = 20.434346174623958
Iteration 159: Best Value = 69.1062579592097, Temperature = 20.230002712877717
Iteration 160: Best Value = 69.1062579592097, Temperature = 20.027702685748938
Iteration 161: Best Value = 69.1062579592097, Temperature = 19.82742565889145
Iteration 162: Best Value = 69.1062579592097, Temperature = 19.629151402302536
Iteration 163: Best Value = 69.1062579592097, Temperature = 19.43285988827951
Iteration 164: Best Value = 69.1062579592097, Temperature = 19.238531289396715
Iteration 165: Best Value = 69.1062579592097, Temperature = 19.046145976502746
Iteration 166: Best Value = 69.1062579592097, Temperature = 18.855684516737718
Iteration 167: Best Value = 69.1062579592097, Temperature = 18.66712767157034
Iteration 168: Best Value = 69.1062579592097, Temperature = 18.480456394854638
Iteration 169: Best Value = 69.1062579592097, Temperature = 18.295651830906092
Iteration 170: Best Value = 69.1062579592097, Temperature = 18.11269531259703
Iteration 171: Best Value = 69.1062579592097, Temperature = 17.931568359471058
Iteration 172: Best Value = 69.1062579592097, Temperature = 17.752252675876345
Iteration 173: Best Value = 69.1062579592097, Temperature = 17.57473014911758
Iteration 174: Best Value = 69.1062579592097, Temperature = 17.398982847626407
Iteration 175: Best Value = 69.1062579592097, Temperature = 17.22499301915014
Iteration 176: Best Value = 69.1062579592097, Temperature = 17.05274308895864
```

```
Iteration 177: Best Value = 69.1062579592097, Temperature = 16.882215658069054
Iteration 178: Best Value = 69.1062579592097, Temperature = 16.713393501488362
Iteration 179: Best Value = 69.1062579592097, Temperature = 16.546259566473477
Iteration 180: Best Value = 69.1062579592097, Temperature = 16.380796970808742
Iteration 181: Best Value = 69.1062579592097, Temperature = 16.216989001100654
Iteration 182: Best Value = 69.1062579592097, Temperature = 16.054819111089646
Iteration 183: Best Value = 69.1062579592097, Temperature = 15.89427091997875
Iteration 184: Best Value = 69.1062579592097, Temperature = 15.735328210778961
Iteration 185: Best Value = 69.1062579592097, Temperature = 15.577974928671171
Iteration 186: Best Value = 69.1062579592097, Temperature = 15.42219517938446
Iteration 187: Best Value = 69.1062579592097, Temperature = 15.267973227590614
Iteration 188: Best Value = 69.1062579592097, Temperature = 15.115293495314708
Iteration 189: Best Value = 69.1062579592097, Temperature = 14.96414056036156
Iteration 190: Best Value = 69.1062579592097, Temperature = 14.814499154757945
Iteration 191: Best Value = 69.1062579592097, Temperature = 14.666354163210364
Iteration 192: Best Value = 69.1062579592097, Temperature = 14.519690621578262
Iteration 193: Best Value = 69.1062579592097, Temperature = 14.374493715362478
Iteration 194: Best Value = 69.1062579592097, Temperature = 14.230748778208852
Iteration 195: Best Value = 69.1062579592097, Temperature = 14.088441290426763
Iteration 196: Best Value = 69.1062579592097, Temperature = 13.947556877522496
Iteration 197: Best Value = 69.1062579592097, Temperature = 13.808081308747271
Iteration 198: Best Value = 68.74807121999402, Temperature = 13.670000495659798
Iteration 199: Best Value = 68.74807121999402, Temperature = 13.5333004907032
Iteration 200: Best Value = 68.74807121999402, Temperature = 13.397967485796167
Iteration 201: Best Value = 68.74807121999402, Temperature = 13.263987810938206
Iteration 202: Best Value = 68.74807121999402, Temperature = 13.131347932828824
Iteration 203: Best Value = 68.74807121999402, Temperature = 13.000034453500536
Iteration 204: Best Value = 68.74807121999402, Temperature = 12.87003410896553
Iteration 205: Best Value = 68.74807121999402, Temperature = 12.741333767875876
Iteration 206: Best Value = 68.74807121999402, Temperature = 12.613920430197117
Iteration 207: Best Value = 66.05131933615112, Temperature = 12.487781225895146
Iteration 208: Best Value = 66.05131933615112, Temperature = 12.362903413636195
Iteration 209: Best Value = 64.3501869050265, Temperature = 12.239274379499832
Iteration 210: Best Value = 63.09496393312524, Temperature = 12.116881635704834
Iteration 211: Best Value = 62.50489718072146, Temperature = 11.995712819347785
Iteration 212: Best Value = 62.50489718072146, Temperature = 11.875755691154307
Iteration 213: Best Value = 62.50489718072146, Temperature = 11.756998134242764
Iteration 214: Best Value = 60.57181664722005, Temperature = 11.639428152900336
Iteration 215: Best Value = 60.57181664722005, Temperature = 11.523033871371332
Iteration 216: Best Value = 54.58748194502628, Temperature = 11.407803532657619
Iteration 217: Best Value = 54.58748194502628, Temperature = 11.293725497331042
Iteration 218: Best Value = 54.11357460069095, Temperature = 11.180788242357732
Iteration 219: Best Value = 54.11357460069095, Temperature = 11.068980359934155
```

```
Iteration 220: Best Value = 54.11357460069095, Temperature = 10.958290556334813
Iteration 221: Best Value = 54.11357460069095, Temperature = 10.848707650771464
Iteration 222: Best Value = 54.11357460069095, Temperature = 10.74022057426375
Iteration 223: Best Value = 53.14093163160553, Temperature = 10.632818368521113
Iteration 224: Best Value = 53.14093163160553, Temperature = 10.526490184835902
Iteration 225: Best Value = 53.14093163160553, Temperature = 10.421225282987542
Iteration 226: Best Value = 53.14093163160553, Temperature = 10.317013030157668
Iteration 227: Best Value = 53.14093163160553, Temperature = 10.213842899856092
Iteration 228: Best Value = 52.82273685460972, Temperature = 10.11170447085753
Iteration 229: Best Value = 52.11775604439552, Temperature = 10.010587426148955
Iteration 230: Best Value = 49.80212914144378, Temperature = 9.910481551887466
Iteration 231: Best Value = 47.08214937165032, Temperature = 9.811376736368592
Iteration 232: Best Value = 45.56474621518057, Temperature = 9.713262969004907
Iteration 233: Best Value = 45.56474621518057, Temperature = 9.616130339314857
Iteration 234: Best Value = 45.56474621518057, Temperature = 9.519969035921708
Iteration 235: Best Value = 44.93264409384938, Temperature = 9.42476934556249
Iteration 236: Best Value = 44.93264409384938, Temperature = 9.330521652106865
Iteration 237: Best Value = 44.93264409384938, Temperature = 9.237216435585797
Iteration 238: Best Value = 44.93264409384938, Temperature = 9.144844271229939
Iteration 239: Best Value = 44.93264409384938, Temperature = 9.053395828517639
Iteration 240: Best Value = 44.93264409384938, Temperature = 8.962861870232462
Iteration 241: Best Value = 44.93264409384938, Temperature = 8.873233251530138
Iteration 242: Best Value = 44.8109263386411, Temperature = 8.784500919014837
Iteration 243: Best Value = 44.8109263386411, Temperature = 8.696655909824688
Iteration 244: Best Value = 44.8109263386411, Temperature = 8.609689350726441
Iteration 245: Best Value = 44.8109263386411, Temperature = 8.523592457219177
Iteration 246: Best Value = 44.8109263386411, Temperature = 8.438356532646985
Iteration 247: Best Value = 44.8109263386411, Temperature = 8.353972967320516
Iteration 248: Best Value = 44.750367343230074, Temperature = 8.27043323764731
Iteration 249: Best Value = 44.750367343230074, Temperature = 8.187728905270838
Iteration 250: Best Value = 44.51443167023366, Temperature = 8.105851616218128
Iteration 251: Best Value = 44.51443167023366, Temperature = 8.024793100055946
Iteration 252: Best Value = 44.51443167023366, Temperature = 7.944545169055387
Iteration 253: Best Value = 44.51443167023366, Temperature = 7.865099717364833
Iteration 254: Best Value = 44.51443167023366, Temperature = 7.786448720191185
Iteration 255: Best Value = 44.51443167023366, Temperature = 7.708584232989273
Iteration 256: Best Value = 44.51443167023366, Temperature = 7.63149839065938
Iteration 257: Best Value = 44.51443167023366, Temperature = 7.5551834067527865
Iteration 258: Best Value = 44.51443167023366, Temperature = 7.479631572685259
Iteration 259: Best Value = 44.51443167023366, Temperature = 7.404835256958406
Iteration 260: Best Value = 44.51443167023366, Temperature = 7.330786904388821
Iteration 261: Best Value = 44.51443167023366, Temperature = 7.257479035344933
Iteration 262: Best Value = 44.51443167023366, Temperature = 7.1849042449914835
```

```
Iteration 263: Best Value = 44.51443167023366, Temperature = 7.1130552025415685
Iteration 264: Best Value = 44.51443167023366, Temperature = 7.041924650516153
Iteration 265: Best Value = 44.51443167023366, Temperature = 6.971505404010991
Iteration 266: Best Value = 44.51443167023366, Temperature = 6.901790349970882
Iteration 267: Best Value = 44.51443167023366, Temperature = 6.832772446471173
Iteration 268: Best Value = 42.791853673977826, Temperature = 6.764444722006462
Iteration 269: Best Value = 41.992868070115456, Temperature = 6.696800274786397
Iteration 270: Best Value = 41.992868070115456, Temperature = 6.6298322720385325
Iteration 271: Best Value = 41.02798440058009, Temperature = 6.563533949318147
Iteration 272: Best Value = 41.02798440058009, Temperature = 6.497898609824966
Iteration 273: Best Value = 41.02798440058009, Temperature = 6.432919623726716
Iteration 274: Best Value = 41.02798440058009, Temperature = 6.3685904274894485
Iteration 275: Best Value = 41.02798440058009, Temperature = 6.304904523214554
Iteration 276: Best Value = 41.02798440058009, Temperature = 6.2418554779824085
Iteration 277: Best Value = 41.02798440058009, Temperature = 6.1794369232025845
Iteration 278: Best Value = 41.02798440058009, Temperature = 6.117642553970558
Iteration 279: Best Value = 41.02798440058009, Temperature = 6.056466128430853
Iteration 280: Best Value = 41.02798440058009, Temperature = 5.9959014671465445
Iteration 281: Best Value = 41.02798440058009, Temperature = 5.935942452475079
Iteration 282: Best Value = 41.02798440058009, Temperature = 5.8765830279503275
Iteration 283: Best Value = 41.02798440058009, Temperature = 5.817817197670824
Iteration 284: Best Value = 41.02798440058009, Temperature = 5.759639025694116
Iteration 285: Best Value = 37.14161585459022, Temperature = 5.7020426354371745
Iteration 286: Best Value = 36.79380820618584, Temperature = 5.645022209082803
Iteration 287: Best Value = 36.276431717701264, Temperature = 5.588571986991974
Iteration 288: Best Value = 36.276431717701264, Temperature = 5.532686267122054
Iteration 289: Best Value = 36.276431717701264, Temperature = 5.477359404450834
Iteration 290: Best Value = 36.276431717701264, Temperature = 5.422585810406325
Iteration 291: Best Value = 36.276431717701264, Temperature = 5.368359952302262
Iteration 292: Best Value = 36.276431717701264, Temperature = 5.314676352779239
Iteration 293: Best Value = 36.276431717701264, Temperature = 5.261529589251447
Iteration 294: Best Value = 36.276431717701264, Temperature = 5.208914293358933
Iteration 295: Best Value = 36.276431717701264, Temperature = 5.156825150425344
Iteration 296: Best Value = 35.732137805380304, Temperature = 5.10525689892109
Iteration 297: Best Value = 35.732137805380304, Temperature = 5.0542043299318795
Iteration 298: Best Value = 35.732137805380304, Temperature = 5.003662286632561
Iteration 299: Best Value = 35.612137805380314, Temperature = 4.953625663766235
Iteration 300: Best Value = 35.612137805380314, Temperature = 4.9040894071285726
Iteration 301: Best Value = 35.612137805380314, Temperature = 4.8550485130572865
Iteration 302: Best Value = 35.612137805380314, Temperature = 4.806498027926714
Iteration 303: Best Value = 35.494110312667196, Temperature = 4.758433047647447
Iteration 304: Best Value = 35.494110312667196, Temperature = 4.710848717170973
Iteration 305: Best Value = 35.494110312667196, Temperature = 4.663740229999263
```

```
Iteration 306: Best Value = 35.494110312667196, Temperature = 4.61710282769927
Iteration 307: Best Value = 35.494110312667196, Temperature = 4.570931799422278
Iteration 308: Best Value = 35.494110312667196, Temperature = 4.525222481428055
Iteration 309: Best Value = 35.494110312667196, Temperature = 4.4799702566137745
Iteration 310: Best Value = 35.494110312667196, Temperature = 4.4351705540476365
Iteration 311: Best Value = 35.494110312667196, Temperature = 4.39081884850716
Iteration 312: Best Value = 35.494110312667196, Temperature = 4.346910660022088
Iteration 313: Best Value = 35.494110312667196, Temperature = 4.303441553421867
Iteration 314: Best Value = 35.494110312667196, Temperature = 4.260407137887648
Iteration 315: Best Value = 35.494110312667196, Temperature = 4.217803066508771
Iteration 316: Best Value = 35.494110312667196, Temperature = 4.175625035843683
Iteration 317: Best Value = 35.494110312667196, Temperature = 4.133868785485246
Iteration 318: Best Value = 35.494110312667196, Temperature = 4.0925300976303935
Iteration 319: Best Value = 33.82921770672312, Temperature = 4.05160479665409
Iteration 320: Best Value = 33.82921770672312, Temperature = 4.011088748687548
Iteration 321: Best Value = 33.82921770672312, Temperature = 3.970977861200673
Iteration 322: Best Value = 33.82921770672312, Temperature = 3.9312680825886663
Iteration 323: Best Value = 33.82921770672312, Temperature = 3.8919554017627798
Iteration 324: Best Value = 30.69815160086543, Temperature = 3.853035847745152
Iteration 325: Best Value = 30.69815160086543, Temperature = 3.8145054892677006
Iteration 326: Best Value = 30.69815160086543, Temperature = 3.7763604343750234
Iteration 327: Best Value = 30.69815160086543, Temperature = 3.7385968300312733
Iteration 328: Best Value = 29.71653589300729, Temperature = 3.7012108617309605
Iteration 329: Best Value = 29.00995373874503, Temperature = 3.664198753113651
Iteration 330: Best Value = 29.00995373874503, Temperature = 3.6275567655825145
Iteration 331: Best Value = 29.00995373874503, Temperature = 3.591281197926689
Iteration 332: Best Value = 29.00995373874503, Temperature = 3.555368385947422
Iteration 333: Best Value = 27.351436405409864, Temperature = 3.519814702087948
Iteration 334: Best Value = 26.878754555216624, Temperature = 3.4846165550670687
Iteration 335: Best Value = 26.878754555216624, Temperature = 3.449770389516398
Iteration 336: Best Value = 26.878754555216624, Temperature = 3.415272685621234
Iteration 337: Best Value = 26.878754555216624, Temperature = 3.3811199587650216
Iteration 338: Best Value = 26.878754555216624, Temperature = 3.347308759177371
Iteration 339: Best Value = 26.878754555216624, Temperature = 3.3138356715855974
Iteration 340: Best Value = 26.878754555216624, Temperature = 3.2806973148697414
Iteration 341: Best Value = 26.878754555216624, Temperature = 3.247890341721044
Iteration 342: Best Value = 26.878754555216624, Temperature = 3.2154114383038332
Iteration 343: Best Value = 26.878754555216624, Temperature = 3.183257323920795
Iteration 344: Best Value = 26.878754555216624, Temperature = 3.1514247506815867
Iteration 345: Best Value = 26.878754555216624, Temperature = 3.1199105031747707
Iteration 346: Best Value = 26.878754555216624, Temperature = 3.088711398143023
Iteration 347: Best Value = 26.878754555216624, Temperature = 3.0578242841615926
Iteration 348: Best Value = 26.878754555216624, Temperature = 3.0272460413199767
```

```
Iteration 349: Best Value = 26.878754555216624, Temperature = 2.996973580906777
Iteration 350: Best Value = 26.878754555216624, Temperature = 2.967003845097709
Iteration 351: Best Value = 26.878754555216624, Temperature = 2.9373338066467323
Iteration 352: Best Value = 26.878754555216624, Temperature = 2.907960468580265
Iteration 353: Best Value = 26.878754555216624, Temperature = 2.8788808638944623
Iteration 354: Best Value = 26.878754555216624, Temperature = 2.8500920552555176
Iteration 355: Best Value = 26.878754555216624, Temperature = 2.8215911347029623
Iteration 356: Best Value = 26.878754555216624, Temperature = 2.7933752233559326
Iteration 357: Best Value = 26.878754555216624, Temperature = 2.765441471122373
Iteration 358: Best Value = 26.878754555216624, Temperature = 2.7377870564111495
Iteration 359: Best Value = 26.878754555216624, Temperature = 2.710409185847038
Iteration 360: Best Value = 26.878754555216624, Temperature = 2.6833050939885674
Iteration 361: Best Value = 26.878754555216624, Temperature = 2.656472043048682
Iteration 362: Best Value = 25.668466679431013, Temperature = 2.629907322618195
Iteration 363: Best Value = 25.19027190243522, Temperature = 2.603608249392013
Iteration 364: Best Value = 25.19027190243522, Temperature = 2.577572166898093
Iteration 365: Best Value = 25.19027190243522, Temperature = 2.551796445229112
Iteration 366: Best Value = 25.19027190243522, Temperature = 2.526278480776821
Iteration 367: Best Value = 25.19027190243522, Temperature = 2.501015695969053
Iteration 368: Best Value = 25.19027190243522, Temperature = 2.4760055390093623
Iteration 369: Best Value = 24.642401392390347, Temperature = 2.4512454836192688
Iteration 370: Best Value = 24.642401392390347, Temperature = 2.426733028783076
Iteration 371: Best Value = 24.642401392390347, Temperature = 2.402465698495245
Iteration 372: Best Value = 24.642401392390347, Temperature = 2.3784410415102926
Iteration 373: Best Value = 24.642401392390347, Temperature = 2.3546566310951897
Iteration 374: Best Value = 24.642401392390347, Temperature = 2.3311100647842378
Iteration 375: Best Value = 24.642401392390347, Temperature = 2.307798964136395
Iteration 376: Best Value = 24.642401392390347, Temperature = 2.2847209744950314
Iteration 377: Best Value = 23.66418971624782, Temperature = 2.261873764750081
Iteration 378: Best Value = 23.66418971624782, Temperature = 2.23925502710258
Iteration 379: Best Value = 23.66418971624782, Temperature = 2.2168624768315546
Iteration 380: Best Value = 23.66418971624782, Temperature = 2.194693852063239
Iteration 381: Best Value = 23.66418971624782, Temperature = 2.1727469135426065
Iteration 382: Best Value = 23.66418971624782, Temperature = 2.1510194444071806
Iteration 383: Best Value = 23.66418971624782, Temperature = 2.129509249963109
Iteration 384: Best Value = 23.66418971624782, Temperature = 2.108214157463478
Iteration 385: Best Value = 23.66418971624782, Temperature = 2.087132015888843
Iteration 386: Best Value = 23.66418971624782, Temperature = 2.0662606957299547
Iteration 387: Best Value = 23.66418971624782, Temperature = 2.045598088772655
Iteration 388: Best Value = 23.306323272953925, Temperature = 2.0251421078849283
Iteration 389: Best Value = 23.306323272953925, Temperature = 2.004890686806079
Iteration 390: Best Value = 23.306323272953925, Temperature = 1.9848417799380185
Iteration 391: Best Value = 23.306323272953925, Temperature = 1.9649933621386382
```

```
Iteration 392: Best Value = 22.75422115162275, Temperature = 1.9453434285172517
Iteration 393: Best Value = 22.75422115162275, Temperature = 1.9258899942320793
Iteration 394: Best Value = 22.75422115162275, Temperature = 1.9066310942897584
Iteration 395: Best Value = 22.75422115162275, Temperature = 1.8875647833468607
Iteration 396: Best Value = 22.75422115162275, Temperature = 1.868689135513392
Iteration 397: Best Value = 22.75422115162275, Temperature = 1.8500022441582582
Iteration 398: Best Value = 22.630909076712435, Temperature = 1.8315022217166756
Iteration 399: Best Value = 22.630909076712435, Temperature = 1.8131871994995088
Iteration 400: Best Value = 22.630909076712427, Temperature = 1.7950553275045138
Iteration 401: Best Value = 22.630909076712427, Temperature = 1.7771047742294686
Iteration 402: Best Value = 22.630909076712427, Temperature = 1.7593337264871738
Iteration 403: Best Value = 22.630909076712427, Temperature = 1.741740389222302
Iteration 404: Best Value = 22.630909076712427, Temperature = 1.724322985330079
Iteration 405: Best Value = 22.630909076712427, Temperature = 1.7070797554767783
Iteration 406: Best Value = 22.630909076712427, Temperature = 1.6900089579220106
Iteration 407: Best Value = 22.630909076712427, Temperature = 1.6731088683427904
Iteration 408: Best Value = 22.630909076712427, Temperature = 1.6563777796593624
Iteration 409: Best Value = 22.630909076712427, Temperature = 1.6398140018627687
Iteration 410: Best Value = 22.630909076712427, Temperature = 1.623415861844141
Iteration 411: Best Value = 22.630909076712427, Temperature = 1.6071817032256996
Iteration 412: Best Value = 22.630909076712427, Temperature = 1.5911098861934427
Iteration 413: Best Value = 22.630909076712427, Temperature = 1.5751987873315083
Iteration 414: Best Value = 22.630909076712427, Temperature = 1.5594467994581933
Iteration 415: Best Value = 22.630909076712427, Temperature = 1.5438523314636114
Iteration 416: Best Value = 22.630909076712427, Temperature = 1.5284138081489753
Iteration 417: Best Value = 22.630909076712427, Temperature = 1.5131296700674854
Iteration 418: Best Value = 22.630909076712427, Temperature = 1.4979983733668105
Iteration 419: Best Value = 22.630909076712427, Temperature = 1.4830183896331424
Iteration 420: Best Value = 22.630909076712427, Temperature = 1.468188205736811
Iteration 421: Best Value = 22.630909076712427, Temperature = 1.4535063236794428
Iteration 422: Best Value = 22.630909076712427, Temperature = 1.4389712604426483
Iteration 423: Best Value = 22.630909076712427, Temperature = 1.4245815478382218
Iteration 424: Best Value = 22.630909076712427, Temperature = 1.4103357323598396
Iteration 425: Best Value = 22.630909076712427, Temperature = 1.396232375036241
Iteration 426: Best Value = 22.630909076712427, Temperature = 1.3822700512858788
Iteration 427: Best Value = 22.630909076712427, Temperature = 1.36844735077302
Iteration 428: Best Value = 22.630909076712427, Temperature = 1.3547628772652898
Iteration 429: Best Value = 22.630909076712427, Temperature = 1.341215248492637
Iteration 430: Best Value = 22.630909076712427, Temperature = 1.3278030960077105
Iteration 431: Best Value = 22.630909076712427, Temperature = 1.3145250650476334
Iteration 432: Best Value = 22.630909076712427, Temperature = 1.301379814397157
Iteration 433: Best Value = 22.630909076712427, Temperature = 1.2883660162531856
Iteration 434: Best Value = 22.630909076712427, Temperature = 1.2754823560906536
```

```
Iteration 435: Best Value = 22.630909076712427, Temperature = 1.2627275325297471
Iteration 436: Best Value = 22.630909076712427, Temperature = 1.2501002572044497
Iteration 437: Best Value = 22.357758960565647, Temperature = 1.2375992546324053
Iteration 438: Best Value = 22.357758960565647, Temperature = 1.2252232620860812
Iteration 439: Best Value = 22.357758960565647, Temperature = 1.2129710294652205
Iteration 440: Best Value = 22.357758960565647, Temperature = 1.2008413191705682
Iteration 441: Best Value = 22.357758960565647, Temperature = 1.1888329059788625
Iteration 442: Best Value = 22.116366182140844, Temperature = 1.1769445769190738
Iteration 443: Best Value = 22.116366182140844, Temperature = 1.165175131149883
Iteration 444: Best Value = 22.116366182140844, Temperature = 1.1535233798383842
Iteration 445: Best Value = 22.116366182140844, Temperature = 1.1419881460400003
Iteration 446: Best Value = 22.116366182140844, Temperature = 1.1305682645796002
Iteration 447: Best Value = 22.116366182140844, Temperature = 1.1192625819338042
Iteration 448: Best Value = 22.116366182140844, Temperature = 1.1080699561144662
Iteration 449: Best Value = 22.116366182140844, Temperature = 1.0969892565533215
Iteration 450: Best Value = 22.116366182140844, Temperature = 1.0860193639877882
Iteration 451: Best Value = 22.116366182140844, Temperature = 1.0751591703479102
Iteration 452: Best Value = 22.116366182140844, Temperature = 1.0644075786444311
Iteration 453: Best Value = 22.116366182140844, Temperature = 1.0537635028579868
Iteration 454: Best Value = 22.116366182140844, Temperature = 1.0432258678294068
Iteration 455: Best Value = 22.116366182140844, Temperature = 1.0327936091511127
Iteration 456: Best Value = 22.116366182140844, Temperature = 1.0224656730596016
Iteration 457: Best Value = 22.116366182140844, Temperature = 1.0122410163290056
Iteration 458: Best Value = 22.116366182140844, Temperature = 1.0021186061657157
Iteration 459: Best Value = 22.116366182140844, Temperature = 0.9920974201040585
Iteration 460: Best Value = 22.116366182140844, Temperature = 0.982176445903018
Iteration 461: Best Value = 22.116366182140844, Temperature = 0.9723546814439878
Iteration 462: Best Value = 22.116366182140844, Temperature = 0.962631134629548
Iteration 463: Best Value = 22.116366182140844, Temperature = 0.9530048232832525
Iteration 464: Best Value = 22.116366182140844, Temperature = 0.94347477505042
Iteration 465: Best Value = 22.116366182140844, Temperature = 0.9340400272999158
Iteration 466: Best Value = 22.116366182140844, Temperature = 0.9246996270269165
Iteration 467: Best Value = 22.116366182140844, Temperature = 0.9154526307566474
Iteration 468: Best Value = 22.116366182140844, Temperature = 0.9062981044490809
Iteration 469: Best Value = 22.116366182140844, Temperature = 0.8972351234045901
Iteration 470: Best Value = 22.116366182140844, Temperature = 0.8882627721705442
Iteration 471: Best Value = 22.116366182140844, Temperature = 0.8793801444488387
Iteration 472: Best Value = 22.116366182140844, Temperature = 0.8705863430043503
Iteration 473: Best Value = 22.116366182140844, Temperature = 0.8618804795743068
Iteration 474: Best Value = 22.116366182140844, Temperature = 0.8532616747785637
Iteration 475: Best Value = 22.116366182140844, Temperature = 0.8447290580307781
Iteration 476: Best Value = 22.116366182140844, Temperature = 0.8362817674504703
Iteration 477: Best Value = 22.116366182140844, Temperature = 0.8279189497759656
```

```
Iteration 478: Best Value = 22.116366182140844, Temperature = 0.819639760278206
Iteration 479: Best Value = 22.116366182140844, Temperature = 0.8114433626754239
Iteration 480: Best Value = 22.116366182140844, Temperature = 0.8033289290486697
Iteration 481: Best Value = 22.116366182140844, Temperature = 0.795295639758183
Iteration 482: Best Value = 22.116366182140844, Temperature = 0.7873426833606011
Iteration 483: Best Value = 22.116366182140844, Temperature = 0.7794692565269951
Iteration 484: Best Value = 22.116366182140844, Temperature = 0.7716745639617251
Iteration 485: Best Value = 22.116366182140844, Temperature = 0.7639578183221079
Iteration 486: Best Value = 22.116366182140844, Temperature = 0.7563182401388867
Iteration 487: Best Value = 22.116366182140844, Temperature = 0.7487550577374978
Iteration 488: Best Value = 22.116366182140844, Temperature = 0.7412675071601229
Iteration 489: Best Value = 22.116366182140844, Temperature = 0.7338548320885216
Iteration 490: Best Value = 22.116366182140844, Temperature = 0.7265162837676364
Iteration 491: Best Value = 22.116366182140844, Temperature = 0.71925112092996
Iteration 492: Best Value = 22.116366182140844, Temperature = 0.7120586097206604
Iteration 493: Best Value = 22.116366182140844, Temperature = 0.7049380236234538
Iteration 494: Best Value = 22.116366182140844, Temperature = 0.6978886433872192
Iteration 495: Best Value = 22.116366182140844, Temperature = 0.690909756953347
Iteration 496: Best Value = 22.116366182140844, Temperature = 0.6840006593838135
Iteration 497: Best Value = 22.116366182140844, Temperature = 0.6771606527899754
Iteration 498: Best Value = 22.116366182140844, Temperature = 0.6703890462620756
Iteration 499: Best Value = 22.116366182140844, Temperature = 0.6636851557994549
Iteration 500: Best Value = 22.116366182140844, Temperature = 0.6570483042414603
Iteration 501: Best Value = 22.116366182140844, Temperature = 0.6504778211990457
Iteration 502: Best Value = 22.116366182140844, Temperature = 0.6439730429870553
Iteration 503: Best Value = 22.116366182140844, Temperature = 0.6375333125571847
Iteration 504: Best Value = 22.116366182140844, Temperature = 0.6311579794316129
Iteration 505: Best Value = 22.116366182140844, Temperature = 0.6248463996372967
Iteration 506: Best Value = 22.116366182140844, Temperature = 0.6185979356409237
Iteration 507: Best Value = 22.116366182140844, Temperature = 0.6124119562845145
Iteration 508: Best Value = 22.116366182140844, Temperature = 0.6062878367216693
Iteration 509: Best Value = 22.116366182140844, Temperature = 0.6002249583544526
Iteration 510: Best Value = 22.116366182140844, Temperature = 0.594222708770908
Iteration 511: Best Value = 22.116366182140844, Temperature = 0.5882804816831989
Iteration 512: Best Value = 22.116366182140844, Temperature = 0.5823976768663669
Iteration 513: Best Value = 22.116366182140844, Temperature = 0.5765737000977033
Iteration 514: Best Value = 22.116366182140844, Temperature = 0.5708079630967262
Iteration 515: Best Value = 22.116366182140844, Temperature = 0.5650998834657589
Iteration 516: Best Value = 22.116366182140844, Temperature = 0.5594488846311013
Iteration 517: Best Value = 22.116366182140844, Temperature = 0.5538543957847903
Iteration 518: Best Value = 22.116366182140844, Temperature = 0.5483158518269424
Iteration 519: Best Value = 22.116366182140844, Temperature = 0.542832693308673
Iteration 520: Best Value = 22.116366182140844, Temperature = 0.5374043663755863
```

```
Iteration 521: Best Value = 22.116366182140844, Temperature = 0.5320303227118305
Iteration 522: Best Value = 22.116366182140844, Temperature = 0.5267100194847122
Iteration 523: Best Value = 22.116366182140844, Temperature = 0.5214429192898651
Iteration 524: Best Value = 22.116366182140844, Temperature = 0.5162284900969665
Iteration 525: Best Value = 22.116366182140844, Temperature = 0.5110662051959968
Iteration 526: Best Value = 22.116366182140844, Temperature = 0.5059555431440368
Iteration 527: Best Value = 22.116366182140844, Temperature = 0.5008959877125965
Iteration 528: Best Value = 22.116366182140844, Temperature = 0.4958870278354705
Iteration 529: Best Value = 22.116366182140844, Temperature = 0.4909281575571158
Iteration 530: Best Value = 22.116366182140844, Temperature = 0.4860188759815446
Iteration 531: Best Value = 22.116366182140844, Temperature = 0.4811586872217291
Iteration 532: Best Value = 22.116366182140844, Temperature = 0.47634710034951183
Iteration 533: Best Value = 22.116366182140844, Temperature = 0.4715836293460167
Iteration 534: Best Value = 22.116366182140844, Temperature = 0.46686779305255655
Iteration 535: Best Value = 22.116366182140844, Temperature = 0.462199115122031
Iteration 536: Best Value = 22.116366182140844, Temperature = 0.4575771239708107
Iteration 537: Best Value = 22.116366182140844, Temperature = 0.4530013527311026
Iteration 538: Best Value = 22.116366182140844, Temperature = 0.44847133920379156
Iteration 539: Best Value = 22.116366182140844, Temperature = 0.44398662581175363
Iteration 540: Best Value = 22.116366182140844, Temperature = 0.4395467595536361
Iteration 541: Best Value = 22.116366182140844, Temperature = 0.4351512919580997
Iteration 542: Best Value = 22.116366182140844, Temperature = 0.4307997790385187
Iteration 543: Best Value = 22.116366182140844, Temperature = 0.4264917812481335
Iteration 544: Best Value = 22.116366182140844, Temperature = 0.42222686343565213
Iteration 545: Best Value = 22.116366182140844, Temperature = 0.4180045948012956
Iteration 546: Best Value = 22.116366182140844, Temperature = 0.4138245488532826
Iteration 547: Best Value = 22.116366182140844, Temperature = 0.4096863033647498
Iteration 548: Best Value = 22.116366182140844, Temperature = 0.40558944033110234
Iteration 549: Best Value = 22.116366182140844, Temperature = 0.40153354592779134
Iteration 550: Best Value = 22.116366182140844, Temperature = 0.39751821046851343
Iteration 551: Best Value = 22.116366182140844, Temperature = 0.3935430283638283
Iteration 552: Best Value = 22.116366182140844, Temperature = 0.38960759808019
Iteration 553: Best Value = 22.116366182140844, Temperature = 0.38571152209938814
Iteration 554: Best Value = 22.116366182140844, Temperature = 0.3818544068783942
Iteration 555: Best Value = 22.116366182140844, Temperature = 0.37803586280961027
Iteration 556: Best Value = 22.116366182140844, Temperature = 0.37425550418151415
Iteration 557: Best Value = 22.116366182140844, Temperature = 0.370512949139699
Iteration 558: Best Value = 22.116366182140844, Temperature = 0.366807819648302
Iteration 559: Best Value = 22.116366182140844, Temperature = 0.363139741451819
Iteration 560: Best Value = 22.116366182140844, Temperature = 0.3595083440373008
Iteration 561: Best Value = 22.11468680700349, Temperature = 0.3559132605969278
Iteration 562: Best Value = 22.11468680700349, Temperature = 0.3523541279909585
Iteration 563: Best Value = 22.11468680700349, Temperature = 0.3488305867110489
```

```
Iteration 564: Best Value = 22.11468680700349, Temperature = 0.34534228084393837
Iteration 565: Best Value = 22.11468680700349, Temperature = 0.34188885803549895
Iteration 566: Best Value = 22.11468680700349, Temperature = 0.33846996945514396
Iteration 567: Best Value = 22.11468680700349, Temperature = 0.3350852697605925
Iteration 568: Best Value = 22.11468680700349, Temperature = 0.33173441706298656
Iteration 569: Best Value = 22.11468680700349, Temperature = 0.3284170728923567
Iteration 570: Best Value = 22.11468680700349, Temperature = 0.32513290216343316
Iteration 571: Best Value = 22.11468680700349, Temperature = 0.3218815731417988
Iteration 572: Best Value = 22.11468680700349, Temperature = 0.31866275741038086
Iteration 573: Best Value = 22.11468680700349, Temperature = 0.31547612983627704
Iteration 574: Best Value = 22.11468680700349, Temperature = 0.31232136853791426
Iteration 575: Best Value = 22.11468680700349, Temperature = 0.3091981548525351
Iteration 576: Best Value = 22.11468680700349, Temperature = 0.30610617330400974
Iteration 577: Best Value = 22.11468680700349, Temperature = 0.30304511157096964
Iteration 578: Best Value = 22.11468680700349, Temperature = 0.30001466045525993
Iteration 579: Best Value = 22.11468680700349, Temperature = 0.2970145138507073
Iteration 580: Best Value = 22.11468680700349, Temperature = 0.29404436871220024
Iteration 581: Best Value = 22.11468680700349, Temperature = 0.29110392502507826
Iteration 582: Best Value = 22.11468680700349, Temperature = 0.2881928857748275
Iteration 583: Best Value = 22.11468680700349, Temperature = 0.2853109569170792
Iteration 584: Best Value = 22.11468680700349, Temperature = 0.2824578473479084
Iteration 585: Best Value = 22.11468680700349, Temperature = 0.2796332688744293
Iteration 586: Best Value = 22.11468680700349, Temperature = 0.276836936185685
Iteration 587: Best Value = 22.11468680700349, Temperature = 0.27406856682382813
Iteration 588: Best Value = 22.11468680700349, Temperature = 0.27132788115558987
Iteration 589: Best Value = 22.11468680700349, Temperature = 0.26861460234403395
Iteration 590: Best Value = 22.11468680700349, Temperature = 0.2659284563205936
Iteration 591: Best Value = 22.11468680700349, Temperature = 0.2632691717573876
Iteration 592: Best Value = 22.11468680700349, Temperature = 0.26063648003981377
Iteration 593: Best Value = 22.11468680700349, Temperature = 0.2580301152394156
Iteration 594: Best Value = 22.11468680700349, Temperature = 0.25544981408702144
Iteration 595: Best Value = 22.11468680700349, Temperature = 0.2528953159461512
Iteration 596: Best Value = 22.11468680700349, Temperature = 0.2503663627866897
Iteration 597: Best Value = 22.11468680700349, Temperature = 0.24786269915882284
Iteration 598: Best Value = 22.11468680700349, Temperature = 0.2453840721672346
Iteration 599: Best Value = 22.11468680700349, Temperature = 0.24293023144556225
Iteration 600: Best Value = 22.11468680700349, Temperature = 0.24050092913110663
Iteration 601: Best Value = 22.11468680700349, Temperature = 0.23809591983979556
Iteration 602: Best Value = 22.11468680700349, Temperature = 0.2357149606413976
Iteration 603: Best Value = 22.11468680700349, Temperature = 0.23335781103498363
Iteration 604: Best Value = 22.11468680700349, Temperature = 0.23102423292463378
Iteration 605: Best Value = 22.11468680700349, Temperature = 0.22871399059538744
Iteration 606: Best Value = 22.11468680700349, Temperature = 0.22642685068943355
```

```
Iteration 607: Best Value = 22.11468680700349, Temperature = 0.2241625821825392
Iteration 608: Best Value = 22.11468680700349, Temperature = 0.22192095636071382
Iteration 609: Best Value = 22.11468680700349, Temperature = 0.2197017467971067
Iteration 610: Best Value = 22.11468680700349, Temperature = 0.21750472932913562
Iteration 611: Best Value = 22.11468680700349, Temperature = 0.21532968203584427
Iteration 612: Best Value = 22.11468680700349, Temperature = 0.21317638521548582
Iteration 613: Best Value = 22.11468680700349, Temperature = 0.21104462136333096
Iteration 614: Best Value = 22.11468680700349, Temperature = 0.20893417514969764
Iteration 615: Best Value = 22.11468680700349, Temperature = 0.20684483339820067
Iteration 616: Best Value = 22.11468680700349, Temperature = 0.20477638506421866
Iteration 617: Best Value = 22.11468680700349, Temperature = 0.2027286212135765
Iteration 618: Best Value = 22.11468680700349, Temperature = 0.20070133500144072
Iteration 619: Best Value = 22.11468680700349, Temperature = 0.1986943216514263
Iteration 620: Best Value = 22.11468680700349, Temperature = 0.19670737843491204
Iteration 621: Best Value = 22.11468680700349, Temperature = 0.1947403046505629
Iteration 622: Best Value = 22.11468680700349, Temperature = 0.1927929016040573
Iteration 623: Best Value = 22.11468680700349, Temperature = 0.1908649725880167
Iteration 624: Best Value = 22.11468680700349, Temperature = 0.18895632286213654
Iteration 625: Best Value = 22.11468680700349, Temperature = 0.18706675963351518
Iteration 626: Best Value = 22.11468680700349, Temperature = 0.18519609203718004
Iteration 627: Best Value = 22.11468680700349, Temperature = 0.18334413111680825
Iteration 628: Best Value = 22.11468680700349, Temperature = 0.18151068980564017
Iteration 629: Best Value = 22.11468680700349, Temperature = 0.17969558290758378
Iteration 630: Best Value = 22.11468680700349, Temperature = 0.17789862707850793
Iteration 631: Best Value = 22.11468680700349, Temperature = 0.17611964080772285
Iteration 632: Best Value = 22.11468680700349, Temperature = 0.17435844439964562
Iteration 633: Best Value = 22.11468680700349, Temperature = 0.17261485995564915
Iteration 634: Best Value = 22.11468680700349, Temperature = 0.17088871135609265
Iteration 635: Best Value = 22.11468680700349, Temperature = 0.16917982424253172
Iteration 636: Best Value = 22.11468680700349, Temperature = 0.1674880260001064
Iteration 637: Best Value = 22.11468680700349, Temperature = 0.16581314574010533
Iteration 638: Best Value = 22.11468680700349, Temperature = 0.16415501428270426
Iteration 639: Best Value = 22.11468680700349, Temperature = 0.16251346413987722
Iteration 640: Best Value = 22.11468680700349, Temperature = 0.16088832949847845
Iteration 641: Best Value = 22.11468680700349, Temperature = 0.15927944620349366
Iteration 642: Best Value = 22.11468680700349, Temperature = 0.1576866517414587
Iteration 643: Best Value = 22.11468680700349, Temperature = 0.15610978522404412
Iteration 644: Best Value = 22.11468680700349, Temperature = 0.15454868737180366
Iteration 645: Best Value = 22.11468680700349, Temperature = 0.15300320049808563
Iteration 646: Best Value = 22.11468680700349, Temperature = 0.15147316849310477
Iteration 647: Best Value = 22.11468680700349, Temperature = 0.14995843680817372
Iteration 648: Best Value = 22.11468680700349, Temperature = 0.148458852440092
Iteration 649: Best Value = 22.11468680700349, Temperature = 0.14697426391569107
```

```
Iteration 650: Best Value = 22.11468680700349, Temperature = 0.14550452127653415
Iteration 651: Best Value = 22.11468680700349, Temperature = 0.1440494760637688
Iteration 652: Best Value = 22.11468680700349, Temperature = 0.14260898130313113
Iteration 653: Best Value = 22.11468680700349, Temperature = 0.14118289149009983
Iteration 654: Best Value = 22.11468680700349, Temperature = 0.13977106257519883
Iteration 655: Best Value = 22.11468680700349, Temperature = 0.13837335194944683
Iteration 656: Best Value = 22.11468680700349, Temperature = 0.13698961842995236
Iteration 657: Best Value = 22.11468680700349, Temperature = 0.13561972224565283
Iteration 658: Best Value = 22.11468680700349, Temperature = 0.1342635250231963
Iteration 659: Best Value = 22.11468680700349, Temperature = 0.13292088977296435
Iteration 660: Best Value = 22.11468680700349, Temperature = 0.1315916808752347
Iteration 661: Best Value = 22.11468680700349, Temperature = 0.13027576406648236
Iteration 662: Best Value = 22.05908570257249, Temperature = 0.12897300642581755
Iteration 663: Best Value = 22.05908570257249, Temperature = 0.12768327636155938
Iteration 664: Best Value = 22.05908570257249, Temperature = 0.12640644359794379
Iteration 665: Best Value = 22.05908570257249, Temperature = 0.12514237916196436
Iteration 666: Best Value = 22.05908570257249, Temperature = 0.12389095537034471
Iteration 667: Best Value = 22.05908570257249, Temperature = 0.12265204581664126
Iteration 668: Best Value = 21.99711319528562, Temperature = 0.12142552535847484
Iteration 669: Best Value = 21.99711319528562, Temperature = 0.12021127010489009
Iteration 670: Best Value = 21.99711319528562, Temperature = 0.11900915740384119
Iteration 671: Best Value = 21.99711319528562, Temperature = 0.11781906582980277
Iteration 672: Best Value = 21.99711319528562, Temperature = 0.11664087517150475
Iteration 673: Best Value = 21.99711319528562, Temperature = 0.1154744664197897
Iteration 674: Best Value = 21.99711319528562, Temperature = 0.1143197217555918
Iteration 675: Best Value = 21.99711319528562, Temperature = 0.11317652453803588
Iteration 676: Best Value = 21.99711319528562, Temperature = 0.11204475929265552
Iteration 677: Best Value = 21.99711319528562, Temperature = 0.11092431169972897
Iteration 678: Best Value = 21.99711319528562, Temperature = 0.10981506858273167
Iteration 679: Best Value = 21.99711319528562, Temperature = 0.10871691789690435
Iteration 680: Best Value = 21.99711319528562, Temperature = 0.1076297487179353
Iteration 681: Best Value = 21.995433820148268, Temperature = 0.10655345123075595
Iteration 682: Best Value = 21.995433820148268, Temperature = 0.10548791671844839
Iteration 683: Best Value = 21.995433820148268, Temperature = 0.1044330375512639
Iteration 684: Best Value = 21.995433820148268, Temperature = 0.10338870717575126
Iteration 685: Best Value = 21.995266535865557, Temperature = 0.10235482010399374
Iteration 686: Best Value = 21.995266535865557, Temperature = 0.10133127190295381
Iteration 687: Best Value = 21.995266535865557, Temperature = 0.10031795918392428
Iteration 688: Best Value = 21.995266535865557, Temperature = 0.09931477959208504
Iteration 689: Best Value = 21.995266535865557, Temperature = 0.09832163179616418
Iteration 690: Best Value = 21.995266535865557, Temperature = 0.09733841547820254
Iteration 691: Best Value = 21.995266535865557, Temperature = 0.09636503132342052
Iteration 692: Best Value = 21.995266535865557, Temperature = 0.0954013810101863
```

```
Iteration 693: Best Value = 21.995266535865557, Temperature = 0.09444736720008444
Iteration 694: Best Value = 21.995266535865557, Temperature = 0.0935028935280836
Iteration 695: Best Value = 21.995266535865557, Temperature = 0.09256786459280275
Iteration 696: Best Value = 21.995266535865557, Temperature = 0.09164218594687472
Iteration 697: Best Value = 21.995266535865557, Temperature = 0.09072576408740597
Iteration 698: Best Value = 21.995266535865557, Temperature = 0.08981850644653191
Iteration 699: Best Value = 21.995266535865557, Temperature = 0.0889203213820666
Iteration 700: Best Value = 21.995266535865557, Temperature = 0.08803111816824594
Iteration 701: Best Value = 21.995266535865557, Temperature = 0.08715080698656348
Iteration 702: Best Value = 21.995266535865557, Temperature = 0.08627929891669785
Iteration 703: Best Value = 21.995266535865557, Temperature = 0.08541650592753086
Iteration 704: Best Value = 21.995266535865557, Temperature = 0.08456234086825555
Iteration 705: Best Value = 21.995266535865557, Temperature = 0.083716717459573
Iteration 706: Best Value = 21.995266535865557, Temperature = 0.08287955028497727
Iteration 707: Best Value = 21.995266535865557, Temperature = 0.0820507547821275
Iteration 708: Best Value = 21.995266535865557, Temperature = 0.08123024723430623
Iteration 709: Best Value = 21.995266535865557, Temperature = 0.08041794476196316
Iteration 710: Best Value = 21.995266535865557, Temperature = 0.07961376531434353
Iteration 711: Best Value = 21.995266535865557, Temperature = 0.0788176276612001
Iteration 712: Best Value = 21.995266535865557, Temperature = 0.0780294513845881
Iteration 713: Best Value = 21.995266535865557, Temperature = 0.07724915687074221
Iteration 714: Best Value = 21.995266535865557, Temperature = 0.07647666530203479
Iteration 715: Best Value = 21.995266535865557, Temperature = 0.07571189864901444
Iteration 716: Best Value = 21.995266535865557, Temperature = 0.0749547796625243
Iteration 717: Best Value = 21.995266535865557, Temperature = 0.07420523186589906
Iteration 718: Best Value = 21.995266535865557, Temperature = 0.07346317954724006
Iteration 719: Best Value = 21.995266535865557, Temperature = 0.07272854775176767
Iteration 720: Best Value = 21.995266535865557, Temperature = 0.07200126227425
Iteration 721: Best Value = 21.995266535865557, Temperature = 0.07128124965150749
Iteration 722: Best Value = 21.995266535865557, Temperature = 0.07056843715499242
Iteration 723: Best Value = 21.995266535865557, Temperature = 0.06986275278344249
Iteration 724: Best Value = 21.995266535865557, Temperature = 0.06916412525560807
Iteration 725: Best Value = 21.995266535865557, Temperature = 0.06847248400305199
Iteration 726: Best Value = 21.995266535865557, Temperature = 0.06778775916302147
Iteration 727: Best Value = 21.995266535865557, Temperature = 0.06710988157139126
Iteration 728: Best Value = 21.93833868942773, Temperature = 0.06643878275567736
Iteration 729: Best Value = 21.93833868942773, Temperature = 0.06577439492812058
Iteration 730: Best Value = 21.93833868942773, Temperature = 0.06511665097883937
Iteration 731: Best Value = 21.93833868942773, Temperature = 0.06446548446905097
Iteration 732: Best Value = 21.93833868942773, Temperature = 0.06382082962436046
Iteration 733: Best Value = 21.93833868942773, Temperature = 0.06318262132811686
Iteration 734: Best Value = 21.93833868942773, Temperature = 0.06255079511483569
Iteration 735: Best Value = 21.93833868942773, Temperature = 0.06192528716368733
```

```
Iteration 736: Best Value = 21.93833868942773, Temperature = 0.061306034292050456
Iteration 737: Best Value = 21.93833868942773, Temperature = 0.06069297394912995
Iteration 738: Best Value = 21.93833868942773, Temperature = 0.06008604420963865
Iteration 739: Best Value = 21.93833868942773, Temperature = 0.05948518376754226
Iteration 740: Best Value = 21.93833868942773, Temperature = 0.05889033192986684
Iteration 741: Best Value = 21.93833868942773, Temperature = 0.058301428610568175
Iteration 742: Best Value = 21.93833868942773, Temperature = 0.057718414324462494
Iteration 743: Best Value = 21.93833868942773, Temperature = 0.05714123018121787
Iteration 744: Best Value = 21.93833868942773, Temperature = 0.05656981787940569
Iteration 745: Best Value = 21.93833868942773, Temperature = 0.05600411970061163
Iteration 746: Best Value = 21.93833868942773, Temperature = 0.05544407850360551
Iteration 747: Best Value = 21.93833868942773, Temperature = 0.05488963771856946
Iteration 748: Best Value = 21.93833868942773, Temperature = 0.054340741341383766
Iteration 749: Best Value = 21.93833868942773, Temperature = 0.053797333927969926
Iteration 750: Best Value = 21.93833868942773, Temperature = 0.05325936058869023
Iteration 751: Best Value = 21.93833868942773, Temperature = 0.052726766982803326
Iteration 752: Best Value = 21.93833868942773, Temperature = 0.052199499312975295
Iteration 753: Best Value = 21.93833868942773, Temperature = 0.05167750431984554
Iteration 754: Best Value = 21.93833868942773, Temperature = 0.05116072927664708
Iteration 755: Best Value = 21.93833868942773, Temperature = 0.050649121983880614
Iteration 756: Best Value = 21.93833868942773, Temperature = 0.05014263076404181
Iteration 757: Best Value = 21.93833868942773, Temperature = 0.04964120445640139
Iteration 758: Best Value = 21.93833868942773, Temperature = 0.049144792411837376
Iteration 759: Best Value = 21.93833868942773, Temperature = 0.048653344487719005
Iteration 760: Best Value = 21.93833868942773, Temperature = 0.04816681104284182
Iteration 761: Best Value = 21.93833868942773, Temperature = 0.047685142932413396
Iteration 762: Best Value = 21.93833868942773, Temperature = 0.047208291503089264
Iteration 763: Best Value = 21.93833868942773, Temperature = 0.046736208588058374
Iteration 764: Best Value = 21.93833868942773, Temperature = 0.04626884650217779
Iteration 765: Best Value = 21.93833868942773, Temperature = 0.04580615803715601
Iteration 766: Best Value = 21.93833868942773, Temperature = 0.04534809645678445
Iteration 767: Best Value = 21.93833868942773, Temperature = 0.04489461549221661
Iteration 768: Best Value = 21.93833868942773, Temperature = 0.04444566933729444
Iteration 769: Best Value = 21.93833868942773, Temperature = 0.0440012126439215
Iteration 770: Best Value = 21.93833868942773, Temperature = 0.04356120051748228
Iteration 771: Best Value = 21.93833868942773, Temperature = 0.04312558851230746
Iteration 772: Best Value = 21.93833868942773, Temperature = 0.04269433262718438
Iteration 773: Best Value = 21.93833868942773, Temperature = 0.04226738930091254
Iteration 774: Best Value = 21.93833868942773, Temperature = 0.04184471540790342
Iteration 775: Best Value = 21.93833868942773, Temperature = 0.04142626825382438
Iteration 776: Best Value = 21.93833868942773, Temperature = 0.041012005571286134
Iteration 777: Best Value = 21.93833868942773, Temperature = 0.04060188551557327
Iteration 778: Best Value = 21.93833868942773, Temperature = 0.04019586666041754
```

```
Iteration 779: Best Value = 21.93833868942773, Temperature = 0.03979390799381336
Iteration 780: Best Value = 21.93833868942773, Temperature = 0.03939596891387523
Iteration 781: Best Value = 21.93833868942773, Temperature = 0.03900200922473648
Iteration 782: Best Value = 21.93833868942773, Temperature = 0.03861198913248911
Iteration 783: Best Value = 21.93833868942773, Temperature = 0.038225869241164216
Iteration 784: Best Value = 21.93833868942773, Temperature = 0.03784361054875257
Iteration 785: Best Value = 21.93833868942773, Temperature = 0.03746517444326505
Iteration 786: Best Value = 21.93833868942773, Temperature = 0.037090522698832395
Iteration 787: Best Value = 21.93833868942773, Temperature = 0.036719617471844074
Iteration 788: Best Value = 21.93833868942773, Temperature = 0.03635242129712563
Iteration 789: Best Value = 21.93833868942773, Temperature = 0.03598889708415438
Iteration 790: Best Value = 21.93833868942773, Temperature = 0.035629008113312835
Iteration 791: Best Value = 21.93833868942773, Temperature = 0.035272718032179705
Iteration 792: Best Value = 21.93833868942773, Temperature = 0.03491999085185791
Iteration 793: Best Value = 21.93833868942773, Temperature = 0.03457079094333933
Iteration 794: Best Value = 21.93833868942773, Temperature = 0.03422508303390594
Iteration 795: Best Value = 21.93833868942773, Temperature = 0.03388283220356688
Iteration 796: Best Value = 21.93833868942773, Temperature = 0.03354400388153121
Iteration 797: Best Value = 21.93833868942773, Temperature = 0.0332085638427159
Iteration 798: Best Value = 21.93833868942773, Temperature = 0.03287647820428874
Iteration 799: Best Value = 21.93833868942773, Temperature = 0.03254771342224585
Iteration 800: Best Value = 21.93833868942773, Temperature = 0.03222223628802339
Iteration 801: Best Value = 21.93833868942773, Temperature = 0.031900013925143154
Iteration 802: Best Value = 21.93833868942773, Temperature = 0.031581013785891725
Iteration 803: Best Value = 21.93833868942773, Temperature = 0.031265203648032806
Iteration 804: Best Value = 21.93833868942773, Temperature = 0.030952551611552477
Iteration 805: Best Value = 21.93833868942773, Temperature = 0.030643026095436954
Iteration 806: Best Value = 21.93833868942773, Temperature = 0.030336595834482585
Iteration 807: Best Value = 21.93833868942773, Temperature = 0.030033229876137758
Iteration 808: Best Value = 21.93833868942773, Temperature = 0.02973289757737638
Iteration 809: Best Value = 21.93833868942773, Temperature = 0.029435568601602617
Iteration 810: Best Value = 21.93833868942773, Temperature = 0.02914121291558659 2
Iteration 811: Best Value = 21.93833868942773, Temperature = 0.028849800786430724
Iteration 812: Best Value = 21.93833868942773, Temperature = 0.028561302778566416
Iteration 813: Best Value = 21.93833868942773, Temperature = 0.028275689750780753
Iteration 814: Best Value = 21.93833868942773, Temperature = 0.027992932853272947
Iteration 815: Best Value = 21.93833868942773, Temperature = 0.027713003524740216
Iteration 816: Best Value = 21.93833868942773, Temperature = 0.027435873489492814
Iteration 817: Best Value = 21.93833868942773, Temperature = 0.027161514754597885
Iteration 818: Best Value = 21.93833868942773, Temperature = 0.026889899607051907
Iteration 819: Best Value = 21.93833868942773, Temperature = 0.02662100061098139
Iteration 820: Best Value = 21.93833868942773, Temperature = 0.026354790604871576
Iteration 821: Best Value = 21.93833868942773, Temperature = 0.02609124269882286
```

```
Iteration 822: Best Value = 21.93833868942773, Temperature = 0.025830330271834633
Iteration 823: Best Value = 21.93833868942773, Temperature = 0.02557202696911629
Iteration 824: Best Value = 21.93833868942773, Temperature = 0.025316306699425126
Iteration 825: Best Value = 21.93833868942773, Temperature = 0.025063143632430875
Iteration 826: Best Value = 21.93833868942773, Temperature = 0.024812512196106567
Iteration 827: Best Value = 21.93833868942773, Temperature = 0.024564387074145502
Iteration 828: Best Value = 21.93833868942773, Temperature = 0.024318743203404047
Iteration 829: Best Value = 21.93833868942773, Temperature = 0.024075555771370008
Iteration 830: Best Value = 21.93833868942773, Temperature = 0.023834800213656308
Iteration 831: Best Value = 21.918338689427735, Temperature = 0.023596452211519743
Iteration 832: Best Value = 21.918338689427735, Temperature = 0.023360487689404547
Iteration 833: Best Value = 21.918338689427735, Temperature = 0.0231268828125105
Iteration 834: Best Value = 21.918338689427735, Temperature = 0.022895613984385395
Iteration 835: Best Value = 21.918338689427735, Temperature = 0.02266665784454154
Iteration 836: Best Value = 21.918338689427735, Temperature = 0.022439991266096124
Iteration 837: Best Value = 21.918338689427735, Temperature = 0.022215591353435163
Iteration 838: Best Value = 21.918338689427735, Temperature = 0.021993435439900812
Iteration 839: Best Value = 21.918338689427735, Temperature = 0.021773501085501804
Iteration 840: Best Value = 21.918338689427735, Temperature = 0.021555766074646786
Iteration 841: Best Value = 21.918338689427735, Temperature = 0.021340208413900316
Iteration 842: Best Value = 21.918338689427735, Temperature = 0.021126806329761313
Iteration 843: Best Value = 21.918338689427735, Temperature = 0.0209155382664637
Iteration 844: Best Value = 21.918338689427735, Temperature = 0.02070638288379906
Iteration 845: Best Value = 21.918338689427735, Temperature = 0.02049931905496107
Iteration 846: Best Value = 21.918338689427735, Temperature = 0.020294325864411458
Iteration 847: Best Value = 21.918338689427735, Temperature = 0.020091382605767344
Iteration 848: Best Value = 21.918338689427735, Temperature = 0.01989046877970967
Iteration 849: Best Value = 21.918338689427735, Temperature = 0.019691564091912573
Iteration 850: Best Value = 21.918338689427735, Temperature = 0.019494648450993447
Iteration 851: Best Value = 21.918338689427735, Temperature = 0.019299701966483514
Iteration 852: Best Value = 21.918338689427735, Temperature = 0.019106704946818678
Iteration 853: Best Value = 21.918338689427735, Temperature = 0.01891563789735049
Iteration 854: Best Value = 21.918338689427735, Temperature = 0.018726481518376987
Iteration 855: Best Value = 21.918338689427735, Temperature = 0.018539216703193216
Iteration 856: Best Value = 21.918338689427735, Temperature = 0.018353824536161283
Iteration 857: Best Value = 21.918338689427735, Temperature = 0.01817028629079967
Iteration 858: Best Value = 21.918338689427735, Temperature = 0.017988583427891672
Iteration 859: Best Value = 21.918338689427735, Temperature = 0.017808697593612755
Iteration 860: Best Value = 21.918338689427735, Temperature = 0.017630610617676627
Iteration 861: Best Value = 21.918338689427735, Temperature = 0.01745430451149986
Iteration 862: Best Value = 21.918338689427735, Temperature = 0.017279761466384862
Iteration 863: Best Value = 21.918338689427735, Temperature = 0.017106963851721012
Iteration 864: Best Value = 21.918338689427735, Temperature = 0.016935894213203802
```

```
Iteration 865: Best Value = 21.918338689427735, Temperature = 0.016766535271071765
Iteration 866: Best Value = 21.918338689427735, Temperature = 0.016598869918361046
Iteration 867: Best Value = 21.918338689427735, Temperature = 0.016432881219177436
Iteration 868: Best Value = 21.918338689427735, Temperature = 0.016268552406985663
Iteration 869: Best Value = 21.918338689427735, Temperature = 0.016105866882915806
Iteration 870: Best Value = 21.918338689427735, Temperature = 0.01594480821408665
Iteration 871: Best Value = 21.918338689427735, Temperature = 0.015785360131945782
Iteration 872: Best Value = 21.918338689427735, Temperature = 0.015627506530626323
Iteration 873: Best Value = 21.918338689427735, Temperature = 0.01547123146532006
Iteration 874: Best Value = 21.918338689427735, Temperature = 0.01531651915066686
Iteration 875: Best Value = 21.918338689427735, Temperature = 0.015163353959160191
Iteration 876: Best Value = 21.918338689427735, Temperature = 0.01501172041956859
Iteration 877: Best Value = 21.918338689427735, Temperature = 0.014861603215372903
Iteration 878: Best Value = 21.918338689427735, Temperature = 0.014712987183219174
Iteration 879: Best Value = 21.918338689427735, Temperature = 0.014565857311386982
Iteration 880: Best Value = 21.918338689427735, Temperature = 0.014420198738273111
Iteration 881: Best Value = 21.918338689427735, Temperature = 0.01427599675089038
Iteration 882: Best Value = 21.918338689427735, Temperature = 0.014133236783381476
Iteration 883: Best Value = 21.918338689427735, Temperature = 0.013991904415547662
Iteration 884: Best Value = 21.918338689427735, Temperature = 0.013851985371392185
Iteration 885: Best Value = 21.918338689427735, Temperature = 0.013713465517678262
Iteration 886: Best Value = 21.918338689427735, Temperature = 0.01357633086250148
Iteration 887: Best Value = 21.918338689427735, Temperature = 0.013440567553876464
Iteration 888: Best Value = 21.918338689427735, Temperature = 0.0133061618783377
Iteration 889: Best Value = 21.918338689427735, Temperature = 0.013173100259554323
Iteration 890: Best Value = 21.918338689427735, Temperature = 0.01304136925695878
Iteration 891: Best Value = 21.918338689427735, Temperature = 0.012910955564389192
Iteration 892: Best Value = 21.918338689427735, Temperature = 0.012781846008745299
Iteration 893: Best Value = 21.918338689427735, Temperature = 0.012654027548657847
Iteration 894: Best Value = 21.918338689427735, Temperature = 0.012527487273171267
Iteration 895: Best Value = 21.918338689427735, Temperature = 0.012402212400439554
Iteration 896: Best Value = 21.918338689427735, Temperature = 0.012278190276435159
Iteration 897: Best Value = 21.918338689427735, Temperature = 0.012155408373670807
Iteration 898: Best Value = 21.918338689427735, Temperature = 0.0120338542899341
Iteration 899: Best Value = 21.918338689427735, Temperature = 0.011913515747034758
Iteration 900: Best Value = 21.918338689427735, Temperature = 0.011794380589564411
Iteration 901: Best Value = 21.918338689427735, Temperature = 0.011676436783668767
Iteration 902: Best Value = 21.918338689427735, Temperature = 0.011559672415832079
Iteration 903: Best Value = 21.918338689427735, Temperature = 0.011444075691673758
Iteration 904: Best Value = 21.918338689427735, Temperature = 0.01132963493475702
Iteration 905: Best Value = 21.918338689427735, Temperature = 0.01121633858540945
Iteration 906: Best Value = 21.918338689427735, Temperature = 0.011104175199555354
Iteration 907: Best Value = 21.918338689427735, Temperature = 0.0109931334475598
```

```
Iteration 908: Best Value = 21.918338689427735, Temperature = 0.010883202113084202
Iteration 909: Best Value = 21.918338689427735, Temperature = 0.01077437009195336
Iteration 910: Best Value = 21.918338689427735, Temperature = 0.010666626391033827
Iteration 911: Best Value = 21.918338689427735, Temperature = 0.010559960127123489
Iteration 912: Best Value = 21.918338689427735, Temperature = 0.010454360525852253
Iteration 913: Best Value = 21.918338689427735, Temperature = 0.010349816920593731
Iteration 914: Best Value = 21.918338689427735, Temperature = 0.010246318751387793
Iteration 915: Best Value = 21.918338689427735, Temperature = 0.010143855563873915
Iteration 916: Best Value = 21.918338689427735, Temperature = 0.010042417008235176
Iteration 917: Best Value = 21.918338689427735, Temperature = 0.009941992838152823
Iteration 918: Best Value = 21.918338689427735, Temperature = 0.009842572909771296
Iteration 919: Best Value = 21.918338689427735, Temperature = 0.009744147180673582
Iteration 920: Best Value = 21.918338689427735, Temperature = 0.009646705708866846
Iteration 921: Best Value = 21.918338689427735, Temperature = 0.009550238651778177
Iteration 922: Best Value = 21.918338689427735, Temperature = 0.009454736265260395
Iteration 923: Best Value = 21.918338689427735, Temperature = 0.00936018890260779
Iteration 924: Best Value = 21.918338689427735, Temperature = 0.009266587013581714
Iteration 925: Best Value = 21.918338689427735, Temperature = 0.009173921143445896
Iteration 926: Best Value = 21.918338689427735, Temperature = 0.009082181932011437
Iteration 927: Best Value = 21.918338689427735, Temperature = 0.008991360112691322
Iteration 928: Best Value = 21.918338689427735, Temperature = 0.00890144651156441
Iteration 929: Best Value = 21.918338689427735, Temperature = 0.008812432046448765
Iteration 930: Best Value = 21.918338689427735, Temperature = 0.008724307725984277
Iteration 931: Best Value = 21.918338689427735, Temperature = 0.008637064648724433
Iteration 932: Best Value = 21.918338689427735, Temperature = 0.008550694002237189
Iteration 933: Best Value = 21.918338689427735, Temperature = 0.008465187062214817
Iteration 934: Best Value = 21.918338689427735, Temperature = 0.008380535191592669
Iteration 935: Best Value = 21.918338689427735, Temperature = 0.008296729839676742
Iteration 936: Best Value = 21.918338689427735, Temperature = 0.008213762541279975
Iteration 937: Best Value = 21.918338689427735, Temperature = 0.008131624915867176
Iteration 938: Best Value = 21.918338689427735, Temperature = 0.008050308666708503
Iteration 939: Best Value = 21.918338689427735, Temperature = 0.007969805580041418
Iteration 940: Best Value = 21.918338689427735, Temperature = 0.007890107524241003
Iteration 941: Best Value = 21.918338689427735, Temperature = 0.007811206448998593
Iteration 942: Best Value = 21.918338689427735, Temperature = 0.007733094384508607
Iteration 943: Best Value = 21.918338689427735, Temperature = 0.007655763440663521
Iteration 944: Best Value = 21.918338689427735, Temperature = 0.007579205806256885
Iteration 945: Best Value = 21.918338689427735, Temperature = 0.007503413748194317
Iteration 946: Best Value = 21.918338689427735, Temperature = 0.007428379610712374
Iteration 947: Best Value = 21.918338689427735, Temperature = 0.00735409581460525
Iteration 948: Best Value = 21.918338689427735, Temperature = 0.007280554856459198
Iteration 949: Best Value = 21.918338689427735, Temperature = 0.007207749307894606
Iteration 950: Best Value = 21.918338689427735, Temperature = 0.00713567181481566
```

```
Iteration 951: Best Value = 21.918338689427735, Temperature = 0.007064315096667504
Iteration 952: Best Value = 21.918338689427735, Temperature = 0.006993671945700828
Iteration 953: Best Value = 21.918338689427735, Temperature = 0.00692373522624382
Iteration 954: Best Value = 21.918338689427735, Temperature = 0.006854497873981381
Iteration 955: Best Value = 21.918338689427735, Temperature = 0.006785952895241567
Iteration 956: Best Value = 21.918338689427735, Temperature = 0.006718093366289151
Iteration 957: Best Value = 21.918338689427735, Temperature = 0.00665091243262626
Iteration 958: Best Value = 21.918338689427735, Temperature = 0.006584403308299997
Iteration 959: Best Value = 21.918338689427735, Temperature = 0.006518559275216997
Iteration 960: Best Value = 21.918338689427735, Temperature = 0.006453373682464827
Iteration 961: Best Value = 21.918338689427735, Temperature = 0.0063888399456401785
Iteration 962: Best Value = 21.918338689427735, Temperature = 0.006324951546183777
Iteration 963: Best Value = 21.918338689427735, Temperature = 0.006261702030721939
Iteration 964: Best Value = 21.918338689427735, Temperature = 0.0061990850104147194
Iteration 965: Best Value = 21.918338689427735, Temperature = 0.006137094160310572
Iteration 966: Best Value = 21.918338689427735, Temperature = 0.006075723218707466
Iteration 967: Best Value = 21.918338689427735, Temperature = 0.006014965986520391
Iteration 968: Best Value = 21.918338689427735, Temperature = 0.005954816326655187
Iteration 969: Best Value = 21.918338689427735, Temperature = 0.0058952681633886355
Iteration 970: Best Value = 21.918338689427735, Temperature = 0.005836315481754749
Iteration 971: Best Value = 21.918338689427735, Temperature = 0.0057779523269372015
Iteration 972: Best Value = 21.918338689427735, Temperature = 0.005720172803667829
Iteration 973: Best Value = 21.918338689427735, Temperature = 0.005662971075631151
Iteration 974: Best Value = 21.918338689427735, Temperature = 0.0056063413648748395
Iteration 975: Best Value = 21.918338689427735, Temperature = 0.005550277951226091
Iteration 976: Best Value = 21.918338689427735, Temperature = 0.00549477517171383
Iteration 977: Best Value = 21.918338689427735, Temperature = 0.0054398274199966914
Iteration 978: Best Value = 21.918338689427735, Temperature = 0.005385429145796724
Iteration 979: Best Value = 21.918338689427735, Temperature = 0.005331574854338757
Iteration 980: Best Value = 21.918338689427735, Temperature = 0.005278259105795369
Iteration 981: Best Value = 21.918338689427735, Temperature = 0.005225476514737416
Iteration 982: Best Value = 21.918338689427735, Temperature = 0.0051732217495900415
Iteration 983: Best Value = 21.918338689427735, Temperature = 0.005121489532094141
Iteration 984: Best Value = 21.918338689427735, Temperature = 0.0050702746367732
Iteration 985: Best Value = 21.918338689427735, Temperature = 0.005019571890405468
Iteration 986: Best Value = 21.918338689427735, Temperature = 0.004969376171501413
Iteration 987: Best Value = 21.918338689427735, Temperature = 0.004919682409786399
Iteration 988: Best Value = 21.918338689427735, Temperature = 0.004870485585688535
Iteration 989: Best Value = 21.918338689427735, Temperature = 0.0048217807298316494
Iteration 990: Best Value = 21.918338689427735, Temperature = 0.004773562922533333
Iteration 991: Best Value = 21.918338689427735, Temperature = 0.004725827293308
Iteration 992: Best Value = 21.918338689427735, Temperature = 0.004678569020374919
Iteration 993: Best Value = 21.918338689427735, Temperature = 0.00463178333017117
```

```
Iteration 994: Best Value = 21.918338689427735, Temperature = 0.004585465496869458
Iteration 995: Best Value = 21.918338689427735, Temperature = 0.004539610841900764
Iteration 996: Best Value = 21.918338689427735, Temperature = 0.004494214733481756
Iteration 997: Best Value = 21.918338689427735, Temperature = 0.004449272586146938
Iteration 998: Best Value = 21.918338689427735, Temperature = 0.004404779860285469
Iteration 999: Best Value = 21.918338689427735, Temperature = 0.004360732061682614
Iteration 1000: Best Value = 21.918338689427735, Temperature = 0.004317124741065788
Best Solution: [-0.9999999999999993, -1.3357370765021415e-16, -1.9799999999999993, 3.979999999999
Best Value (Minimum): 21.918338689427735
```

### 15.3.5. Particle swarm optimization (PSO)

Particle swarm optimization is a metaheuristic optimization algorithm inspired by the social behavior of birds flocking or fish schooling. It is used to solve optimization problems by iteratively improving a candidate solution based on the velocity and position of particles (potential solutions) in the search space. The PSO algorithm differs from other methods in a key way, that instead of updating a single candidate solution at each iteration, we update a population (set) of candidate solutions, called a swarm. Each candidate solution is the swarm is called a particle. We think of a swarm as an apparently disorganized population of moving individuals that tend to cluster together while each individual seems to be moving in a random direction. The POS algorithm aims to mimic the social behavior of animals and insects.

```julia
using Random, Plots

# Define the Rosenbrock function
function rosenbrock(x)
    return (1 - x[1])^2 + 100 * (x[2] - x[1]^2)^2
end

# PSO Implementation
function particle_swarm_optimization(objective, n_particles, n_iterations, bounds, dim)
    # Initialize particles
    positions = [rand(bounds[1]:0.1:bounds[2], dim) for _ in 1:n_particles]
    velocities = [rand(-1.0:0.1:1.0, dim) for _ in 1:n_particles]
    personal_best_positions = deepcopy(positions)
    personal_best_scores = [objective(p) for p in positions]
    global_best_position = personal_best_positions[argmin(personal_best_scores)]
    global_best_score = minimum(personal_best_scores)

    # PSO parameters
    ω = 0.5          # Inertia weight
    c1, c2 = 2.0, 2.0  # Cognitive and social learning factors
```

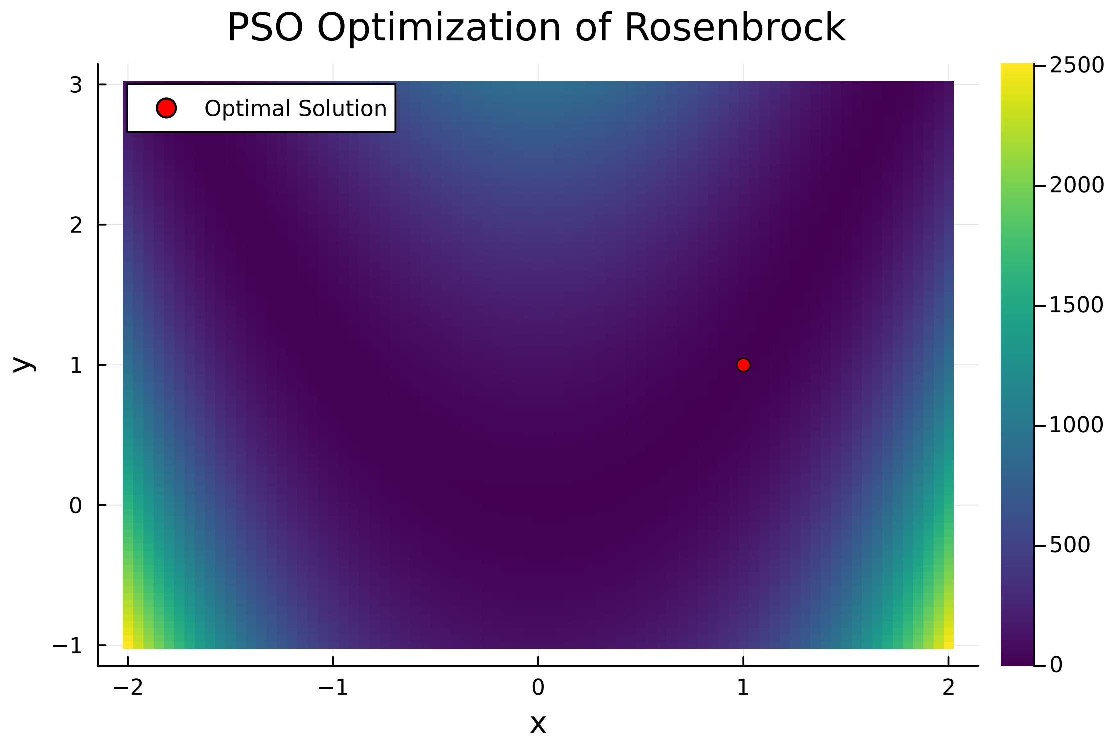
```
    # Optimization loop
    for iter in 1:n_iterations
        for i in 1:n_particles
            # Update velocity
            r1, r2 = rand(), rand()
            velocities[i] .= ω .* velocities[i] +
                             c1 * r1 .* (personal_best_positions[i] - positions[i]) +
                             c2 * r2 .* (global_best_position - positions[i])

            # Update position
            positions[i] .= positions[i] .+ velocities[i]

            # Clamp positions within bounds
            positions[i] .= clamp.(positions[i], bounds[1], bounds[2])

            # Evaluate fitness
            score = objective(positions[i])
            if score < personal_best_scores[i]
                personal_best_positions[i] = deepcopy(positions[i])
                personal_best_scores[i] = score
            end

            if score < global_best_score
                global_best_position = deepcopy(positions[i])
                global_best_score = score
            end
        end
        println("Iteration $iter: Best Score = $global_best_score")
    end
    return global_best_position, global_best_score
end

# Parameters
n_particles = 30
n_iterations = 100
bounds = (-2.0, 2.0)
dim = 2

# Run PSO
best_position, best_score = particle_swarm_optimization(rosenbrock, n_particles, n_iterations,

println("Best Position: $best_position")
```

```
println("Best Score: $best_score")

# Visualization
x = -2.0:0.05:2.0
y = -1.0:0.05:3.0
Z = [(1 - xi)^2 + 100 * (yi - xi^2)^2 for yi in y, xi in x]

heatmap(x, y, Z, color=:viridis, title="PSO Optimization of Rosenbrock", xlabel="x", ylabel="y
scatter!([best_position[1]], [best_position[2]], color=:red, label="Optimal Solution")
```

```
Iteration 1: Best Score = 0.7394683086878044
Iteration 2: Best Score = 0.7394683086878044
Iteration 3: Best Score = 0.1378423976864202
Iteration 4: Best Score = 0.1378423976864202
Iteration 5: Best Score = 0.1378423976864202
Iteration 6: Best Score = 0.09166036917907905
Iteration 7: Best Score = 0.08021195639862844
Iteration 8: Best Score = 0.07662425556347846
Iteration 9: Best Score = 0.02250234958291733
Iteration 10: Best Score = 0.005587481688806854
Iteration 11: Best Score = 0.002352801665689825
Iteration 12: Best Score = 0.000779154867043009
Iteration 13: Best Score = 0.00018872238297217115
Iteration 14: Best Score = 7.031062622903141e-5
Iteration 15: Best Score = 7.031062622903141e-5
Iteration 16: Best Score = 5.609383598784373e-5
Iteration 17: Best Score = 5.609383598784373e-5
Iteration 18: Best Score = 5.609383598784373e-5
Iteration 19: Best Score = 2.232173139229835 8e-5
Iteration 20: Best Score = 1.7046650715944367e-6
Iteration 21: Best Score = 1.7046650715944367e-6
Iteration 22: Best Score = 4.517252548493867e-7
Iteration 23: Best Score = 4.517252548493867e-7
Iteration 24: Best Score = 4.517252548493867e-7
Iteration 25: Best Score = 4.517252548493867e-7
Iteration 26: Best Score = 4.517252548493867e-7
Iteration 27: Best Score = 4.517252548493867e-7
Iteration 28: Best Score = 4.517252548493867e-7
Iteration 29: Best Score = 4.517252548493867e-7
Iteration 30: Best Score = 4.517252548493867e-7
Iteration 31: Best Score = 4.517252548493867e-7
Iteration 32: Best Score = 4.517252548493867e-7
Iteration 33: Best Score = 4.474697961383702e-7
```

```
Iteration 34: Best Score = 4.474697961383702e-7
Iteration 35: Best Score = 8.63863908710677e-8
Iteration 36: Best Score = 1.0310535389272776e-8
Iteration 37: Best Score = 1.0310535389272776e-8
Iteration 38: Best Score = 1.0310535389272776e-8
Iteration 39: Best Score = 1.0310535389272776e-8
Iteration 40: Best Score = 1.0201524688291877e-8
Iteration 41: Best Score = 4.4342436808044104e-9
Iteration 42: Best Score = 3.2881305649274648e-9
Iteration 43: Best Score = 3.1494783318508666e-9
Iteration 44: Best Score = 3.1494783318508666e-9
Iteration 45: Best Score = 5.94751436923238e-10
Iteration 46: Best Score = 5.94751436923238e-10
Iteration 47: Best Score = 4.197634887617902e-10
Iteration 48: Best Score = 1.7954898863944538e-10
Iteration 49: Best Score = 1.7954898863944538e-10
Iteration 50: Best Score = 1.7954898863944538e-10
Iteration 51: Best Score = 1.7954898863944538e-10
Iteration 52: Best Score = 1.7954898863944538e-10
Iteration 53: Best Score = 1.7954898863944538e-10
Iteration 54: Best Score = 1.4158723346678016e-11
Iteration 55: Best Score = 1.4158723346678016e-11
Iteration 56: Best Score = 1.4158723346678016e-11
Iteration 57: Best Score = 1.4158723346678016e-11
Iteration 58: Best Score = 1.4158723346678016e-11
Iteration 59: Best Score = 1.4158723346678016e-11
Iteration 60: Best Score = 1.0038936285251106e-11
Iteration 61: Best Score = 3.5656547408673725e-12
Iteration 62: Best Score = 3.5656547408673725e-12
Iteration 63: Best Score = 3.5656547408673725e-12
Iteration 64: Best Score = 3.5656547408673725e-12
Iteration 65: Best Score = 3.5656547408673725e-12
Iteration 66: Best Score = 3.5656547408673725e-12
Iteration 67: Best Score = 1.722970522635985e-14
Iteration 68: Best Score = 1.722970522635985e-14
Iteration 69: Best Score = 1.722970522635985e-14
Iteration 70: Best Score = 1.722970522635985e-14
Iteration 71: Best Score = 1.722970522635985e-14
Iteration 72: Best Score = 1.722970522635985e-14
Iteration 73: Best Score = 2.2873033147880103e-16
Iteration 74: Best Score = 2.2873033147880103e-16
Iteration 75: Best Score = 2.2873033147880103e-16
Iteration 76: Best Score = 2.2873033147880103e-16
```

```
Iteration 77: Best Score = 2.2873033147880103e-16
Iteration 78: Best Score = 2.2873033147880103e-16
Iteration 79: Best Score = 2.2873033147880103e-16
Iteration 80: Best Score = 2.2873033147880103e-16
Iteration 81: Best Score = 2.24773684050263e-16
Iteration 82: Best Score = 2.24773684050263e-16
Iteration 83: Best Score = 2.24773684050263e-16
Iteration 84: Best Score = 2.24773684050263e-16
Iteration 85: Best Score = 1.8892446512253845e-16
Iteration 86: Best Score = 1.8892446512253845e-16
Iteration 87: Best Score = 1.8892446512253845e-16
Iteration 88: Best Score = 1.8892446512253845e-16
Iteration 89: Best Score = 4.284696288934294e-17
Iteration 90: Best Score = 4.284696288934294e-17
Iteration 91: Best Score = 1.745593904469215e-17
Iteration 92: Best Score = 1.745593904469215e-17
Iteration 93: Best Score = 1.745593904469215e-17
Iteration 94: Best Score = 1.745593904469215e-17
Iteration 95: Best Score = 1.745593904469215e-17
Iteration 96: Best Score = 1.745593904469215e-17
Iteration 97: Best Score = 5.829480589621494e-19
Iteration 98: Best Score = 5.829480589621494e-19
Iteration 99: Best Score = 5.829480589621494e-19
Iteration 100: Best Score = 5.829480589621494e-19
Best Position: [1.0000000001510196, 1.0000000003768819]
Best Score: 5.829480589621494e-19
```




### 15.3.6. Evolutionary algorithm

An evolutionary algorithm (EA) is a family of optimization algorithms inspired by the principles of biological evolution. They are particularly useful for solving complex optimization problems where traditional gradient-based methods may struggle due to non-linearity, multimodality, or high dimensionality of the search space.

The following shows an example to maximize population fitness in terms of an objective function, with common crossover and mutation processes throughout all generations.

```
using Random

Random.seed!(1234)

# Parameters
population_size = 50          # Number of individuals in the population
chromosome_length = 5          # Number of genes in each individual (dimensionality)
generations = 100              # Number of generations
mutation_rate = 0.1            # Probability of mutation
crossover_rate = 0.7           # Probability of crossover
bounds = (-5.12, 5.12)         # Boundaries for each gene
```

```julia
# Target function: Rastrigin function (a common benchmark for optimization)
function rastrigin(x)
    A = 10
    n = length(x)
    return A * n + sum(xi^2 - A * cos(2 * π * xi) for xi in x)
end

# Initialize population randomly within bounds
function initialize_population()
    return [rand(bounds[1]:0.01:bounds[2], chromosome_length) for _ in 1:population_size]
end

# Fitness function (negative because we are minimizing)
function fitness(individual)
    return -rastrigin(individual)
end

# Selection: Tournament selection
function tournament_selection(population, fitnesses)
    candidates = rand(1:population_size, 2)
    return ifelse(fitnesses[candidates[1]] > fitnesses[candidates[2]],
                  population[candidates[1]], population[candidates[2]])
end

# Crossover: Single-point crossover
function crossover(parent1, parent2)
    if rand() < crossover_rate
        point = rand(1:chromosome_length)
        child1 = vcat(parent1[1:point], parent2[point+1:end])
        child2 = vcat(parent2[1:point], parent1[point+1:end])
        return child1, child2
    else
        return parent1, parent2
    end
end

# Mutation: Randomly change genes with some probability
function mutate(individual)
    for i in 1:chromosome_length
        if rand() < mutation_rate
            individual[i] = rand(bounds[1]:0.01:bounds[2])
        end
    end
```

```
    return individual
end

# Main Genetic Algorithm loop
function genetic_algorithm()
    population = initialize_population()
    best_individual = nothing
    best_fitness = -Inf

    for gen in 1:generations
        # Evaluate fitness
        fitnesses = [fitness(ind) for ind in population]

        # Find best individual in current population
        current_best = argmax(fitnesses)
        if fitnesses[current_best] > best_fitness
            best_fitness = fitnesses[current_best]
            best_individual = population[current_best]
        end

        # Generate new population
        new_population = []
        while length(new_population) < population_size
            # Selection
            parent1 = tournament_selection(population, fitnesses)
            parent2 = tournament_selection(population, fitnesses)

            # Crossover
            child1, child2 = crossover(parent1, parent2)

            # Mutation
            child1 = mutate(child1)
            child2 = mutate(child2)

            # Add children to new population
            push!(new_population, child1, child2)
        end
        population = new_population[1:population_size]

        println("Generation $gen: Best Fitness = ", best_fitness)
    end

    return best_individual, -best_fitness
```

```
end

# Run the genetic algorithm
best_solution, best_value = genetic_algorithm()
println("Best Solution: ", best_solution)
println("Best Value (Minimum): ", best_value)
```

```
Generation 1: Best Fitness = -45.64660837302044
Generation 2: Best Fitness = -31.378046131174845
Generation 3: Best Fitness = -19.662385579175435
Generation 4: Best Fitness = -19.662385579175435
Generation 5: Best Fitness = -19.662385579175435
Generation 6: Best Fitness = -19.662385579175435
Generation 7: Best Fitness = -8.288027804642205
Generation 8: Best Fitness = -5.648027804642204
Generation 9: Best Fitness = -5.648027804642204
Generation 10: Best Fitness = -5.648027804642204
Generation 11: Best Fitness = -5.648027804642204
Generation 12: Best Fitness = -5.648027804642204
Generation 13: Best Fitness = -5.648027804642204
Generation 14: Best Fitness = -5.648027804642204
Generation 15: Best Fitness = -5.648027804642204
Generation 16: Best Fitness = -5.648027804642204
Generation 17: Best Fitness = -5.648027804642204
Generation 18: Best Fitness = -5.648027804642204
Generation 19: Best Fitness = -5.648027804642204
Generation 20: Best Fitness = -5.648027804642204
Generation 21: Best Fitness = -5.648027804642204
Generation 22: Best Fitness = -5.648027804642204
Generation 23: Best Fitness = -5.648027804642204
Generation 24: Best Fitness = -5.648027804642204
Generation 25: Best Fitness = -5.648027804642204
Generation 26: Best Fitness = -5.648027804642204
Generation 27: Best Fitness = -5.648027804642204
Generation 28: Best Fitness = -5.648027804642204
Generation 29: Best Fitness = -5.648027804642204
Generation 30: Best Fitness = -5.648027804642204
Generation 31: Best Fitness = -5.648027804642204
Generation 32: Best Fitness = -5.648027804642204
Generation 33: Best Fitness = -5.648027804642204
Generation 34: Best Fitness = -4.7113788979761395
Generation 35: Best Fitness = -4.7113788979761395
Generation 36: Best Fitness = -4.7113788979761395
```

```
Generation 37: Best Fitness = -4.7113788979761395
Generation 38: Best Fitness = -4.7113788979761395
Generation 39: Best Fitness = -3.4936380019755617
Generation 40: Best Fitness = -3.4936380019755617
Generation 41: Best Fitness = -3.4936380019755617
Generation 42: Best Fitness = -3.4936380019755617
Generation 43: Best Fitness = -3.4936380019755617
Generation 44: Best Fitness = -3.4936380019755617
Generation 45: Best Fitness = -3.4936380019755617
Generation 46: Best Fitness = -3.4936380019755617
Generation 47: Best Fitness = -3.4936380019755617
Generation 48: Best Fitness = -3.4936380019755617
Generation 49: Best Fitness = -3.4936380019755617
Generation 50: Best Fitness = -3.4936380019755617
Generation 51: Best Fitness = -3.4936380019755617
Generation 52: Best Fitness = -3.4936380019755617
Generation 53: Best Fitness = -3.4936380019755617
Generation 54: Best Fitness = -3.4936380019755617
Generation 55: Best Fitness = -3.4936380019755617
Generation 56: Best Fitness = -3.4936380019755617
Generation 57: Best Fitness = -3.4936380019755617
Generation 58: Best Fitness = -3.4936380019755617
Generation 59: Best Fitness = -3.4936380019755617
Generation 60: Best Fitness = -3.4936380019755617
Generation 61: Best Fitness = -3.4936380019755617
Generation 62: Best Fitness = -3.4936380019755617
Generation 63: Best Fitness = -3.4936380019755617
Generation 64: Best Fitness = -3.4936380019755617
Generation 65: Best Fitness = -3.4936380019755617
Generation 66: Best Fitness = -3.4936380019755617
Generation 67: Best Fitness = -3.4936380019755617
Generation 68: Best Fitness = -3.4936380019755617
Generation 69: Best Fitness = -3.4936380019755617
Generation 70: Best Fitness = -3.4936380019755617
Generation 71: Best Fitness = -3.4936380019755617
Generation 72: Best Fitness = -3.4936380019755617
Generation 73: Best Fitness = -3.4936380019755617
Generation 74: Best Fitness = -3.4936380019755617
Generation 75: Best Fitness = -3.4936380019755617
Generation 76: Best Fitness = -3.4936380019755617
Generation 77: Best Fitness = -3.4936380019755617
Generation 78: Best Fitness = -3.4936380019755617
Generation 79: Best Fitness = -3.4936380019755617
```

```
Generation 80: Best Fitness = -3.4936380019755617
Generation 81: Best Fitness = -3.4936380019755617
Generation 82: Best Fitness = -3.4936380019755617
Generation 83: Best Fitness = -3.4936380019755617
Generation 84: Best Fitness = -3.4936380019755617
Generation 85: Best Fitness = -3.4936380019755617
Generation 86: Best Fitness = -3.4936380019755617
Generation 87: Best Fitness = -3.4936380019755617
Generation 88: Best Fitness = -3.4936380019755617
Generation 89: Best Fitness = -3.4936380019755617
Generation 90: Best Fitness = -2.9892957379930465
Generation 91: Best Fitness = -2.9892957379930465
Generation 92: Best Fitness = -2.9892957379930465
Generation 93: Best Fitness = -2.9892957379930465
Generation 94: Best Fitness = -2.9892957379930465
Generation 95: Best Fitness = -2.9892957379930465
Generation 96: Best Fitness = -2.9892957379930465
Generation 97: Best Fitness = -2.9892957379930465
Generation 98: Best Fitness = -2.9892957379930465
Generation 99: Best Fitness = -2.9892957379930465
Generation 100: Best Fitness = -2.9892957379930465
Best Solution: [-0.02, -1.02, -0.04, 0.02, -1.04]
Best Value (Minimum): 2.9892957379930465
```

### 15.3.7. Bayesian optimization

Bayesian Optimization (BO) is a powerful technique for global optimization of expensive-to-evaluate black-box functions. It leverages probabilistic models to predict the objective function's behavior across the search space and uses these models to make informed decisions about where to evaluate the function next. This approach efficiently balances exploration (searching for promising regions) and exploitation (exploiting regions likely to yield optimal values), making it particularly suitable for optimization problems where function evaluations are costly, such as tuning hyperparameters of machine learning models or optimizing parameters of complex simulations.

```julia
using Random

# Define your objective function to be optimized
function objective(x::Float64)
    return -(x^2 + 0.1 * sin(5 * x))  # Example objective function (negative because we seek m
end
# Bayesian optimization function
function bayesian_optimization(objective, bounds::Tuple{Float64,Float64}, num_iterations::Int)
```

```
    Random.seed!(1234)  # Setting a seed for reproducibility
    X = Float64[]  # List to store evaluated points
    Y = Float64[]  # List to store objective values
    # Initial random point (you can choose other initial points as well)
    x_init = rand() * (bounds[2] - bounds[1]) + bounds[1]
    push!(X, x_init)
    push!(Y, objective(x_init))
    # Main loop
    for i in 1:num_iterations
        # Fit a model to the observed data (Gaussian Process in this case)
        # For simplicity, let's just use the current best observed value
        x_next = rand() * (bounds[2] - bounds[1]) + bounds[1]  # Random sampling
        # Evaluate the objective function at the chosen point
        y_next = objective(x_next)
        # Update the data with the new observation
        push!(X, x_next)
        push!(Y, y_next)
        # Here, we will just print the current best observed value
        println("Iteration $i: Best value = $(maximum(Y))")
    end
    # Return the best observed value and corresponding parameter
    best_idx = argmax(Y)
    return X[best_idx], Y[best_idx]
end

best_x, best_value = bayesian_optimization(objective, (-5.0, 5.0), 10)
println("Best x found: $best_x, Best value: $best_value")
```

```
Iteration 1: Best value = -0.3041807254074535
Iteration 2: Best value = -0.3041807254074535
Iteration 3: Best value = -0.3041807254074535
Iteration 4: Best value = -0.3041807254074535
Iteration 5: Best value = -0.3041807254074535
Iteration 6: Best value = -0.3041807254074535
Iteration 7: Best value = -0.3041807254074535
Iteration 8: Best value = 0.02504980758369785
Iteration 9: Best value = 0.02504980758369785
Iteration 10: Best value = 0.02504980758369785
Best x found: -0.057501331095793695, Best value: 0.02504980758369785
```

### 15.3.8. BFGS

The Broyden-Fletcher-Goldfarb-Shanno (BFGS) method is a popular iterative optimization algorithm used for unconstrained optimization problems. It belongs to the family of quasi-Newton methods, which are designed to find the local minimum of a differentiable objective function without needing its Hessian matrix directly. Instead, BFGS iteratively constructs an approximation to the inverse Hessian matrix using gradients of the objective function.

The following example uses the `Optim` package available in Julia to do the BFGS optimization.

```
using Optim

# Define the objective function to minimize
function objective_function(x)
    return sum(x .* x)
end

# Initial guess for the minimization
initial_x = [1.0]
# Perform optimization using BFGS method
result = optimize(objective_function, initial_x, BFGS())
# Extract the optimized solution
solution = result.minimizer
minimum_value = result.minimum

# Print the result
println("Optimized solution: x = ", solution)
println("Minimum value found: ", minimum_value)
```

```
Optimized solution: x = [-6.359357485052897e-13]
Minimum value found: 4.0441427622698303e-25
```

## 15.4. Model fitting

### 15.4.1. Root finding

Root finding, also known as root approximation or root isolation, is the process of finding the values of the independent variable (usually denoted as $x$) for which a given function equals zero. In mathematical terms, if we have a function $f(x)$, root finding involves finding values of $x$ such that $f(x) = 0$.

There are various algorithms for root finding, each with its own advantages and disadvantages depending on the characteristics of the function and the requirements of the problem. One notable approach is Newton's method, an iterative method that uses the derivative of the function to approximate the root with increasing accuracy in each iteration.

We will again use a simple function to illustrate the process:

```
using Zygote

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

```
Warning: Maximum number of iterations reached.
Approximate root: -1.391591884376212
```

### 15.4.2. Bracketed search algorithm

A bracketed search algorithm is a technique used in optimization and numerical methods to confine or "bracket" a minimum or maximum of a function within a specified interval. The primary goal is to reduce the search space systematically until a satisfactory solution or range containing the optimal value is found.

```
function bisection_method(f, a, b; tol=1e-6, max_iter=100)
    """
    Bisection method to find a root of the function f(x) within the interval [a, b].
```

```
    Parameters:
    - f: Function to find the root of.
    - a, b: Initial interval [a, b] where the root is expected to be.
    - tol: Tolerance for the root (default is 1e-6).
    - max_iter: Maximum number of iterations allowed (default is 100).

    Returns:
    - root: Approximate root found within the tolerance.
    - iterations: Number of iterations taken to converge.
    """
    fa = f(a)
    fb = f(b)
    if fa * fb > 0
        error("The function values at the endpoints must have opposite signs.")
    end
    iterations = 0
    while (b - a) / 2 > tol && iterations < max_iter
        c = (a + b) / 2
        fc = f(c)
        if fc == 0
            return c, iterations
        end
        if fa * fc < 0
            b = c
            fb = fc
        else
            a = c
            fa = fc
        end
        iterations += 1
    end
    root = (a + b) / 2
    return root, iterations
end

# Define the function we want to find the root of
function f(x)
    return x^3 - 6x^2 + 11x - 6.1
end

# Initial interval [a, b] and tolerance
a = 0.5
```

```
b = 10
tolerance = 1e-6
# Apply the bisection method
root, iterations = bisection_method(f, a, b, tol=tolerance)

# Print results
println("Approximate root: ", root)
println("Iterations taken: ", iterations)
println("Function value at root: ", f(root))
```

```
Approximate root: 3.046680122613907
Iterations taken: 23
Function value at root: -9.356632642010254e-7
```

### 15.4.3. Best fitting curve

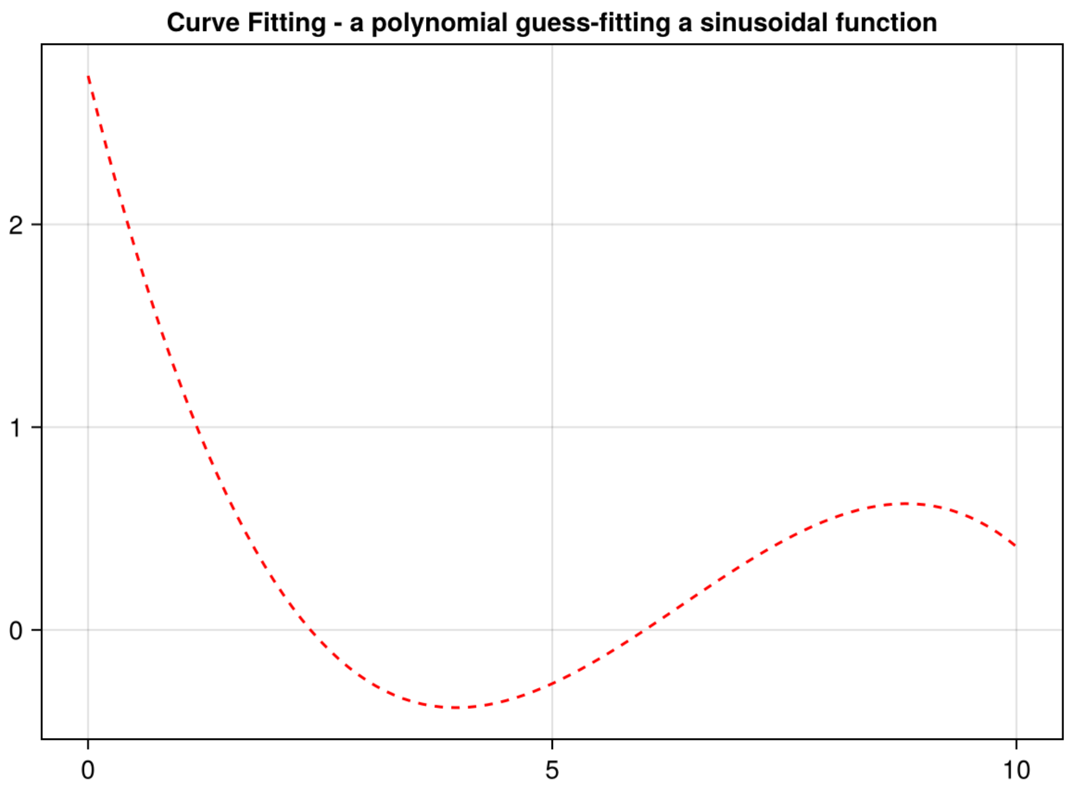
In model fitting, the “best fitting curve” refers to the curve or function that best describes the relationship between the independent and dependent variables in the data. The goal of model fitting is to find the parameters of the chosen curve or function that minimize the difference between the observed data points and the values predicted by the model.

The process of finding the best fitting curve typically involves:

- Choosing a model: Based on the nature of the data and the underlying relationship between the variables, a suitable model or family of models are selected.
- Estimating parameters: Using the chosen model, one estimates the parameters that best describe the relationship between the variables. This is often done using optimization techniques such as least squares regression, maximum likelihood estimation, or Bayesian inference.
- Evaluating the fit: Once the parameters are estimated, one evaluates the goodness of fit of the model by comparing the predicted values to the observed data. Common metrics for evaluating fit, or error functions, include the residual sum of squares, the coefficient of determination (R-squared), and visual inspection of the residuals.
- Iterating if necessary: If the fit is not satisfactory, one may need to iterate on the model or consider alternative models until you find a satisfactory fit to the data.

```
using LsqFit, CairoMakie

x_data = 0:0.1:10
y_data = 2 .* sin.(x_data) .+ 0.5 .* randn(length(x_data))
```

```
# Define the model function
curve_model(x, p) = p[1] * x .^ 3 + p[2] * x .^ 2 .+ p[3] * x .+ p[4]
# Initial parameter guess
p₀ = [1.0, 1.0, 1.0, 1.0]
# Fit the model to the data
fit_result = curve_fit(curve_model, x_data, y_data, p₀)
# Extract the fitted parameters
params = coef(fit_result)
# Evaluate the model with the fitted parameters
y_fit = curve_model(x_data, params)
# Plot the data and the fitted curve
fig = Figure()
Axis(fig[1, 1], title="Curve Fitting - a polynomial guess-fitting a sinusoidal function")
scatter!(x_data, y_data, label="Data")
lines!(x_data, y_fit, label="Fitted Curve", linestyle=:dash, color=:red)
fig
```



# 16. 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[1] (c. 1960)

Stochastic modeling is a mathematical approach used to model systems or processes that exhibit randomness or uncertainty. It is widely applied in fields like finance, engineering, biology, and operations research. We will start by taking a look at key elements of stochastic modeling.

## 16.1. Key elements

### 16.1.1. Random variables

A random variable represents a quantity whose value is determined by the outcome of a random event. It can be discrete or continuous. It is essentially a mapping from event space to numerical values. Examples include stock prices, waiting time in queues and number of claims in insurance, etc. Random variables form the basis of stochastic models by introducing uncertainty into the model.

### 16.1.2. Probability distributions

A probability distribution describes the likelihood of different outcomes for a random variable. Examples include normal distribution, Poisson distribution and exponential distribution, etc. Probability distributions help in modeling the behavior of random variables and in defining how likely different events or outcomes are.

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

### 16.1.3. State space

The state space represents all possible states or values that a stochastic process can take. For example, in a Markov process, the state space might be the set of all possible values that the system can occupy at any given time. The state space helps in defining the scope of the model by specifying possible outcomes.

### 16.1.4. Stochastic processes

A stochastic process is a collection of random variables indexed by time (or some other variable) that evolve in a probabilistic manner. Examples include Brownian motion, Markov chains and Poisson processes. Stochastic processes model how random variables change over time, which is essential for understanding dynamic systems influenced by randomness.

### 16.1.5. Transition probabilities

These represent the probabilities of transitioning from one state to another in a stochastic process. For example, in a Markov chain, the transition matrix contains the probabilities of moving from one state to another. Transition probabilities determine how the system evolves from one time step to the next, reflecting the underlying randomness.

### 16.1.6. Time horizon

The time horizon refers to the period over which the stochastic process is observed. It can be discrete (e.g., steps in a Markov chain) or continuous (e.g., continuous-time models like Brownian motion). The time horizon helps in determining how the process behaves over short or long periods.

### 16.1.7. Initial conditions

These are the starting points or initial values of the random variables or the system at time zero. They may be initial price of a stock, initial number of customers in a queue, etc. The starting condition influences the future evolution of the process, and different initial conditions can lead to different outcomes.

### 16.1.8. Expectation and variance

The expected value (mean) represents the average or mean outcome of a random variable over many trials or realizations. The variance measures the spread or variability of outcomes around the expected value. These statistical measures provide insights into the central tendency and the uncertainty or risk in a stochastic model.

### 16.1.9. Covariance and correlation

Covariance measures how two random variables change together. Positive covariance indicates that the variables tend to increase together. On the other hand correlation is the standardized version of covariance that measures the strength of the linear relationship between two variables. Understanding how different random variables interact helps in building more complex models, especially in multivariate stochastic processes.

### 16.1.10. Stationarity

A stochastic process is stationary if its statistical properties, such as mean and variance, do not change over time. Strict stationarity means all moments of the process are invariant over time, while weak Stationarity means only the mean and variance are invariant over time. Stationarity simplifies the analysis and modeling of stochastic processes, especially for time series data.

### 16.1.11. Martingales

A martingale is a stochastic process where the expected future value, given all prior information, is equal to the current value. Examples include fair gambling games or financial asset prices in efficient markets. Martingales are important in financial modeling and risk-neutral pricing, especially for derivatives.

### 16.1.12. Noise (random shocks)

Random noise represents unpredictable random fluctuations that can affect the outcome of a stochastic process. This can be market volatility, measurement errors or environmental variations. Noise is a critical element in stochastic models as it introduces randomness and uncertainty into otherwise deterministic systems.

### 16.1.13. Markov property

A stochastic process has the Markov property if the future state depends only on the current state and not on the past states (memorylessness). Markov chains and hidden Markov models contain this property. The Markov property simplifies modeling and computation by reducing dependencies on past states.

### 16.1.14. Brownian motion (Wiener process)

A continuous-time stochastic process where changes over time are independent and normally distributed. This is extensively used in financial models for asset price movements (e.g., the Black-Scholes model). In addition to financial markets, Brownian motion is also widely used in modeling physical systems (diffusion).

### 16.1.15. Stochastic differential equations (SDEs)

These are differential equations that incorporate stochastic terms (typically driven by Brownian motion or other noise sources). The well-known Black-Scholes equation for option pricing is one example of how SDE can be applied in real world. SDEs are essential for modeling systems where both deterministic and random factors drive behavior over time.

## 16.2. Applications

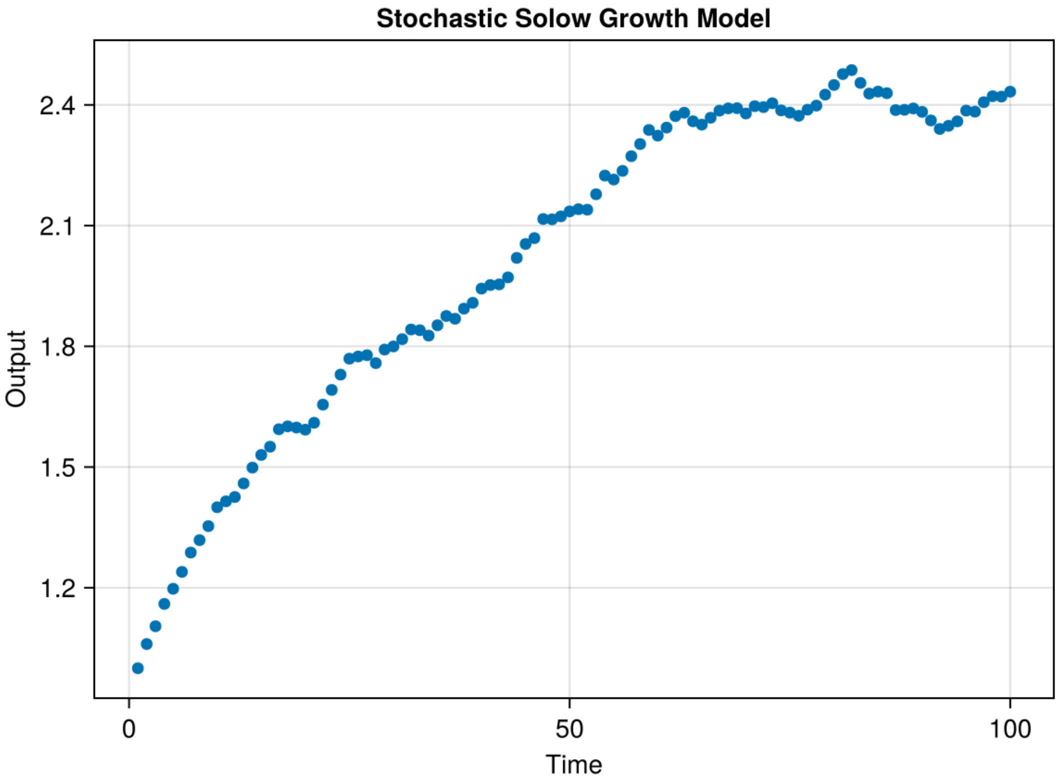
Due to technology advancements in recent years, stochastic analyses can be found in a plethora of fields. Refer to **?@sec-portop** for portfolio optimization applications. Refer to **?@sec-optimization** for the usage of gene algorithms for certain optimization problems.

Here we show still another stochastic process in macroeconomic analysis. Stochastic macroeconomic analysis often involves modeling random shocks and their effects on macroeconomic variables such as output, consumption, inflation, and employment. One common approach is through Dynamic Stochastic General Equilibrium (DSGE) models, which are widely used in macroeconomic analysis. These models incorporate randomness (stochastic elements) to capture real-world uncertainty in economic systems.

```
using Random, CairoMakie, Distributions

# Parameters
α = 0.33                # Capital share of output
```

```
δ = 0.05                # Depreciation rate
s = 0.2                 # Savings rate
n = 0.01                # Population growth rate
g = 0.02                # Technology growth rate
σ = 0.01                # Standard deviation of productivity shocks
T = 100                 # Number of periods to simulate
K0 = 1.0                # Initial capital stock
A0 = 1.0                # Initial productivity

# Shock distribution (normal distribution for productivity shocks)
shock_distribution = Normal(0, σ)

# Function to simulate the model
function simulate_stochastic_solow(T, α, δ, s, n, g, σ, K0, A0)
    K = zeros(T)        # Capital over time
    Y = zeros(T)        # Output over time
    A = zeros(T)        # Productivity shocks over time
    A[1] = A0           # Initial productivity
    K[1] = K0           # Initial capital

    for t in 1:T-1
        # Apply random productivity shock
        ε_t = rand(shock_distribution)
        A[t+1] = A[t] * exp(ε_t)  # Productivity evolves stochastically

        # Output based on Cobb-Douglas production function
        Y[t] = A[t] * K[t]^α

        # Capital accumulation equation
        K[t+1] = s * Y[t] + (1 - δ) * K[t]

        # Population and technology growth
        K[t+1] *= (1 + n) * (1 + g)
    end

    Y[T] = A[T] * K[T]^α  # Final output
    return K, Y, A
end

# Simulate the model
K, Y, A = simulate_stochastic_solow(T, α, δ, s, n, g, σ, K0, A0)

# Plot the results
```

```
f = Figure()
axis = Axis(f[1, 1], xlabel="Time", ylabel="Output", title="Stochastic Solow Growth Model")
scatter!(1:T, Y, label="Output (Y)")
f
```




## 16.3. Conclusion

Stochastic modeling involves the study of systems influenced by random factors and uncertainty. By combining random variables, probability distributions, and processes like Markov chains or Brownian motion, stochastic models provide insights into systems that cannot be described purely deterministically.

# 17. Visualizations

## 17.1. 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.

## 17.2. Key elements

Visualization is a crucial component in data analysis, statistical modeling, and decision-making for several important reasons. It serves as a bridge between raw data and actionable insights, helping to simplify, explore, and communicate complex information effectively. Here are key reasons why visualization is important:

### 17.2.1. Understanding and exploring data

Depending on purposes, different types of plots can help understand underlying relationships in the data. A scatter plot can reveal correlations between two variables, while a histogram can show the distribution of a single variable. - Pattern Recognition: Visualization helps in identifying trends, patterns, and outliers in data that might not be immediately evident through raw numbers. - Exploratory Data Analysis (EDA): It allows analysts to explore the data interactively, enabling a better understanding of underlying distributions, relationships, and anomalies before applying formal statistical models.

### 17.2.2. Simplifying complex information

Large datasets with many variables or dimensions can be overwhelming. Visualization simplifies complex data by representing it in intuitive visual formats like graphs, charts, and heatmaps. Besides, instead of dealing with thousands of rows of data, a well-designed graph can distill essential information for quick understanding. For example, in multivariate datasets, a principal component analysis (PCA) biplot reduces the dimensions of the data, making it easier to interpret.

### 17.2.3. Identifying relationships and correlations

Relationships between variables, such as correlation, dependency, or causality, can be better understood through visualizations like scatter plots, line graphs, and network diagrams. For example, a heatmap can show the strength of correlations between multiple variables, helping in the selection of important features for modeling.

### 17.2.4. Supporting decision making

Visualization aids decision-makers by presenting data in a clear and actionable form, often enabling quicker and more informed decisions. By visualizing risks, such as potential financial losses, decision-makers can understand the range of possible outcomes and make better choices. For example, visualizing portfolio risk through VaR (Value at Risk) or loss distributions helps in assessing potential future losses.

### 17.2.5. Effective communication

A good visualization is a powerful tool for communicating insights to stakeholders, especially non-technical audiences, in a way that is easy to grasp. It can also be used to tell a compelling story by emphasizing important findings and making the data more engaging. A time-series line chart can tell the story of stock price movements over time, showing trends and volatility to investors.

### 17.2.6. Spotting anomalies and outliers

Anomalies, such as errors or unexpected behaviors in data, are easier to spot with visual representations like box plots, scatter plots, or time series graphs. This can be applied in fraud detection, where visualizing transactions over time can reveal unusual spikes or patterns that may indicate fraudulent activity.

### 17.2.7. Interactive exploration

Interactive visualizations, like those created using tools such as Tableau or Plotly in Python/Julia, allow users to zoom in on areas of interest, filter data, and dynamically explore different views of the data. This enables users to adjust parameters, select specific variables, or focus on different time frames, leading to more personalized insights. Interactive dashboards let business analysts explore sales trends by geography, time, or product category in real-time.

### 17.2.8. Enhancing models and hypotheses

Visualizing the results of statistical models (such as regression, classification, or clustering) allows for easier validation and refinement of models. It can also be used to compare model predictions with actual outcomes, identifying where the model performs well or poorly. For example a residual plot can help diagnose model fit in a regression analysis, highlighting whether the errors are randomly distributed or exhibit patterns.

### 17.2.9. Handling large datasets

Visualization can handle large datasets by summarizing the data in visual formats, providing a way to understand vast amounts of information quickly. This can be applied in machine learning, where feature importance can be visualized across many features, helping data scientists to focus on the most critical variables for prediction.

### 17.2.10. Revealing hidden structures in data

Techniques like clustering and dimensionality reduction can reveal hidden structures in data, such as natural groupings or latent variables, which become apparent only when visualized. For one application in unsupervised learning, visualizing clusters from a k-means algorithm can show the natural grouping of customers in a marketing dataset.

### 17.2.11. Fostering collaboration

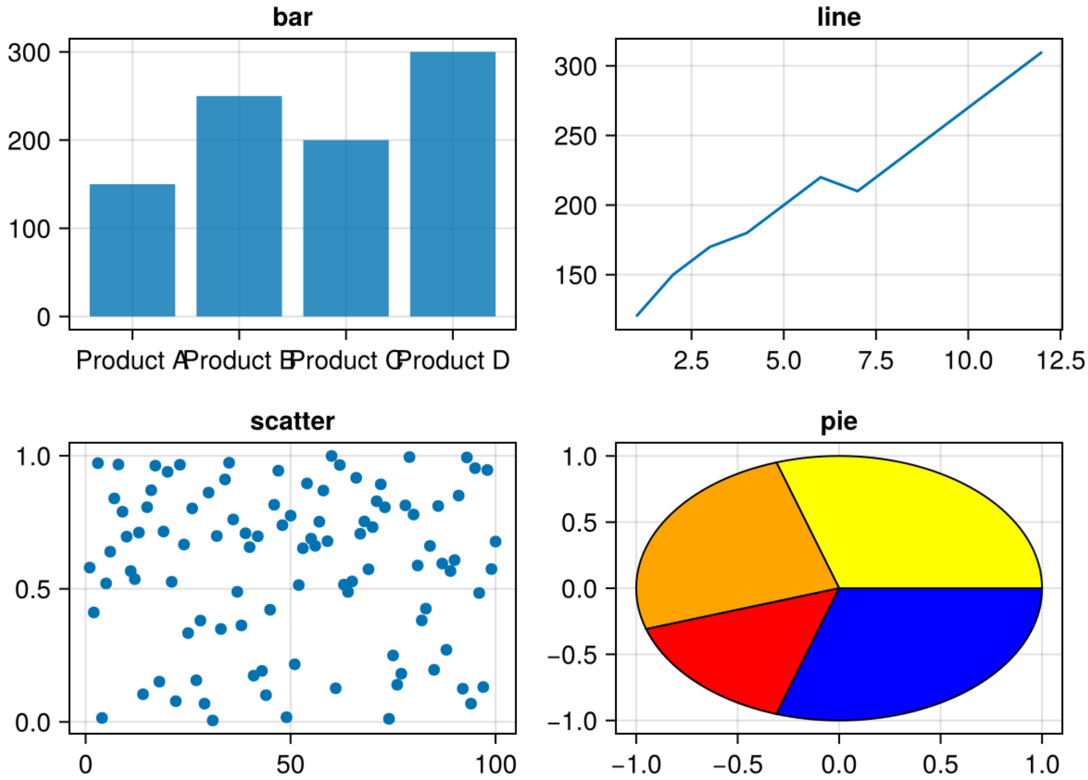
Visualization helps to bridge communication gaps between different teams (technical, business, management), enabling better collaboration. A dashboard summarizing KPIs (key performance indicators) for different departments can help teams across the organization align with the same goals.

## 17.3. Types of visualization tools

- Basic Charts and Graphs: Bar charts, line graphs, scatter plots, histograms, pie charts.
  - Bar charts are best for comparing categorical data or discrete values across different categories. Sometimes categories can be grouped for a stacked bar chart to show for example how each category changes over time.
  - Line graphs are best for showing trends over time or when we want to highlight the rate of change. It is very intuitive to use line graphs to track trends or patterns.

- Scatter plots are best for showing relationships or correlations between two variables. It is used a lot when one looks for patterns, clusters or outliers, or would like to explore the distribution of data points across different dimensions.
- Histograms are best for showing the distribution of a single continuous variable, or visualizing the distribution of data points across different ranges or intervals.
- Pie charts are best for showing parts of a whole or proportional data. It is often used when one has fewer categories (typically 3-5) that sum to 100%.

```
using Random, CairoMakie

# Data for the plots
categories = ["Product A", "Product B", "Product C", "Product D"]
sales = [150, 250, 200, 300]  # For bar chart

months = 1:12
sales_growth = [120, 150, 170, 180, 200, 220, 210, 230, 250, 270, 290, 310]  # For line graph

Random.seed!(1234)
x = rand(100)  # For scatter plot

proportions = [30, 25, 15, 30]  # For pie chart

# Combine individual plots into a 2x2 layout
f = Figure()
barplot(f[1, 1], 1:4, sales, axis=(xticks=(1:4, categories), title="bar"))
axis = Axis(f[1, 2], title="line")
lines!(f[1, 2], months, sales_growth)
axis = Axis(f[2, 1], title="scatter")
scatter!(axis, x)
axis = Axis(f[2, 2], title="pie")
pie!(axis, proportions, color=[:yellow, :orange, :red, :blue])
f
```

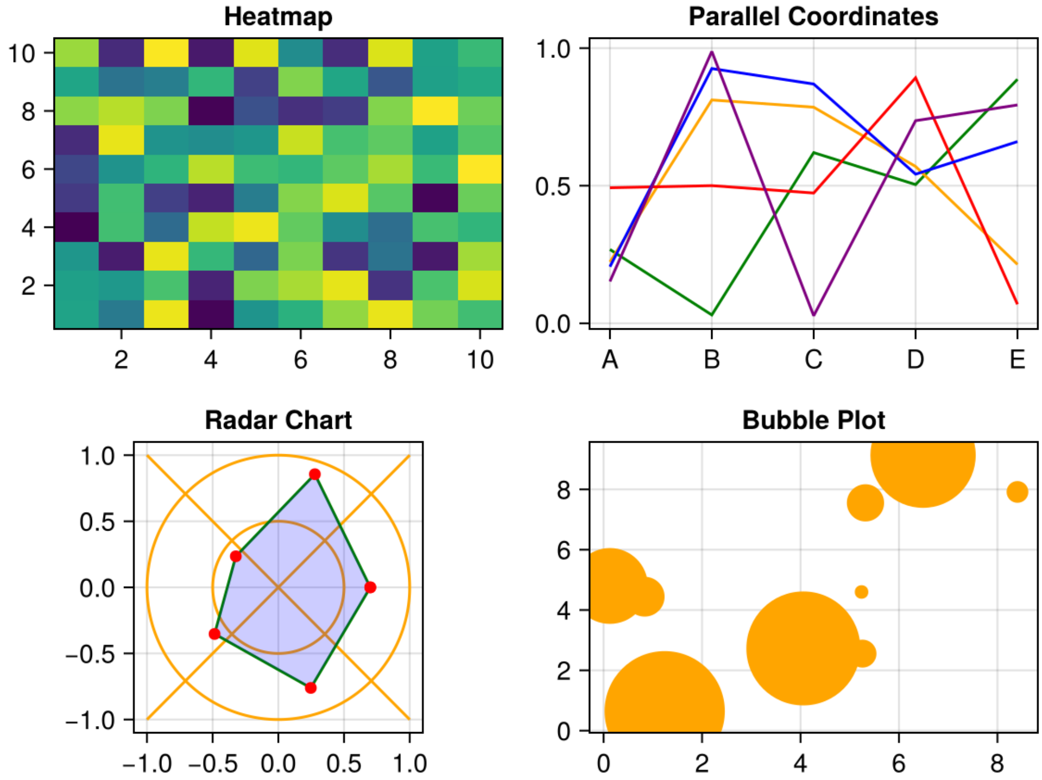



- Multivariate Visualizations: Heatmaps, parallel coordinates plots, radar charts, bubble charts.
  - Heatmaps are best for visualizing the intensity, interactions or relationships of values across two dimensions.
  - Bubble charts which are variants of scatter plots are best for showing relationships between three variables. One can easily highlight relative importance or magnitude using the size of bubbles (e.g., revenue, population).
  - Parallel coordinates plots are best for comparing multiple variables across different observations. They are often used for detecting patterns, correlations or relationships across multiple dimensions.
  - Radar charts are best for comparing multiple variables for a single or few observations, especially when one needs to show comparisons of several quantitative variables for one or more items, with each variable represented on an axis.

```
using Random, CairoMakie

Random.seed!(1234)
```

```
# Data for plots
z = rand(10, 10) # For heatmap
bubble_x = rand(10) * 10
bubble_y = rand(10) * 10
bubble_size = rand(10) * 100

# Dummy data for radar chart
radar_data = [0.7, 0.9, 0.4, 0.6, 0.8]
radar_labels = ["A", "B", "C", "D", "E"]

# Dummy data for parallel coordinates plot
parallel_data = rand(10, 5)
parallel_color = [:green, :orange, :red, :blue, :purple]

f = Figure()

# Heatmap (1,1)
ax1 = Axis(f[1, 1], title="Heatmap")
heatmap!(ax1, z)

# Parallel coordinates (1,2)
ax2 = Axis(f[1, 2], title="Parallel Coordinates")
for i in 1:size(parallel_data, 2)
    lines!(ax2, 1:size(parallel_data, 2), parallel_data[i, :], color=parallel_color[i])
end
ax2.xticks = (1:5, radar_labels)

# Radar Chart (2,1)
ax3 = Axis(f[2, 1], title="Radar Chart", aspect=1)
angles = range(0, 2π, length=length(radar_data) + 1)
r = [radar_data; radar_data[1]]
arc!(ax3, Point2f(0), 0.5, -π, π, color=:orange)
arc!(ax3, Point2f(0), 1, -π, π, color=:orange)
lines!(ax3, [-1, 1], [1, -1], color=:orange)
lines!(ax3, [-1, 1], [-1, 1], color=:orange)
lines!(ax3, cos.(angles) .* r, sin.(angles) .* r, color=:green)
poly!(ax3, cos.(angles) .* r, sin.(angles) .* r, color=(:blue, 0.2))
scatter!(ax3, cos.(angles) .* r, sin.(angles) .* r, color=:red)

# Bubble Plot (2,2)
ax4 = Axis(f[2, 2], title="Bubble Plot")
scatter!(ax4, bubble_x, bubble_y, markersize=bubble_size, color=:orange)
```

```
f
```

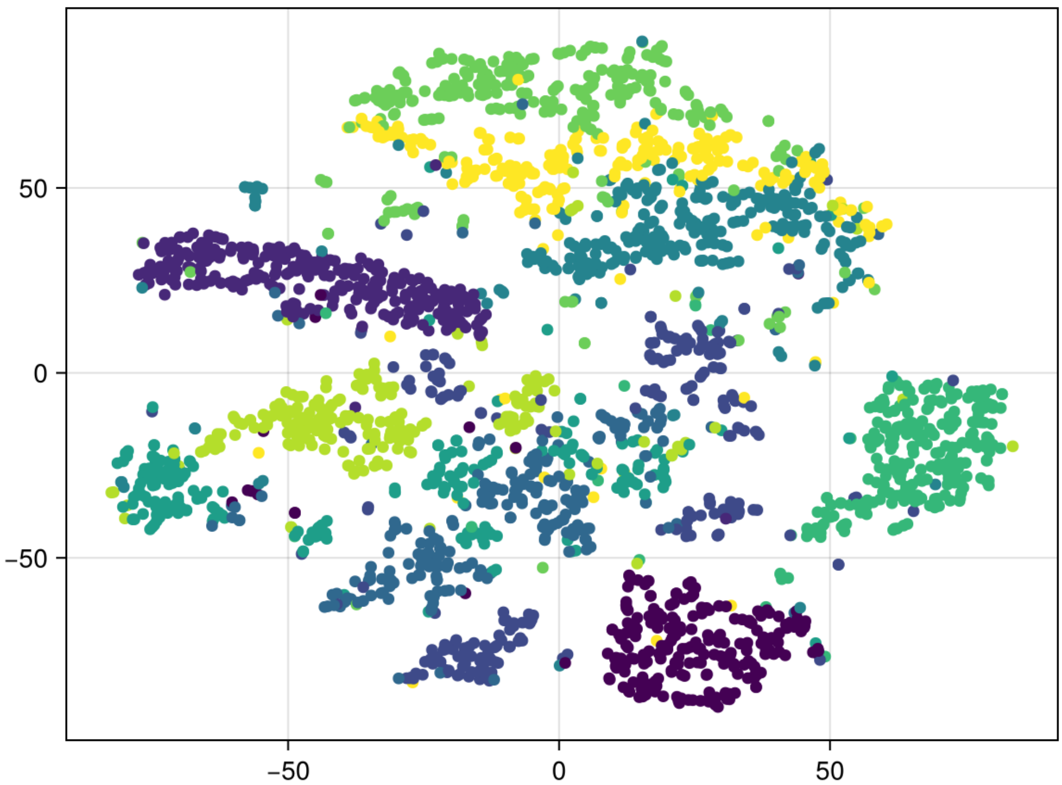



- Dimensionality Reduction: PCA plots, t-SNE, and UMAP for visualizing high-dimensional data. Here we show an example how high-dimensional data can be shown on a t-SNE plot.

```
using TSne, Statistics, MLDatasets, CairoMakie

alldata, allabels = MNIST(split=:train)[:];
data = reshape(permutedims(alldata[:, :, 1:2500], (3, 1, 2)), 2500, size(alldata, 1) * size(al

# Normalize the data, this should be done if there are large scale differences in the dataset
rescale(A; dims=1) = (A .- mean(A, dims=dims)) ./ max.(std(A, dims=dims), eps())

X = rescale(data, dims=1);

Y = tsne(X, 2, 50, 1000, 20.0);

f = Figure()
axis = Axis(f[1, 1])
```

```
scatter!(axis, Y[:, 1], Y[:, 2], color=Int.(allabels[1:size(Y, 1)]))
f
```

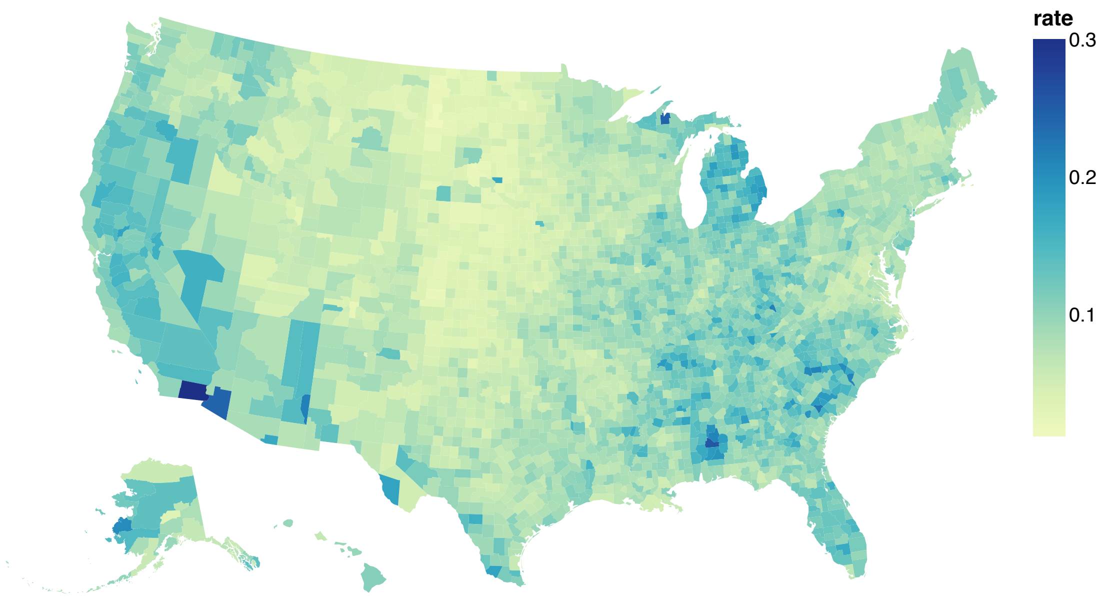
```
┌ Warning: ProgressMeter by default refresh meters with additional information in IJulia via `IJul
│  - To prevent this behaviour, do `ProgressMeter.ijulia_behavior(:append)`.
│  - To disable this warning message, do `ProgressMeter.ijulia_behavior(:clear)`.
└ @ ProgressMeter ~/.julia/packages/ProgressMeter/kVZZH/src/ProgressMeter.jl:594
Computing t-SNE 100%|████████████████████████████████████████| Time: 0:00:11
  KL_divergence:  1.0782
```




- Time Series Visualization: Line charts, area charts, time-series decomposition plots. Refer to Chapter 19 for a time series plot.
- Geospatial Visualization: Maps, choropleth maps for visualizing spatial data. Here we show how spatial data can be visualized using a choropleth map.

```
using VegaLite, VegaDatasets

us10m = dataset("us-10m")
```

```
unemployment = dataset("unemployment")

@vlplot(
    :geoshape,
    width = 500, height = 300,
    data = {
        values = us10m,
        format = {
            type = :topojson,
            feature = :counties
        }
    },
    transform = [{
        lookup = :id,
        from = {
            data = unemployment,
            key = :id,
            fields = ["rate"]
        }
    }],
    projection = {
        type = :albersUsa
    },
    color = "rate:q"
)
```



- Interactive Dashboards: Tools like Tableau, Power BI, Pluto for interactive and dynamic data exploration.

## 17.4. Julia packages

Julia has several powerful packages for data visualization, each with different strengths depending on your needs (e.g., interactive vs. static plots, ease of use vs. customization). Below are some of the most common visualization packages in Julia:

### 17.4.1. Makie.jl

`Makie` is designed for high-performance, interactive, and 3D visualization. It supports real-time interaction and is highly customizable. It supports 2D and 3D plotting and real-time interactivity. It is also extremely fast with GPU acceleration for certain operations.

### 17.4.2. Plots.jl

`Plots` is one of the most versatile and popular Julia plotting libraries. It provides a high-level interface for different plotting backends (e.g., GR, Plotly, PyPlot, PGFPlotsX, etc.). It uses a high-level syntax, easy to use. It supports multiple backends for both static and interactive plots with extensive customization options.

### 17.4.3. Gadfly.jl

`Gadfly` is a grammar-of-graphics plotting system similar to ggplot2 in R. It is very flexible for producing static visualizations. It supports grammar of graphics syntax, great for statistical graphics. It is also easy to learn, especially for users familiar with `ggplot2`. Besides it is good for quick prototyping and publication-quality plots.

### 17.4.4. PlotlyJS.jl

`PlotlyJS` is the Julia wrapper for the interactive `Plotly.js` library. It provides rich interactive visualizations with support for hover information, zooming, and more. It is great for interactive web-based visualizations. It supports a wide range of plot types including 3D, maps, and more, and can be used for building dashboards.

### 17.4.5. VegaLite.jl

`VegaLite` is a wrapper for the Vega-Lite visualization grammar, which focuses on interactive, declarative visualizations. It works well for complex, layered visualizations. It is grammar-of-graphics based which easily produces interactive charts. It is built for declarative specification of visualizations.

### 17.4.6. PyPlot.jl

`PyPlot` is a Julia wrapper for the popular Python `matplotlib` library. It's suitable for users familiar with Python who want the same plotting capabilities in Julia. It leverages the full capabilities of matplotlib and it can produce high-quality, publication-ready plots. It is great for scientific visualization.

### 17.4.7. StatsPlots.jl

`StatsPlots` extends `Plots` by adding statistical plot types such as boxplots, violin plots, histograms, and density plots. It's ideal for users who frequently work with statistical data. It is specialized for statistical visualizations. It allows easy integration with Julia's statistical packages like `DataFrames` and `StatsBase`.

### 17.4.8. GraphPlot.jl

This package is used to plot graphs (networks), such as social network visualizations or other graph-related problems. It supports integration with the `LightGraphs` package for graph analytics.

### 17.4.9. UnicodePlots.jl

`UnicodePlots` provides simple plotting capabilities in the terminal using Unicode characters, making it lightweight and fast. There are no external dependencies. It is great for quick plotting within the terminal.

Each package has unique strengths depending on the use case, so the best choice depends on our specific needs, the type of data, and whether we need interactive or static visualizations.

# 18. Matrices and Their Uses

## 18.1. 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.

## 18.2. Matrix manipulation

We first review basic matrix manipulation routines before going into more advanced topics.

### 18.2.1. Addition and subtraction

This is how matrices are added and subtracted (element-wise) in Julia.

```
# Define two matrices
A = [1 2 3;
    4 5 6;
    7 8 9]
B = [9 8 7;
    6 5 4;
    3 2 1]
# Perform element-wise matrix addition and subtraction
C = A .+ B
D = A .- B
# Display the result
println("Result of matrix addition:")
println(C)
println("Result of matrix subtraction:")
println(D)
```

```
Result of matrix addition:
[10 10 10; 10 10 10; 10 10 10]
Result of matrix subtraction:
[-8 -6 -4; -2 0 2; 4 6 8]
```

### 18.2.2. Transpose

The transpose of a matrix swaps its rows and columns. This is how matrices can be transposed in Julia.

```
# Define a matrix
A = [1 2 3;
    4 5 6;
    7 8 9]
# Perform matrix transpose
B = A'
# Display the result
println("Result of matrix transpose:")
println(B)
```

```
Result of matrix transpose:
[1 4 7; 2 5 8; 3 6 9]
```

### 18.2.3. Determinant

The determinant provides useful properties about a matrix, such as whether it is invertible.

Given a matrix A

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

the determinant of matrix A can be calculated as

$$\det(\mathbf{A}) = \sum_{j=1}^{n} (-1)^{1+j} a_{1j} \det(\mathbf{A}_{1j})$$

This is how determinant of matrices can be calculated in Julia.

```
using LinearAlgebra

# Define a matrix
A = [1 2 3;
    5 10 20;
    7 8 9]
# Perform matrix determinant calculation
B = det(A)
# Display the result
println("Result of matrix determinant:")
println(B)
```

```
Result of matrix determinant:
30.000000000000007
```

### 18.2.4. Trace

The trace of a square matrix is the sum of its diagonal elements. This is how trace of matrices can be calculated in Julia.

```
using LinearAlgebra

# Define a matrix
A = [1 2 3;
    5 10 20;
    7 8 9]
# Perform matrix determinant calculation
B = tr(A)
# Display the result
println("Result of matrix trace:")
println(B)
```

```
Result of matrix trace:
20
```

### 18.2.5. Norm

A matrix norm provides a way to measure the size or "magnitude" of a matrix. Common usages of norms include error analysis, regularization in machine learning and measuring similarity and distance.

The Frobenius norm of a matrix is defined as:

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|A_{ij}|^2}$$

This is how the norm can be calculated in Julia.

```
using LinearAlgebra

# Define a matrix
A = [1 2 3;
    4 5 6;
    7 8 9]
# Perform matrix determinant calculation
B = norm(A)
# Display the result
println("Result of matrix norm:")
println(B)
```

```
Result of matrix norm:
16.881943016134134
```

### 18.2.6. Multiplication

Matrix multiplication (non-element-wise) plays a critical role in various fields, such as mathematics, physics, computer science, economics, and machine learning. Some key areas include:

- Linear transformation. A matrix represents a linear transformation (e.g., rotation, scaling, translation) in vector spaces. When we multiply a matrix by a vector, essentially we apply the corresponding transformation to the vector.
- Computer graphics and image processing. Images are represented as matrices of pixels, and matrix operations such as convolutions are used for filtering and feature extraction.
- Neural network construction. Matrix multiplication is fundamental for training and using neural networks.
- Systems of linear equations. Many real-world problems reduce to solving systems of linear equations.

This is how matrices are multiplied non-element-wise in Julia.

```
# Define two matrices
A = [1 2 3;
    4 5 6;
    7 8 9]
B = [9 8 7;
    6 5 4;
    3 2 1]
# Perform non-element-wise matrix multiplication
C = A * B
# Display the result
println("Result of matrix multiplication:")
println(C)
```

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

On the other hand, this is how matrices are multiplied element-wise in Julia.

```
# Define two matrices
A = [1 2 3;
    4 5 6;
    7 8 9]
B = [9 8 7;
    6 5 4;
    3 2 1]
# Perform element-wise matrix multiplication
C = A .* B
# Display the result
println("Result of matrix multiplication:")
println(C)
```

```
Result of matrix multiplication:
[9 16 21; 24 25 24; 21 16 9]
```

### 18.2.7. Inversion

Matrix inversion also plays a critical role in various fields mentioned above, to solve systems of equations or to reverse transformations.

This is how matrices can be inversed in Julia.

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

```
Inverse of matrix A:
[-1.9999999999999996 0.9999999999999998; 1.4999999999999998 -0.4999999999999999]
```

**Note**

For a matrix to be inversed, it must meet several important criteria. - Square Matrix. The matrix must be square, meaning it has the same number of rows and columns. - The determinant of the matrix must be non-zero.

## 18.3. Matrix decomposition

### 18.3.1. Eigenvalues

Eigenvalue decomposition, also known as eigendecomposition, is a matrix factorization that decomposes a matrix into its eigenvectors and eigenvalues. Eigenvalues and eigenvectors are fundamental concepts in linear algebra and play key roles include:

- Eigenvalues help in analyzing how linear transformations affect vectors in a vector space.
- Eigenvalues facilitate the diagonalization of matrices and simplify the calculations.
- In systems of differential equations, eigenvalues help determine the stability of equilibrium points.
- In mechanical systems, eigenvalues represent natural frequencies of vibration. Each eigenvalue corresponds to a mode of vibration, which is critical for understanding dynamic behavior and preventing resonance.
- In quantum mechanics, eigenvalues represent measurable quantities (observables) associated with quantum states.
- In graph theory, eigenvalues of the adjacency matrix provide insights into the properties of the graph, such as connectivity, stability, and clustering.
- Many algorithms in data science, like clustering and factorization methods, rely on eigenvalues to identify patterns and reduce dimensionality, which enhances computational efficiency and interpretability.

```
using LinearAlgebra

# Create a square matrix
A = [1 2 3;
    4 5 6;
    7 8 9]
# Perform eigenvalue decomposition
eigen_A = eigen(A)
# Extract eigenvalues and eigenvectors
λ = eigen_A.values
V = eigen_A.vectors

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

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

Eigenvalues:
[-1.1168439698070434, -8.582743335036247e-16, 16.11684396980703]

Eigenvectors:
[-0.7858302387420671 0.4082482904638635 -0.2319706872462857; -0.0867513392566285 -0.8164965809277
```

### 18.3.2. Singular values

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

- Matrix factorization, which simplifies many matrix operations, making it easier to analyze and manipulate data.
- Dimensionality reduction. This is particularly useful in high-dimensional data scenarios, where reducing dimensions helps eliminate noise and improve computational efficiency.
- SVD can be used for data compression, particularly in image processing.
- SVD helps filter out noise in data analysis.

- SVD provides a robust method for solving linear equations, particularly when the matrix is ill-conditioned or singular.
- In machine learning, SVD helps extract important features from datasets.
- SVD provides insights into the relationships within data. The singular values indicate the strength of the relationship, while the singular vectors offer a way to visualize and interpret those relationships.

```julia
using LinearAlgebra

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

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

```
Original Matrix:
[0.09534170901520977 0.45381877044838814 0.6527168485298647; 0.7200907359307361 0.86412979412193

Left Singular Vectors:
[-0.39924239902150505 0.2128392404999452 0.8917363446003508; -0.733307383946467 -0.26194636747860

Singular Values:
[1.8904854510328033, 0.7000146472087132, 0.24878325806784982]

Right Singular Vectors:
[-0.43717531056579234 -0.3275009166094779 -0.8376281379297467; -0.565900788992606 -0.623702215760
```

```
Reconstructed Matrix:
[0.09534170901521027 0.4538187704483879 0.6527168485298642; 0.7200907359307359 0.864129794121931
```

### 18.3.3. Matrix factorization and fatorization machines

Matrix factorization is a popular technique in recommendation systems for modeling user-item interactions and making personalized recommendations. The core idea behind matrix factorization is to decompose the user-item interaction matrix into two lower-dimensional matrices, capturing latent factors that represent user preferences and item characteristics. By learning these latent factors, the recommendation system can make predictions for unseen user-item pairs.

Factorization Machines (FM) are a type of supervised machine learning model designed for tasks such as regression and classification, especially in the context of recommendation systems and predictive modeling with sparse data. FM models extend traditional linear models by incorporating interactions between features, allowing them to capture complex relationships within the data.

The following shows an example how to use matrix factorization to do recommendations.

```julia
using Recommendation, SparseArrays, MLDataUtils

# Generate synthetic user-item interaction data
num_users = 100
num_items = 50
num_ratings = 500
user_ids = rand(1:num_users, num_ratings)
item_ids = rand(1:num_items, num_ratings)
ratings = rand(1:5, num_ratings)
# Create a sparse user-item matrix
user_item_matrix = sparse(user_ids, item_ids, ratings)
# Split data into training and testing sets
train_data, test_data = splitobs(user_item_matrix, 0.8)
# Set parameters for matrix factorization
num_factors = 10
num_iterations = 10
# Train matrix factorization model
data = DataAccessor(user_item_matrix)
recommender = MF(data) # FactorizationMachines(data) alternatively
fit!(recommender)
# Predict ratings for the test set
rec = Dict()
for user in 1:num_users
```

```
    rec[user] = recommend(recommender, user, num_items, collect(1:num_items))
end
# Evaluate model performance
predictions = []
for (i, j, v) in zip(findnz(test_data.data)[1], findnz(test_data.data)[2], findnz(test_data.da
    for p in rec[i]
        if p[1] == j
            push!(predictions, p[2])
            break
        end
    end
end
rmse = measure(RMSE(), predictions, nonzeros(test_data.data))
println("Root Mean Squared Error (RMSE): ", rmse)
```

```
Root Mean Squared Error (RMSE): 1.334942466436463
```

```
WARNING: using Recommendation.isdefined in module Main conflicts with an existing identifier.
```

### 18.3.4. Principal component analysis

Principal Component Analysis (PCA) is a widely used technique in various fields for dimensionality reduction, data visualization, feature extraction, and noise reduction. PCA can also be applied to detect anomalies or outliers in the data by identifying data points that deviate significantly from the normal patterns captured by the principal components. Anomalies may appear as data points with large reconstruction errors or as outliers in the low-dimensional space spanned by the principal components.

```
using MultivariateStats

# Generate some synthetic data
data = randn(100, 5)  # 100 samples, 5 features
# Perform PCA
pca_model = fit(PCA, data; maxoutdim=2)  # Project to 2 principal components
# Transform the data
transformed_data = transform(pca_model, data)
# Access principal components and explained variance ratio
principal_components = pca_model.prinvars
explained_variance_ratio = pca_model.prinvars / sum(pca_model.prinvars)
```

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

```
Principal Components:
[31.215988300869377, 26.900582562776133]
Explained Variance Ratio:
[0.5371271538733604, 0.4628728461266396]
```

# 19. Learning from Data

## 19.1. In this chapter

We will touch on how to use data to inform a model: fitting parameters, forecasting, and fundamental limitations on prediction.

## 19.2. How to learn from data

### 19.2.1. Understand the problem and define goals

- Clarify objectives: What we want to achieve with the data (e.g., prediction, classification, clustering, or insight extraction).
- Identify key metrics: Determine how success will be measured (accuracy, RMSE, precision, etc.).
- Know the context: Understand the domain and business problem one is addressing to shape the data analysis process.

### 19.2.2. Collect data

Various data may be available in different formats. - Ensure data relevance: The data should be relevant to the problem. - Consider data quality: Collect data with high accuracy, completeness, and consistency.

### 19.2.3. Explore and preprocess the data

This involves data cleaning and preparation to ensure the dataset is suitable for analysis.

- Handle missing data: We could impute missing values (mean, median, or KNN imputation), or drop rows/columns with excessive missing data.
- Deal with outliers: Use statistical techniques (e.g., z-scores) to detect and remove or cap extreme values.

- Feature scaling: Apply normalization or standardization to ensure features are on comparable scales (important for algorithms like SVM, K-means, etc.).
- Encode categorical data: Use techniques such as: one-hot encoding for nominal data, or label encoding or ordinal encoding for ordered categories.
- Data visualization: Use tools like `Makie.jl` to visualize distributions, correlations, and missing values.

### 19.2.4. Exploratory data analysis (EDA)

EDA helps discover patterns, relationships, and insights within the data. One can do the following, but not limited, to these analyses:

- Summary statistics: Check mean, variance, skewness, and correlations between variables.
- Visualize relationships: Use histograms, scatter plots, box plots, and heatmaps to identify trends and correlations.
- Detect multicollinearity: Check correlations between independent variables (e.g., Pearson's correlation matrix).

### 19.2.5. Select and engineer features

Feature selection and engineering help improve model performance by focusing on the most relevant information.

### 19.2.6. Choose the right algorithm or model

Depending on our problem type, choose appropriate algorithms for learning from the data:

- Supervised Learning (with labeled data):
  - Classification: Logistic regression, SVM, decision trees, random forests, or neural networks.
  - Regression: Linear regression, ridge regression, or gradient boosting.
- Unsupervised Learning (without labeled data):
  - Clustering: K-means, DBSCAN, hierarchical clustering.
  - Dimensionality Reduction: PCA, t-SNE, or UMAP.
- Reinforcement Learning: Learn from interactions with an environment (e.g., Q-learning, Deep Q-Networks).

### 19.2.7. Train and evaluate the model

- Split the data: Use either a train-test split (e.g., 80/20 or 70/30 split) or a cross-validation (e.g., k-fold cross-validation).
- Fit the model: Train the model on the training set.
- Evaluate the model: Use evaluation metrics appropriate to the task.

### 19.2.8. Tune hyperparameters

Hyperparameters control how models learn. One can use techniques like the following to tune hyperparameters:

- Grid search: Test a range of hyperparameter values.
- Random search: Randomly explore combinations of hyperparameters.
- Bayesian optimization: Use probabilistic models to guide hyperparameter search.

### 19.2.9. Deploy and Monitor the Model

Once the model performs well, deploy it to make predictions on new data.

- Model deployment platforms: Use tools like Flask, FastAPI, or MLOps platforms.
- Monitor performance: Continuously monitor metrics to detect concept drift or performance degradation.

### 19.2.10. Draw Insights and Make Decisions

Finally, interpret the results and use insights to make decisions or recommendations. Effective communication of findings is essential, especially for stakeholders.

- Visualization: Use dashboards or reports to communicate findings.
- Interpretability: Use explainable AI (e.g., SHAP values) to make model predictions transparent.

### 19.2.11. Limitations

However, there are certain fundamental limitations:

- There may often be inherent uncertainty and noise in the data itself.
- Every model has its own assumptions and simplifications.

- There may be non-stationarity in the data, especially in financial data. Non-stationary processes change over time, meaning that patterns learned from past data may no longer be valid in the future.
- Models may be overfitting or underfitting. Overfitting occurs when a model is too complex and captures noise instead of the underlying pattern, leading to poor generalization to new data. Underfitting occurs when the model is too simple to capture the relevant structure in the data.
- Sometimes in high-dimensional spaces, data becomes sparse, and meaningful patterns are harder to identify.
- Some predictions may be limited by ethical concerns (e.g., predicting criminal behavior) or legal restrictions (e.g., privacy laws that limit data collection).

## 19.3. Applications

### 19.3.1. Parameter fitting

Refer to **?@sec-optimization** on Optimization for more details.

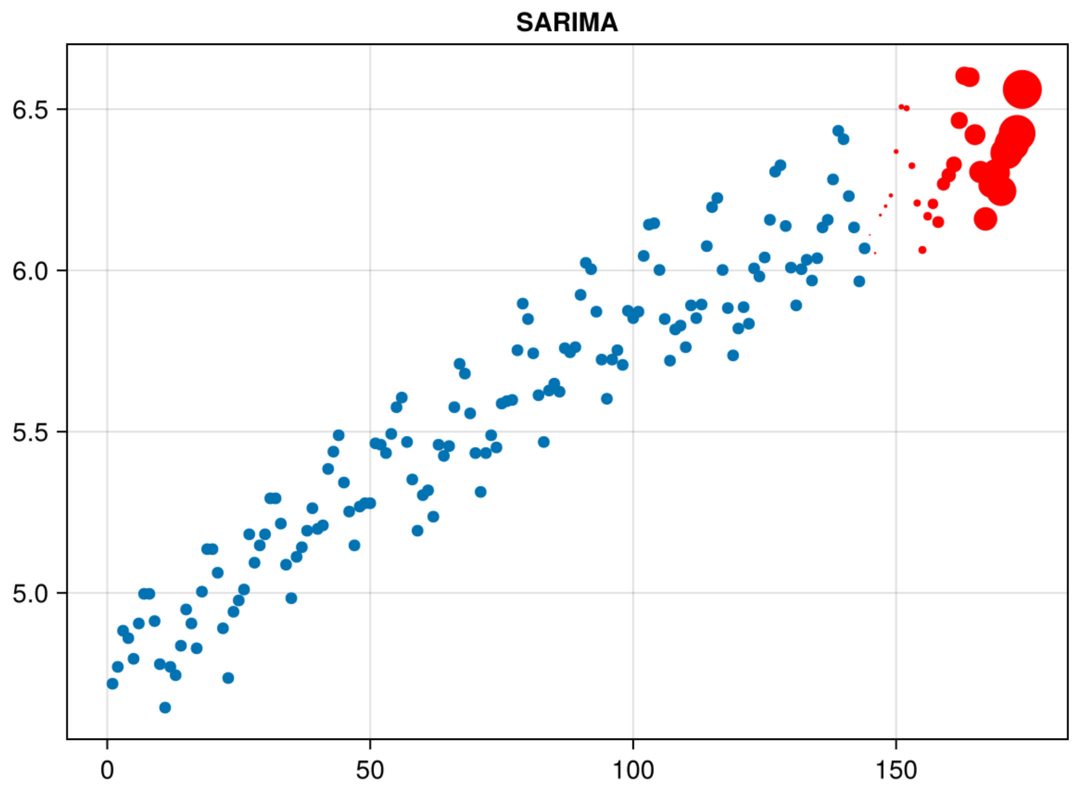
### 19.3.2. Forecasting

Forecasting is the process of making predictions about future events or outcomes based on historical data, patterns, and trends. It involves the use of statistical methods, machine learning models, or expert judgment to estimate future values in a time series or predict the likelihood of specific events. Forecasting is widely used in fields like economics, finance, meteorology, supply chain management, and business planning.

Here is an example how to do time series forecasting in Julia, where point sizes show covariance of predictions:

```
using CSV, DataFrames, CairoMakie, StateSpaceModels

airp = CSV.read(StateSpaceModels.AIR_PASSENGERS, DataFrame)
log_air_passengers = log.(airp.passengers)
steps_existing = length(log_air_passengers)
steps_ahead = 30

# SARIMA
model_sarima = SARIMA(log_air_passengers; order = (0, 1, 1), seasonal_order = (0, 1, 1, 12))
fit!(model_sarima)
forec_sarima = forecast(model_sarima, steps_ahead)

f = Figure()
```

```
axis = Axis(f[1, 1], title="SARIMA")
scatter!(1:steps_existing, log_air_passengers)
scatter!(steps_existing+1:steps_existing+steps_ahead, map(x -> x[1], forec_sarima.expected_val
f
```




### 19.3.3. Static and dynamic validation

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

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

The following example shows how to do a static validation in Julia.

```
using Statistics

# Generate synthetic time series data
num_samples = 100
data = rand(num_samples)
X = [ones(num_samples) data]
y = 2data .+ 1 .+ 0.1 * randn(num_samples, 1)  # dependent variable with noise
# Train the model on the training set
θ = X \ y
# Predictions
y_pred = θ[2] .* data .+ θ[1]
# Compute evaluation metrics
mse = mean((y_pred .- y) .^ 2)
mae = mean(abs.(y_pred .- y))

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

```
Static validation results:
Mean Squared Error (MSE): 0.007085378286062892
Mean Absolute Error (MAE): 0.0649437555244629
```

The following example shows how to do a dynamic validation in Julia.

```
using Statistics

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

```
    push!(mae_dyn, mae)
end

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

```
Dynamic validation results:
Mean Squared Error (MSE): 0.010171104902429975
Mean Absolute Error (MAE): 0.07970658065722214
```

### 19.3.4. Implied rate analysis

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

```
using Zygote

# Define the bond cash flows and prices
cash_flows = [100, 100, 100, 100, 1000]  # Coupons and principal
prices = [950, 960, 1010, 1020, 1050]  # Market prices

# Define a function to calculate the present value of cash flows given a rate
function present_value(rate, cash_flows)
    pv = 0
    for (i, cf) in enumerate(cash_flows)
        pv += cf / (1 + rate) ^ i
    end
    return pv
end

# Define a function to calculate the implied rate using bisection method
function implied_rate(cash_flows, price)
    f(rate) = present_value(rate, cash_flows) - price
    return rootassign(f, 0.0, 1.0)
end
function rootassign(f, l, u)
    # Define an initial value
    x = 0.05
    # tolerance of difference in value
```

```
    tol = 1e-6
    # maximum number of iteration of the algorithm
    max_iter = 100
    iter = 0
    while abs(f(x)) > tol && iter < max_iter
        x -= f(x) / gradient(f, x)[1]
        iter += 1
    end
    if iter < max_iter && l < x < u
        return x
    else
        return -1.0
    end
end

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

```
Implied rate for bond 1: 0.09658339166435045
Implied rate for bond 2: 0.09380219311021369
Implied rate for bond 3: 0.08046244727376842
Implied rate for bond 4: 0.0779014164014789
Implied rate for bond 5: 0.07041724037694008
```

# Part V.

# Developing In Julia

Aside from the essentials necessary to convey the topics, the prior chapters have provided relatively limited insight into the *workflow* of developing in Julia. An analogy is that we have, up to this point, talked about using hammers and saws to create widgets. And some advanced topics like how to make a widget run faster and smoother in different machines. This section of the book talks about the *process* of building and designing widgets, helping you to make your widget making factory run smoothly.

This chapter is heavy with Julia-specific tips and advice. We have deliberately delayed this content until well into the book, focusing on the concepts instead of getting bogged down on language-specific details. In this section, we dive into the 'messy' business of building bigger, integrated things and tools to make that easier. Readers taking the concepts to other languages need not burden themselves with the details of Julia workflows and therefore can jump to the section beginning with Chapter 24.

> **i Note**
>
> The chapters in this section are adapted from Modern Julia Workflows, originally written by G. Dalle, J. Smit, A. Hill. These chapters are a derivative of that work, and is also licensed CC BY-SA 4.0.
> The content was modified to fit the other content of this book (adding cross-references, removing duplicated content) and to add/subtract elements that the authors of this book deemed more apprнезrpriate for the financial modeler.

# 20. Wrting Julia Code

## 20.1. In this chapter

Installing and setting up your Julia environment. Text editor and REPL editing environments. Setting up your global environment for development. Creating packages. Logging and debugging code.

## 20.2. Getting help

Before you write any line of code, it's good to know where to find help. The official help page is a good place to start. In particular, the Julia community is always happy to guide beginners.

As a rule of thumb, the Discourse forum is where you should ask your questions to make the answers discoverable for future users. If you just want to chat with someone, you have a choice between the open source Zulip and the closed source Slack.

## 20.3. Installation

The most natural starting point to install Julia onto your system is the Julia downloads page, which will tell you to use `juliaup`.

1. Windows users can download Julia and `juliaup` together from the Windows Store.
2. OSX or Linux users can execute the following terminal command:

```
curl -fsSL https://install.julialang.org | sh
```

In both cases, this will make the `juliaup` and `julia` commands accessible from the terminal (or Windows Powershell). On Windows this will also create an application launcher. All users can start Julia by running

```
julia
```

Meanwhile, `juliaup` provides various utilities to download, update, organize and switch between different Julia versions. As a bonus, you no longer have to manually specify the path to your executable. This all works thanks to adaptive shortcuts called "channels", which allow you to access specific Julia versions without giving their exact number.

For instance, the `release` channel will always point to the current stable version, and the `lts` channel will always point to the long-term support version. Upon installation of `juliaup`, the current stable version of Julia is downloaded and selected as the default.

> **Tip**
>
> To use other channels, add them to `juliaup` and put a `+` in front of the channel name when you start Julia:
>
> ```
> juliaup add lts
> julia +lts
> ```
>
> You can get an overview of the channels installed on your computer with
>
> ```
> juliaup status
> ```
>
> When new versions are tagged, the version associated with a given channel can change, which means a new executable needs to be downloaded. If you want to catch up with the latest developments, just do
>
> ```
> juliaup update
> ```

## 20.4. REPL

The Read-Eval-Print Loop (or REPL) is the most basic way to interact with Julia, check out its documentation for details. You can start a REPL by typing `julia` into a terminal, or by clicking on the Julia application in your computer. It will allow you to play around with arbitrary Julia code:

```
julia> a, b = 1, 2;

julia> a + b
3
```

This is the standard (Julia) mode of the REPL, but there are three other modes you need to know. Each mode is entered by typing a specific character after the `julia>` prompt.

Once you're in a non-Julia mode, you stay there for every command you run. To exit it, hit backspace after the prompt and you'll get the `julia>` prompt back.

### 20.4.1. Help mode (?)

By pressing ? you can obtain information and metadata about Julia objects (functions, types, etc.) or unicode symbols. The query fetches the docstring of the object, which explains how to use it.

```
help?> println
search: println print sprint pointer printstyled

  println([io::IO], xs...)

  Print (using print) xs to io followed by a newline. If io is not supplied, prints to the def

  See also printstyled to add colors etc.

  Examples
  ========

  julia> println("Hello, world")
  Hello, world

  julia> io = IOBuffer();

  julia> println(io, "Hello", ',', " world.")

  julia> String(take!(io))
  "Hello, world.\n"
```

If you don't know the exact name you are looking for, type a word surrounded by quotes to see in which docstrings it pops up.

### 20.4.2. Package mode (])

By pressing ] you access Pkg.jl, Julia's integrated package manager, whose documentation is an absolute must-read. Pkg.jl allows you to:

- `]activate` different local, shared or temporary environments;
- `]instantiate` them by downloading the necessary packages;
- `]add`, `]update` (or `]up`) and `]remove` (or `]rm`) packages;

- get the `]status` (or `]st`) of your current environment.

As an illustration, we download the package Example.jl inside a new environment we call `demo` (which will create an associated folder if it does not exist):

```
(demo) pkg> activate demo
  Activating new project at `~/demo`

(demo) pkg> add Example
   Resolving package versions...
    Updating `~/demo/Project.toml`
  [7876af07] + Example v0.5.5
    Updating `~/demo/Manifest.toml`
  [7876af07] + Example v0.5.5


(demo) pkg> status
Status `~/demo/Project.toml`
  [7876af07] Example v0.5.5
```

Note that the same keywords are also available in Julia mode:

```
julia> using Pkg

julia> Pkg.rm("Example")
    Updating `~/demo/Project.toml`
  [7876af07] - Example v0.5.5
    Updating `~/demo/Manifest.toml`
  [7876af07] - Example v0.5.5
```

The package mode itself also has a help mode, accessed with `?`, in case you're lost among all these new keywords.

### 20.4.3. Shell mode (;)

By pressing **;** you enter a terminal, where you can execute any command you want, such as changing the working directory to the folder we just created:

```
shell> cd demo
/Users/myself/demo
```

## 20.5. Editor

In theory, any text editor suffices to write and modify Julia code. In practice, an Integrated Development Environment (or IDE) makes the experience much more pleasant, thanks to code-related utilities and language-specific plugins.

The best IDE for Julia is Visual Studio Code, or VSCode, developed by Microsoft. The Julia VSCode extension is the most feature-rich of all Julia IDE plugins. You can download it from the VSCode Marketplace and read its documentation.

> **Tip**
>
> In what follows, we will sometimes mention commands and keyboard shortcuts provided by this extension. But the only shortcut you need to remember is `Ctrl + Shift + P` (or `Cmd + Shift + P` on Mac): this opens the VSCode command palette, in which you can search for any command. Type `julia` in the command palette to see what you can do.

## 20.6. Running code

You can execute a Julia script from your terminal, but in most cases that is not what you want to do.

```
julia myfile.jl  # avoid this
```

Julia has a rather high startup and compilation latency. If you only use scripts, you will pay this cost every time you run a slightly modified version of your code. That is why many Julia developers fire up a REPL at the beginning of the day and run all of their code there, chunk by chunk, in an interactive way. Full files can be run interactively from the REPL with the `include` function.

```
julia> include("myfile.jl")
```

Alternatively, `includet` from the Revise.jl package can be used to "include and track" a file. This will automatically update changes to function definitions in the file in the running REPL session.

> **Tip**
>
> Running code is made much easier by the following commands:
>
> - `Julia: Restart REPL` (shortcut `Alt + J` then `Alt + R`) - this will open or restart the integrated Julia REPL. It is different from opening a plain VSCode

> terminal and launching Julia manually from there.
> - `Julia: Execute Code in REPL and Move` (shortcut `Shift + Enter`) - this will execute the selected code in the integrated Julia REPL, like a notebook.

When keeping the same REPL open for a long time, it's common to end up with a "polluted" workspace where the definitions of certain variables or functions have been overwritten in unexpected ways. This, along with other events like `struct` redefinitions, might force you to restart your REPL now and again, and that's okay.

## 20.7. Notebooks

Notebooks are a popular alternative to IDEs when it comes to short and self-contained code, typically in data science. They are also a good fit for literate programming, where lines of code are interspersed with comments and explanations.

The most well-known notebook ecosystem is Jupyter, which supports **Ju**lia, **Py**thon and **R** as its three core languages. To use it with Julia, you will need to install the IJulia.jl backend. Then, if you have also installed Jupyter with `pip install jupyterlab`, you can run this command to launch the server:

```
jupyter lab
```

If you only have IJulia.jl on your system, you can run this snippet instead:

```
julia> using IJulia

julia> IJulia.notebook()
```

> Tip
>
> Jupyter notebooks can be opened, modified and run directly from VSCode. Thanks to the Julia extension, you don't even need to install IJulia.jl or Jupyter first.

`Pluto.jl` is a newer, pure-Julia tool, adding reactivity and interactivity. It is also more amenable to version control than Jupyter notebooks because notebooks are saved as plain Julia scripts. Pluto is unique to Julia because of the language's ability to introspect and analyze dependencies in its own code. Pluto also has built-in package/environment management, meaning that Pluto notebooks contains all the code needed to reproduce results (as long as Julia and Pluto are installed).

To try out Pluto, install the package and then run

```
julia> using Pluto

julia> Pluto.run()
```

## 20.8. Markdown

Markdown is a markup language used to add formatting elements to plain text content, such as Julia docstrings. Additionally, other tools such as Quarto (described below) are built using Markdown notation as the basis for their formatting, so it's useful to know about Markdown and the most essential

### 20.8.1. Plain Text Markdown

Plain text markdown files, which have the `.md` extension, are not used for interactive programming, meaning one cannot run code written in the file. As a result, plain text markdown files are usually rendered into a final product by other software.

This is an example of a plain text markdown file, including a code example contained within the """""" block:

````
# Title

## Section Header

This is example text.

```julia
println("hello world")
```
````

### 20.8.2. Quarto

Quarto "is an open-source scientific and technical publishing system." Quarto makes a plain text markdown file (`.md`) alternative called Quarto markdown file (`.qmd`).

Quarto markdown files like plain text markdown files also integrate with editors, such as VSCode.

> **Tip**
>
> Install the Quarto extension for a streamlined experience.

Unlike plain text markdown files, Quarto markdown files have executable code chunks. These code chunks provide a functionality similar to notebooks, thus Quarto markdown files are an alternative to notebooks. Additionally, Quarto markdown files give users additional control over output and styling via the YAML header at the top of the `.qmd` file.

As of Quarto version 1.5, users can choose from two Julia engines to execute code - a native Julia engine and IJulia.jl. The primary difference between the native Julia engine and IJulia.jl is that the native Julia engine does not depend on Python and can utilize local environments. For this reason it's recommended to start with the native Julia engine. Learn more about the native Julia engine in Quarto's documentation.

This book is built using Quarto documents to create the associated typeset book and website.

## 20.9. Environments and Dependencies

Julia comes bundled with Pkg.jl, an environment and package manager. It enables installation of packages from registries, pinning versions for compatibility, and analyzing your dependencies. Environment 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.)

### 20.9.1. Project.toml

A `Project.toml` file defines attributes about the current project and its dependencies. Julia uses this to understand how to reference your current project and what dependencies it should look for from registries when instantiating the project.

> **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.com> and contributors"]
version = "2.1.0"                                             ③

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

[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[1].
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.

[1] 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.

⑤ The `compat` section defines compatibility with packages can be enforced (via SemVer) to clarify which versions are allowed to be installed in case incompatibilities arise.

When you `instantiate` a project (see **?@sec-environment-details** 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.

**i** 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).

### 20.9.2. Manifest.toml

The `Manifest.toml` file includes a record of all external dependencies used by the project at hand. Unlike `Project.toml`, this file gets machine generated when Julia instantiates or updates the environment. The contents are basically a long list of your direct dependencies and the dependencies of those direct dependencies and looks something like this:

```
julia_version = "1.10.0"
manifest_format = "2.0"
project_hash = "5fea00df4808d89f9c977d15b8ee992bd408081b"

[[deps.AbstractFFTs]]
deps = ["LinearAlgebra"]
git-tree-sha1 = "d92ad398961a3ed262d8bf04a1a2b8340f915fef"
uuid = "621f4979-c628-5d54-868e-fcf4e3e8185c"
version = "1.5.0"
weakdeps = ["ChainRulesCore", "Test"]

    [deps.AbstractFFTs.extensions]
    AbstractFFTsChainRulesCoreExt = "ChainRulesCore"
    AbstractFFTsTestExt = "Test"

... many more lines
```

**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.

### 20.9.3. Reproducibility

Reproducibility fulfills both practical and principled goals. *Practical* in that we can record the complex chain of dependencies that is used in modern computing in order to potentially re-create a result or demonstrate an audit trail of the tools used. *Principled* in that there are circumstances (like science research) in which we want to be able to replicate results. The combination of `Project.toml` and `Manifest.toml` go a long way towards accomplishing this, as you can share both and with the same hardware and Julia version should be able to get the exact same set of dependencies and therefore run the same code. In practice, this level of reproducibility isn't *usually* needed, as most time a set of code can be run accurately without requiring the exact same set of dependencies.

Since dependencies can have variation between systems (Windows/Mac) and architectures (x86 vs x64), you may not be able to recreate the Manifest exactly. Nevertheless, it's a fairly low bar if you are trying to maintain the utmost level of rigor around the toolchain and Julia is one of the most robust languages regarding tools to support open replication of results.

> **Artifacts**
>
> Julia has a system called **artifacts** which allows specification of a location and hash (a cryptographic key) for data and binaries. The artifact system used to download and verify the contents of a file match the hash. This is designed for more permanent data and less end-user workflows, but we call it out here as another example where Julia takes steps to promote consistency and reproducibility.
> For more on data workflows for the end-user, see Chapter 11.

> **Tip**
>
> You can configure the environment in which a VSCode Julia REPL opens. Just click the `Julia env: ...` button at the bottom. Note however that the Julia version itself will always be the default one from `juliaup`.

## 20.10. Creating Local packages

Once your code base grows beyond a few scripts, you will want to create a package of your own. The first advantage is that you don't need to specify the path of every file: `using MyPackage` is enough to get access to the names you define and export (or `using MyPackage: myfunc1, myfunc2` to use bring non-exported functions into your environment). Furthermore, by structuring your project as a Pacakge, you can specify versions for your package and its dependencies, making your code easier and safer to reuse.

To create a new package locally, the easy way is to use `]generate` (we will discuss a more sophisticated workflow in the next blog post).

```
Pkg.generate(sitepath("MyPackage")); # ignore sitepath
```

This command initializes a simple folder with a `Project.toml` and a `src` subfolder. As we have seen, the `Project.toml` specifies the dependencies. Meanwhile, the `src` subfolder contains a file `MyPackage.jl`, where a module called `MyPackage` is defined. It is the heart of your package, and will typically look like this when you're done:

```
module MyPackage
```

```
# imported dependencies
using OtherPackage1
using OtherPackage2

# files defining functions, types, etc.
include("file1.jl")
include("subfolder/file2.jl")

# names you want to make public
export myfunc # e.g. defined in `file1.jl`
export MyType

end
```

### 20.10.1. PkgTemplates.jl

PkgTemplates.jl is like `]generate` from Pkg.jl but provides a number of options to pre-configure the repository for things such as continuous integration, testing, and compatibiltiy. If you are not yet making use of that more advanced functionality, the `]generate` method will work just fine for you.

This will walk you through an interactive prompt to create a package in the desired folder. `~/.julia/dev` is a suggested location, but technically any folder will make do:

```
using PkgTemplates
cd("~/.julia/dev")
Template(interactive=true)("MyPkg")
```

## 20.11. Development workflow

Once you have created a package, your development routine might look like this:

1. Open a REPL in which you import `MyPackage`
2. Run some functions interactively, either by writing them directly in the REPL or from a Julia file that you use as a notebook
3. Modify some files in `MyPackage`
4. Go back to step 2

For that to work well, you need code modifications to be taken into account automatically. That is why Revise.jl exists. If you start every REPL session by explicitly Revise.jl (`using Revise`), then all the other packages you import after that will have their code tracked. Whenever you edit a source file and hit save, the REPL will update its state accordingly. To automatically do this for every session, see Section 20.12.

> **Note**
>
> The Julia extension imports Revise.jl by default when it starts a REPL.

This is how you get started using your own package once it's set up:

```
using Revise, Pkg
Pkg.activate("./MyPackage")
using MyPackage
myfunc() # defined and exported in MyPackage
MyPackage.myfunc2() # defined and *not* exported in MyPackage
```

> **Note**
>
> If you are working on a set of interrelated packages, you may need to tell those packages to use the *development* version of the package which you are modifying, instead of using the latest available from a registry. For example, say you are working on revisions to PkgA in the following dependency tree:
>
> ```
> PkgB -- depends on -- > PkgA
> ```
>
> If you are modifying PkgA, then you might need to tell PkgB to use the development version. For this, then you would need to:
>
> 1. Create an outer environment where you want to run the packages for interactive use while developing (say `activate @mydevenv`).
> 2. `]dev PkgB` which will download the associated repository into `~./julia/dev/PkgB`
> 3. Go into the environment `~/.julia/dev/PkgB` and tell that environment to use the development version of PkgA with `]dev PkgB` (assuming you are modifying PkgA also in the `~/.julia/dev/` folder)
>
> Now, in the `@mydevenv` environemnt, when you load `PkgB` it will load the version of `PkgA`

## 20.12. Configuration

Julia accepts startup flags to handle settings such as the number of threads available or the environment in which it launches. In addition, most Julia developers also have a startup file which is run automatically every time the language is started. It is located at `.julia/config/startup.jl`.

The basic component that everyone puts in the startup file is Revise.jl. Users also commonly import packages that affect the REPL experience, as well as esthetic, benchmarking or profiling utilities. A typical example is OhMyREPL.jl which is widely used for syntax highlighting in the REPL. While other packages are often used, we suggest the following as a minimum:

```
# save as a file in /.julia/config/startup.jl
try
    using Revise
    using OhMyREPL
catch e
    @warn "Error with startup packages"
end
```

More generally, the startup file allows you to define your own favorite helper functions and have them immediately available in every Julia session. StartupCustomizer.jl can help you set up your startup file.

> **Tip**
>
> Here are a few more startup packages that can make your life easier once you know the language better:
>
> - AbbreviatedStackTraces.jl allows you to shorten error stacktraces, which can sometimes get pretty long (beware of its interactions with VSCode)
> - Term.jl offers a completely new way to display things like types and errors (see the advanced configuration to enable it by default).

## 20.13. Interactivity

The Julia REPL comes bundled with InteractiveUtils.jl, a bunch of very useful functions for interacting with source code.

Here are a few examples:

```
using InteractiveUtils # not necessary in a REPL session
supertypes(Int64)
```

```
(Int64, Signed, Integer, Real, Number, Any)
```

```
subtypes(Integer)
```

```
4-element Vector{Any}:
 Bool
 GeometryBasics.OffsetInteger
 Signed
 Unsigned
```

```
methodswith(Integer)[1:5] # first five methods that take an integer argument
```

```
[1] Array(s::LinearAlgebra.UniformScaling, m::Integer, n::Integer) @ LinearAlgebra ~/.julia/juli
[2] Float16(x::Integer) @ Base float.jl:154
[3] Integer(x::Integer) @ Core boot.jl:817
[4] Ref(x::Ptr{T}, i::Integer) where T @ Base refpointer.jl:142
[5] Ref(x::Ref, i::Integer) @ Base refpointer.jl:141
```

```
@which exp(1) # where the currently used function is defined
```

```
exp(x::Real)
     @ Base.Math math.jl:1575
```

```
apropos("matrix exponential") # search docstrings
```

```
Base.:^
Base.exp
```

When you ask for help on a Julia forum, you might want to include your local Julia information:

```
versioninfo()
```

```
Julia Version 1.10.5
Commit 6f3fdf7b362 (2024-08-27 14:19 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: 4 default, 0 interactive, 2 GC (on 4 virtual cores)
Environment:
  JULIA_NUM_THREADS = auto
```

**Tip**

The following packages can give you even more interactive power:

- InteractiveCodeSearch.jl to look for a precise implementation of a function.
- InteractiveErrors.jl to navigate through stacktraces.
- CodeTracking.jl to extend InteractiveUtils.jl

# 21. Troubleshooting Julia Code

> "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." - Brian Kernighan

## 21.1. In this Chapter

Debugging in Julia involves a mix of strategies, including using print statements, the Debugger package for step-by-step inspection, logging with the Logging module, and interactive debugging with Infiltrator. These tools and techniques can help you identify and fix issues in our code efficiently.

## 21.2. Error Messages and Stack Traces

Julia's error messages and stack traces can be quite informative. When an error occurs, Julia provides a traceback that shows the function call stack leading to the error, which helps in identifying where things went wrong.

```julia
#| error: true
function mysqrt(x)
    return sqrt(x)
end

mysqrt(-1)  # This will raise a `DomainError`
```

The **stacktrace** will show us the sequence of function calls that led to the error. The print out will show the list of functions that were called (the **callstack**) which led to the code that errored. Additionally, help text is often printed, potentially offering some advice for resolving the issue. When you encounter errors in an interactive session, you can click on different parts of the stacktrace and be taken to the associated code in your editor.

### 21.2.1. Error Types

Notice that errors are given specific types and not just result in a generic `Error`. This aids in understanding for the user: if a `DomainError` then you know that you passed the right type (e.g. a `Float64` to a function that takes a number), just that the value was not acceptable (as in the example above). Constrast that with a `MethodError` which will tell you that you've passed an invalid kind of thing to the function, not just that it's value was off:

```
#| error: true
mysqrt("a string isn't OK")
```

## 21.3. Logging

When you encounter a problem in your code or want to track progress, a common reflex is to add `print` statements everywhere.

```
function printing_func(n)
    for i in 1:n
        println(i^2)
    end
end
```

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

```
printing_func(3)
```

```
1
4
9
```

A slight improvement is given by the `@show` macro, which displays the variable name:

```
function showing_func(n)
    for i in 1:n
        @show i^2
    end
end
```

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

```
showing_func(3)
```

```
i ^ 2 = 1
i ^ 2 = 4
i ^ 2 = 9
```

But you can go even further with the macros `@debug`, `@info`, `@warn` and `@error`. They have several advantages over printing:

- They display variable names and a custom message
- They show the line number they were called from
- They can be disabled and filtered according to source module and severity level
- They work well in multithreaded code
- They can write their output to a file

```julia
function warning_func(n)
    for i in 1:n
        @warn "This is bad" i^2
    end
end
```

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

```julia
warning_func(3)
```

```
┌ Warning: This is bad
│   i ^ 2 = 1
└ @ Main In[6]:3
┌ Warning: This is bad
│   i ^ 2 = 4
└ @ Main In[6]:3
┌ Warning: This is bad
│   i ^ 2 = 9
└ @ Main In[6]:3
```

Refer to the logging documentation for more information.

> **Note**
>
> In particular, note that `@debug` messages are suppressed by default. You can enable them through the `JULIA_DEBUG` environment variable if you specify the source module name, typically `Main` or your package module.

Beyond the built-in logging utilities, ProgressLogging.jl has a macro `@progress`, which interfaces nicely with VSCode and Pluto to display progress bars. And Suppressor.jl can sometimes be handy when you need to suppress warnings or other bothersome messages (use at your own risk).

## 21.4. Commonly Encountered Macros

Aside from those mentioned in the context of Logging, there are a number of different useful macros, many of which are highlighted in the following table:

Table 21.1.: Useful macros for modeling work. There are others related to parallelism which will be covered in Chapter 10.

| Macro | Description |
|---|---|
| `BenchmarkTools.@benchmark` | Runs the given expression multiple times, collecting timing and memory allocation statistics. Useful for benchmarking and performance analysis. |
| `BenchmarkTools.@btime` | Similar to `@benchmark`, but focuses on the minimum execution time and provides a more concise output. |
| `@edit` | Opens the source code of a function or module in an editor for inspection or modification. |
| `@which` | Displays the method that would be called for a given function call, helping to understand method dispatch. |
| `@code_warntype` | Shows the type inference results for a given function call, highlighting any type instabilities or performance issues. |
| `@info`, `@warn`, `@error` | Used for logging messages at different severity levels (info, warning, error) during program execution. |
| `@assert` | Asserts that a given condition is true, throwing an error if the condition is false. Useful for runtime checks and debugging. |
| `@view`, `@views` | Access a subset of an array without copying the data in that slice. `@views` applies to all array slicing operations within the expressions that follow it. |
| `Test.@test`, `Test.@testset` | Used for defining unit tests. `@test` checks that a condition is true, while `@testset` groups related tests together. |
| `@raw` | Encloses a string literal, disabling string interpolation and escape sequences. Useful for writing raw string data. This is especially helpful when working with filepaths where the \ in Windows paths otherwise needs to be escaped with a leading slash (e.g. \\ ). |
| `@fastmath` | Enables aggressive floating-point optimizations within a block, potentially sacrificing strict IEEE compliance for performance. |

| Macro | Description |
|---|---|
| `@inbounds` | Disables bounds checking for array accesses within a block, improving performance but removing safety checks. |
| `@inline` | Suggests to the compiler that a function should be inlined at its call sites, potentially improving performance by reducing function call overhead. |

## 21.5. Debugging

The limitation of printing or logging is that you cannot interact with local variables or save them for further analysis. The following two packages solve this issue (consider adding to your default environment `@v1.X`, like Revise.jl).

### 21.5.1. Setting

Assume you want to debug a function checking whether the $n$-th Fermat number $F_n = 2^{2^n} + 1$ is prime:

```
function fermat_prime(n)
    k = 2^n
    F = 2^k + 1
    for d in 2:isqrt(F)  # integer square root
        if F % d == 0
            return false
        end
    end
    return true
end
```

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

```
fermat_prime(4), fermat_prime(6)
```

```
(true, true)
```

Unfortunately, $F_4 = 65537$ is the largest known Fermat prime, which means $F_6$ is incorrectly classified. Let's investigate why this happens!

### 21.5.2. Infiltrator.jl

Infiltrator.jl is a lightweight inspection package, which will not slow down your code at all. Its `@infiltrate` macro allows you to directly set breakpoints in your code. Calling a function which hits a breakpoint will activate the Infiltrator REPL-mode and change the prompt to `infil>`. Typing `?` in this mode will summarize available commands. For example, typing `@locals` in Infiltrator-mode will print local variables:

```
using Infiltrator

function fermat_prime_infil(n)
    k = 2^n
    F = 2^k + 1
    @infiltrate
    for d in 2:isqrt(F)
        if F % d == 0
            return false
        end
    end
    return true
end
```

What makes Infiltrator.jl even more powerful is the `@exfiltrate` macro, which allows you to move local variables into a global storage called the `safehouse`.

```
julia> fermat_prime_infil(6)
Infiltrating fermat_prime_infil(n::Int64)
  at REPL[2]:4

infil> @exfiltrate k F
Exfiltrating 2 local variables into the safehouse.

infil> @continue

true

julia> safehouse.k
64

julia> safehouse.F
1
```

The diagnosis is a classic one: integer overflow. Indeed, $2^{64}$ is larger than the maximum integer value in Julia:

```
typemax(Int)
2^63-1
```

And the solution is to call our function on "big" integers with an arbitrary number of bits:

```
fermat_prime(big(6))
```

### 21.5.3. Debugger.jl

Debugger.jl allows us to interrupt code execution anywhere we want, even in functions we did not write. Using its `@enter` macro, we can enter a function call and walk through the call stack, at the cost of reduced performance.

The REPL prompt changes to `1|debug>`, allowing you to use custom navigation commands to step into and out of function calls, show local variables and set breakpoints. Typing a backtick `` ` `` will change the prompt to `1|julia>`, indicating evaluation mode. Any expression typed in this mode will be evaluated in the local context. This is useful to show local variables, as demonstrated in the following example:

```
julia> using Debugger

julia> @enter fermat_prime(6)
In fermat_prime(n) at REPL[7]:1
 1  function fermat_prime(n)
>2      k = 2^n
 3      F = 2^k + 1
 4      for d in 2:isqrt(F)  # integer square root
 5          if F % d == 0
 6              return false

About to run: (^)(2, 6)
1|debug> n
In fermat_prime(n) at REPL[7]:1
 1  function fermat_prime(n)
 2      k = 2^n
>3      F = 2^k + 1
 4      for d in 2:isqrt(F)  # integer square root
 5          if F % d == 0
 6              return false
 7          end

About to run: (^)(2, 64)
```

```
1|julia> k
64
```

> Tip
>
> VSCode offers a nice graphical interface for debugging. Click left of a line number in an editor pane to add a *breakpoint*, which is represented by a red circle. In the debugging pane of the Julia extension, click `Run and Debug` to start the debugger. The program will automatically halt when it hits a breakpoint. Using the toolbar at the top of the editor, you can then *continue*, *step over*, *step into* and *step out* of your code. The debugger will open a pane showing information about the code such as local variables inside of the current function, their current values and the full call stack.
> The debugger can be sped up by selectively compiling modules that you will not need to step into via the `+` symbol at the bottom of the debugging pane. It is often easiest to start by adding `ALL_MODULES_EXCEPT_MAIN` to the compiled list, and then selectively remove the modules you need to have interpreted.

# 22. Distributing and Sharing Julia Code

## 22.1. In this Chapter

Applying software engineering best practices (Chapter 11) in Julia, including testing, documentation, and coverage metrics. Collaborating on code. Publishing packages for others to use.

## 22.2. Setup

A vast majority of Julia packages are hosted on GitHub (although less common, other options like GitLab are also possible). GitHub is a platform for collaborative software development, based on the version control system Git (see Chapter 11 for an introduction).

The first step is therefore creating an empty GitHub repository on Github (don't add a README License, etc. at this step).

> **Tip**
>
> You should try to follow package naming guidelines and add a ".jl" extension at the end, like so: "MyAwesomePackage.jl".

Locally, use PkgTemplates.jl (see Section 20.10.1) to then create the package's folder locally on your computer, which will create a package with several subfolders (these will be described as the chapter progresses).

To sync this up with the newly created Github repositrory, you git push this new folder to the remote repository https://github.com/myuser/MyAwesomePackage.jl (Github should show you how to do this on the associated repository page).

## 22.3. GitHub Actions

The most useful aspect of PkgTemplates.jl is that it automatically generates workflows for GitHub Actions. These are stored as YAML files in `.github/workflows`, with a slightly convoluted syntax that you don't need to fully understand. For instance, the file `CI.yml` contains instructions that execute the tests of your package (see below) for each pull request, tag or push to the `main` branch. This is done on a GitHub server and should theoretically cost you money, but if your GitHub repository is public, you get an unlimited workflow budget for free.

A variety of workflows and functionalities are available through optional plugins. The interactive setting `Template(..., interactive=true)` allows you to select the ones you want for a given package. Otherwise, you will get the default selection, which you are encouraged to look at.

## 22.4. Testing

The purpose of the `test` subfolder in your package is unit testing: automatically checking that your code behaves the way you want it to. For instance, if you write your own square root function, you may want to test that it gives the correct results for positive numbers, and errors for negative numbers.

```
using Test

@test sqrt(4) ≈ 2

@testset "Invalid inputs" begin
    @test_throws DomainError sqrt(-1)
    @test_throws MethodError sqrt("abc")
end;
```

Such tests belong in `test/runtests.jl`, and they are executed with the `]test` command (in the REPL's Pkg mode). Unit testing may seem rather naive, or even superfluous, but as your code grows more complex, it becomes easier to break something without noticing. Testing each part separately will increase the reliability of the software you write.

> **Tip**
>
> To test the arguments provided to the functions within your code (for instance their sign or value), avoid `@assert` (which can be deactivated) and use ArgCheck.jl instead.
> That is, avoid this:
>
> ```
> function mysqrt(x)
>     @assert x >= 0
> ```

```
    ...
```

And do this instead:

```
function mysqrt(x)
    @argcheck x < 0 DomainError
    ...
end
```

At some point, your package may require test-specific dependencies. In essence, you give the `test` subfolder it's own environment and `Project.toml` file. This often happens when you need to test compatibility with another package, on which you do not depend for the source code itself. Or it may simply be due to testing-specific packages like the ones we will encounter below. For interactive testing work, use TestEnv.jl to activate the full test environment (faster than running `]test` repeatedly).

> **Tip**
>
> The Julia extension also offers a more advanced own testing framework, which relies on defining "test items" the code. The benefit of this is that the tests will integrate more directly with the VS Code interface and specific subgroups of tests can be run independently, on-demand. See TestItemRunner.jl for more.

> **Tip**
>
> If you want to have more control over your tests, you can try
>
> - ReferenceTests.jl to compare function outputs with reference files.
> - ReTest.jl to define tests next to the source code and control their execution.
> - TestSetExtensions.jl to make test set outputs more readable.
> - TestReadme.jl to test whatever code samples are in your README.
> - ReTestItems.jl for an alternative take on VSCode's test item framework.

### 22.4.1. Code Coverage

Code coverage refers to the fraction of lines in your source code that are covered by tests (described in more detail in Section 11.3.2). Codecov is a website that provides easy visualization of this coverage, and many Julia packages use it. It is available as a PkgTemplates.jl plugin, but you have to perform an additional configuration step on the repo for Codecov to communicate with it.

## 22.5. Code Style

To make your code easy to read, it is recommended to follow a consistent set of guidelines. The official style guide is very short, so most people use third party style guides like BlueStyle or SciMLStyle.

JuliaFormatter.jl is an automated formatter for Julia files which can help you enforce the style guide of your choice. Just add a file `.JuliaFormatter.toml` at the root of your repository, containing a single line like

```
style = "blue"
```

Then, the package directory will be formatted in the BlueStyle whenever you call

```
using JuliaFormatter
JuliaFormatter.format(MyAwesomePackage)
```

**Note**

The default formatter uses JuliaFormatter.jl.

**Tip**

You can format code automatically in GitHub pull requests with the `julia-format` action, or add the formatting check directly to your test suite.

## 22.6. Code quality

Of course, there is more to code quality than just formatting. Aqua.jl (Auto QUality Assurance) provides a set of routines that examine other aspects of your package, from ensuring that there are no unused dependencies to catching ambiguous methods statically.

Include the following in your tests to have Aqua.jl run various checks against your code each time tests get run.:

```
using Aqua, MyAwesomePackage
Aqua.test_all(MyAwesomePackage)
```

JET.jl is tool that is similar to a static linter in other languages. This means that it can inspect your code and 'understand' it well enough to catch many types of errors before runtime. It does this by running type inference and figuring out how a given type will flow through the call stack of methods.

You can either use it in report mode (with a nice VSCode display) or in test mode as follows:

```
using JET, MyAwesomePackage
JET.report_package(MyAwesomePackage)
JET.test_package(MyAwesomePackage)
```

Note that both Aqua.jl and JET.jl might pick up false positives: refer to their respective documentations for ways to make them less sensitive.

Finally, ExplicitImports.jl can help you get rid of generic imports to specify where each of the variables in your package comes from. As a project gets more complex, `using SomePackage` can bring many, sometimes conflicting symbols into your current namespace. ExplicitImports forces you to either qualify the usage (e.g. `SomePackage.somefunction(...)`) or explicitly opt into importing certain variables.

## 22.7. Documentation

Refer to for more detail on documentation and it's importance in Section 11.4.2. Here are some additional workflow tips for setting up documentation for your package.

DocStringExtensions.jl provides a few shortcuts that can speed up docstring creation by taking care of the obvious parts.

In addition to docstrings, Documenter.jl allows you to design a website for all of this, based on Markdown files contained in the `docs` subfolder of your package. Unsurprisingly, its own documentation is excellent and will teach you a lot. To build the documentation locally, just run

```
julia> using Pkg

julia> Pkg.activate("docs")

julia> include("docs/make.jl")
```

Then, use LiveServer.jl from your package folder to visualize and automatically update the website as the code changes (similar to Revise.jl, but for your docpages instead of your code):

```
julia> using LiveServer

julia> servedocs()
```

To host the documentation online easily, just select the `Documenter` plugin from PkgTemplates.jl during creation. Not only will this fill the `docs` subfolder with the appropriate starting files: it will also initialize a GitHub Actions workflow to build and deploy your website on GitHub pages. Lastly, select the `gh-pages` branch as source in the Github settings for your repository.

## 22.8. Literate programming

Literate programming is so-called for combining written documents with the output of prgorams (literature + code = literate programming). These tools allow you to interleave code with texts, formulas, images and so on.

In addition to the Pluto.jl and Jupyter notebooks, take a look at Literate.jl to enrich your code with comments and translate it to various formats. Books.jl is relevant to draft long documents in a pure Julia way.

Quarto is an open-source scientific and technical publishing system that supports Python, R and Julia. Quarto can render markdown files (`.md`), Quarto markdown files (`.qmd`), and Jupyter Notebooks (`.ipynb`) into documents (Word, PDF, presentations), web pages, blog posts, books, and more. Additionally, Quarto makes it easy to share or publish rendered content to various online hosts.

PPTX.jl will create Microsoft PowerPoint files.

## 22.9. Versions and registration

### 22.9.1. Versions and Compatibility

The Julia community has adopted semantic versioning, which means every package must have a version, and the version numbering follows strict rules (the concept of versioning was covered in Section 11.6.2).

To comply with the versioning requirements in Pkg's resolver, you need to specify compatibility bounds for your dependencies: this happens in the `[compat]` section of your `Project.toml`. To initialize these bounds with current dependecy versions, use the `]compat` command in the Pkg mode of the REPL, or the package PackageCompatUI.jl.

Over time, new versions of your dependencies will be released. The CompatHelper.jl GitHub Action will help you monitor upstream Julia dependencies and suggest changes to your `Project.toml`'s `[compat]` section accordingly. In addition, Dependabot can monitor the dependencies... of your GitHub actions themselves. Both of these can are included in the default PkgTemplates setup.

> **Tip**
>
> It may also happen that you incorrectly promise compatibility with an old version of a package and not realize it (since Pkg prefers newer versions within the compatibility bounds, not all combinations get tested). To prevent that, the julia-downgrade-compat GitHub action tests your package with the oldest possible version of every dependency, and verifies that everything still works.

### 22.9.2. Registration

If your package is useful to others in the community, it may be a good idea to register it, that is, make it part of the pool of packages that can be installed with

```
pkg> add MyAwesomePackage  # made possible by registration
```

Note that unregistered packages can also be installed by anyone from the GitHub URL, but this a less reproducible solution:

```
pkg> add https://github.com/myuser/MyAwesomePackage  # not ideal
```

To register your package, check out the general registry guidelines. The Registrator.jl bot can help you automate the process. Another handy bot, provided by default with PkgTemplates.jl, is TagBot: it automatically tags new versions of your package following each registry release. If you have performed the necessary SSH configuration, TagBot will also trigger documentation website builds following each release.

#### 22.9.2.1. Local Registry

For distributing privately (or publicly if you make the repository public), LocalRegistry.jl provides convenience functions for creating a new registry, adding new packages, and updating versions for the packages. If you want to share packages internally, create and register packages to a repository that's hosted somewhere you and your team can access. If you wanted to make the repository public, you can publish the registry repository somewhere publically accessable (such as a public GitHub repository).

Once established, other users can add a repository as easily as entering package mode and running `registry add`. Say that we have already put a registry we called `FinancePackages` in a repository on the company intranet:

```
pkg> registry add http://company-intranet.com/git/FinancePackages.git
```

#### 22.9.2.2. Hosted Registries

Alternatively to a self-hosted local registry, third party services such as JuliaHub provide managed registries well suited for corporate environments.

## 22.10. Reproducibility

Obtaining consistent and reproducible results is an essential part of model auditing and compliance. One tool to consider is DrWatson.jl. It is a general toolbox for running and re-running models in an orderly fashion.

Some specific issues come up in attempting to ensure reporducibility

A first hurdle is random number generation, which is not guaranteed to remain stable across Julia versions. To ensure that the random streams remain exactly the same, you need to use StableRNGs.jl. The downside to this is that the random number generation will be considerablly slower than the usual generator.

Another aspect is dataset download and management. The packages DataDeps.jl, DataToolkit.jl and ArtifactUtils.jl can help you bundle non-code elements with your package (some of these rely on artifacts - discussed in Section 11.6.3).

## 22.11. Interoperability

To ensure compatibility with earlier Julia versions, Compat.jl is your best ally.

Making packages play nice with one another is a key goal of the Julia ecosystem. Since Julia 1.9, this can be done with package extensions, which override specific behaviors based on the presence of a given package in the environment. For example, if you want to provide pre-configured plotting, but don't in general need to include a plotting library as part of your package for all users and use cases. PackageExtensionTools.jl eases of setting up extensions for your package.

Furthermore, the Julia ecosystem as a whole plays nice with other programming languages too. C and Fortran are natively supported. Python can be easily interfaced with the combination of CondaPkg.jl and PythonCall.jl. Other language compatibility packages can be found in the JuliaInterop organization, like RCall.jl.

## 22.12. Customization

Part of interoperability is also flexibility and customization: the Preferences.jl package gives a nice way to specify various options in TOML files. These customizable preferences persist across sessions and provide the preferences at both compile and runtime. For example, say different parts of a company had different preferred data sources but otherwise used the same code. This could be set in a way via Preferences.jl so that each team can share the logic while seamlessly defaulting to different data sources.

## 22.13. Collaboration

Once your package grows big enough, you might need to bring in some help. Working together on a software project has its own set of challenges, which are partially addressed by a good set of ground rules liks SciML ColPrac. Of course, collaboration goes both ways: if you find a Julia package you really like, you are more than welcome to contribute as well, for example by opening issues or submitting pull requests.

# 23. Optimizing Julia Code

The two fundamental principles for writing fast Julia code:

1. Ensure that **the compiler can infer the type** of every variable.
2. Avoid **unnecessary (heap) allocations**.

## 23.1. Type Inference

The compiler's job is to optimize and translate Julia code it into runnable machine code. If a variable's type cannot be deduced before the code is run, then the compiler won't generate efficient code to handle that variable. This phenomenon is called "type instability". Enabling type inference means making sure that every variable's type in every function can be deduced from the types of the function inputs alone.

That is, the compiler will be able to create more optimized code if it can analyze the function you've written and determine what type will be returned. In the following function, the compiler can analyze the expressions and determine with certainty that if `mysum` is given two `Int`s as arguments, the return value will also be an `Int`.

```
function mysum(a,b)
    a + b
end
```

## 23.2. Avoiding Heap Allocations

A "heap allocation" (or simply "allocation") occurs when we create a new variable without knowing how much space it will require (like a `Vector` with flexible length). This has two implications:

1. Allocating memory on the heap takes substantially more time than stack allocated memory to be created.
2. Periodically, a **garbage collector** (GC), needs to run to de-allocate (free up) memory on the heap which is no longer used by the program

Execution of code is stopped while the garbage collector runs, so minimising its usage is important.

The vast majority of performance tips come down to these two fundamental ideas.

Typically, the most common beginner pitfall is the use of untyped global variables without passing them as arguments. Why is it bad? Because the type of a global variable can change outside of the body of a function, so it causes type instabilities wherever it is used. Those type instabilities in turn lead to more heap allocations.

Much more detail on performance considerations is covered in Chapter 9.

## 23.3. Measuring performance

The simplest way to measure how fast a piece of code runs is to use the `@time` macro, which returns the result of the code and prints the measured runtime and allocations. Because code needs to be compiled before it can be run, you should first run a function without timing it so it can be compiled, and then time it:

```
sum_abs(vec) = sum(abs(x) for x in vec);
v = rand(100);

using BenchmarkTools
@time sum_abs(v); # Inaccurate, note the >99% compilation time
@time sum_abs(v); # Accurate
```

```
  0.009408 seconds (43.81 k allocations: 2.896 MiB, 99.91% compilation time)
  0.000005 seconds (1 allocation: 16 bytes)
```

Using `@time` is quick but it has flaws, because your function is only measured once. That measurement might have been influenced by other things going on in your computer at the same time. In general, running the same block of code multiple times is a safer measurement method, because it diminishes the probability of only observing an outlier.

### 23.3.1. BenchmarkTools

BenchmarkTools.jl is the most popular package for repeated measurements on function executions. Similarly to `@time`, BenchmarkTools offers `@btime` which can be used in exactly the same way but will run the code multiple times and provide an average. Additionally, by using `$` to interpolate external values, you remove the overhead caused by global variables.

```
using BenchmarkTools
@btime sum_abs(v);
@btime sum_abs($v);

  42.717 ns (1 allocation: 16 bytes)
  29.523 ns (0 allocations: 0 bytes)
```

In more complex settings, you might need to construct variables in a setup phase that is run before each sample. This can be useful to generate a new random input every time, instead of always using the same input.

```
my_matmul(A, b) = A * b;
@btime my_matmul(A, b) setup = (
    A = rand(1000, 1000); # use semi-colons between setup lines
    b = rand(1000)
);

  154.709 μs (1 allocation: 8.00 KiB)
```

For better visualization, the `@benchmark` macro shows performance histograms:

> **Note**
>
> Certain computations may be optimized away by the compiler before the benchmark takes place. If you observe suspiciously fast performance, especially below the nanosecond scale, this is very likely to have happened.

### 23.3.2. Other tools

BenchmarkTools.jl works fine for relatively short and simple blocks of code (microbenchmarking). To find bottlenecks in a larger program, you should rather use a profiler(see next section) or the package TimerOutputs.jl. It allows you to label different sections of your code, then time them and display a table of grouped by label.

Finally, if you know a loop is slow and you'll need to wait for it to be done, you can use ProgressMeter.jl or ProgressLogging.jl to track its progress. This will display a progress bar in VS Code or in a notebook, indicating how far along a loop has progressed.

## 23.4. Profiling

Profiling can identify performance bottlenecks at function level, and graphical tools such as ProfileView.jl are the best way to use it.

### 23.4.1. Sampling

Whereas a benchmark measures the overall performance of some code, a profiler breaks it down function by function to identify bottlenecks. Sampling-based profilers periodically ask the program which line it is currently executing, and aggregate results by line or by function. Julia offers two kinds: one for runtime (in the module `Profile`) and one for memory (in the submodule `Profile.Allocs`).

These built-in profilers print textual outputs, but the result of profiling is best visualized as a flame graph. In a flame graph, each horizontal layer corresponds to a specific level in the call stack, and the width of a tile shows how much time was spent in the corresponding function. Here's an example:

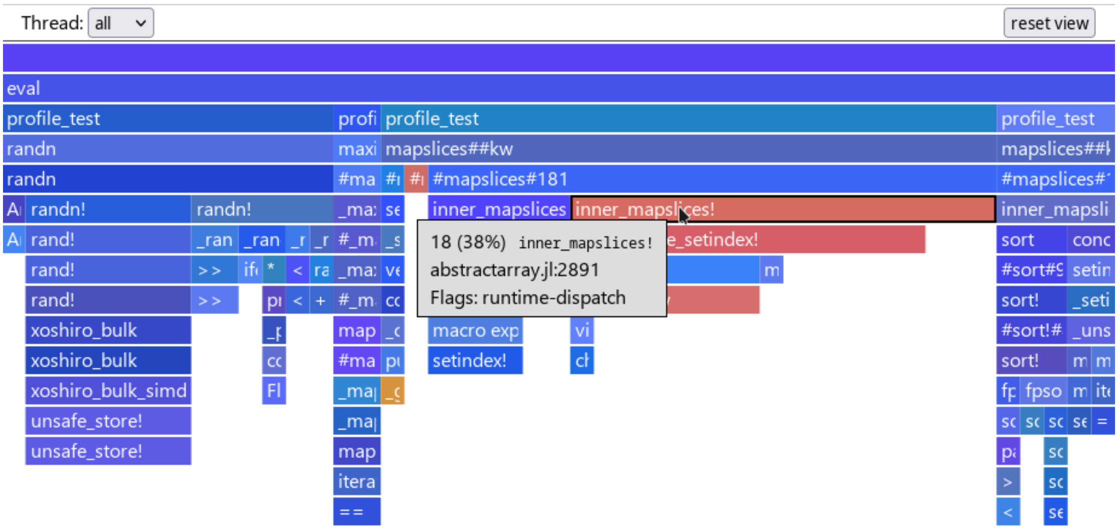



Figure 23.1.: flamegraph

### 23.4.2. Visualization Profile Results

The packages ProfileView.jl and PProf.jl both allow users to record and interact with flame graphs. ProfileView.jl is simpler to use, but PProf is more featureful and is based on pprof, an external tool maintained by Google which applies to more than just Julia code. Here we only demonstrate the former:

```
julia profileview-example using ProfileView @profview do_work(some_input)
```

> **Tip**
>
> Calling `@profview do_work(some_input)` in the integrated Julia REPL will open an interactive flame graph, similar to ProfileView.jl but without requiring a separate package.

To integrate profile visualisations into environments like Jupyter and Pluto, use ProfileSVG.jl or ProfileCanvas.jl, whose outputs can be embedded into a notebook.

No matter which tool you use, if your code is too fast to collect samples, you may need to run it multiple times in a loop.

> **Tip**
>
> To visualize memory allocation profiles, use PProf.jl or VSCode's `@profview_allocs`. A known issue with the allocation profiler is that it is not able to determine the type of every object allocated, instead `Profile.Allocs.UnknownType` is shown instead. Inspecting the call graph can help identify which types are responsible for the allocations.

### 23.4.3. External profilers

Apart from the built-in `Profile` standard library, there are a few external profilers that you can use including Intel VTune (in combination with IntelITT.jl), NVIDIA Nsight Systems (in combination with NVTX.jl), and Tracy.

## 23.5. Type stability

For a section of code to be considered type stable, the type inferred by the compiler must be "concrete", which means that the size of memory that needs to be allocated to store its value is known at compile time. Types declared abstract with `abstract type` are not concrete and neither are parametric types whose parameters are not specified:

```julia
@show isconcretetype(Any)
@show isconcretetype(AbstractVector)
@show isconcretetype(Vector) # Shorthand for `Vector{T} where T`
@show isconcretetype(Vector{Real})
@show isconcretetype(eltype(Vector{Real}))
@show isconcretetype(Vector{Int64})
```

```
isconcretetype(Any) = false
isconcretetype(AbstractVector) = false
isconcretetype(Vector) = false
isconcretetype(Vector{Real}) = true
isconcretetype(eltype(Vector{Real})) = false
isconcretetype(Vector{Int64}) = true
```

```
true
```

> **Note**
>
> `Vector{Real}` is concrete despite `Real` being abstract for subtle typing reasons but it will still be slow in practice because the type of its elements is abstract.

While type-stable function calls compile down to fast `GOTO` statements, type-unstable function calls generate code that must read the list of all methods for a given operation and find the one that matches. This phenomenon called "dynamic dispatch" prevents further optimizations.

Type-stability is a contagous thing: if a variable's type cannot be inferred, then the types of variables that depend on it may not be inferrable either. Most code should be type-stable unless it has a good reason not to be.

### 23.5.1. Detecting instabilities

The simplest way to detect an instability is with the builtin macro `@code_warntype`: The output of `@code_warntype` is difficult to parse, but the key takeaway is the return type of the function's `Body`: if it is an abstract type, like `Any`, something is wrong. In a normal Julia REPL, such cases would show up colored in red as a warning.

```julia
using InteractiveUtils # loaded automatically if using a notebook or REPL
function put_in_vec_and_sum(x)
    v = []
    push!(v, x)
    return sum(v)
end;

@code_warntype put_in_vec_and_sum(1)
```

```
MethodInstance for put_in_vec_and_sum(::Int64)
  from put_in_vec_and_sum(x) @ Main In[7]:2
Arguments
```

```
  #self#::Core.Const(put_in_vec_and_sum)
  x::Int64
Locals
  v::Vector{Any}
Body::Any
1 ─      (v = Base.vect())
│        Main.push!(v, x)
│   %3 = Main.sum(v)::Any
└──      return %3
```

Unfortunately, `@code_warntype` is limited to one function body: calls to other functions are not expanded, which makes deeper type instabilities easy to miss. That is where JET.jl can help: it provides optimization analysis aimed primarily at finding type instabilities. While test integrations are also provided, the interactive entry point of JET is the `@report_opt` macro.

```
using JET
@report_opt put_in_vec_and_sum(1)
```

```
═════ 6 possible errors found ═════
┌ put_in_vec_and_sum(x::Int64) @ Main ./In[7]:5
│┌ sum(a::Vector{Any}) @ Base ./reducedim.jl:1010
││┌ sum(a::Vector{Any}; dims::Colon, kw::@Kwargs{}) @ Base ./reducedim.jl:1010
│││┌ _sum(a::Vector{Any}, ::Colon) @ Base ./reducedim.jl:1014
││││┌ _sum(a::Vector{Any}, ::Colon; kw::@Kwargs{}) @ Base ./reducedim.jl:1014
│││││┌ _sum(f::typeof(identity), a::Vector{Any}, ::Colon) @ Base ./reducedim.jl:1015
││││││┌ _sum(f::typeof(identity), a::Vector{Any}, ::Colon; kw::@Kwargs{}) @ Base ./reducedim.jl:
│││││││┌ mapreduce(f::typeof(identity), op::typeof(Base.add_sum), A::Vector{Any}) @ Base ./reduc
││││││││┌ mapreduce(f::typeof(identity), op::typeof(Base.add_sum), A::Vector{Any}; dims::Colon,
│││││││││┌ _mapreduce_dim(f::typeof(identity), op::typeof(Base.add_sum), ::Base._InitialValue,
││││││││││┌ _mapreduce(f::typeof(identity), op::typeof(Base.add_sum), ::IndexLinear, A::Vector{
│││││││││││┌ mapreduce_impl(f::typeof(identity), op::typeof(Base.add_sum), A::Vector{Any}, ifir
││││││││││││┌ mapreduce_impl(f::typeof(identity), op::typeof(Base.add_sum), A::Vector{…}, ifirs
│││││││││││││ runtime dispatch detected: Base.mapreduce_first(f::typeof(identity), op::typeof(Ba
││││││││││││└────────────────────
││││││││││││┌ mapreduce_impl(f::typeof(identity), op::typeof(Base.add_sum), A::Vector{…}, ifirs
│││││││││││││ runtime dispatch detected: op::typeof(Base.add_sum)(%9::Any, %11::Any)::Any
││││││││││││└────────────────────
││││││││││││┌ mapreduce_impl(f::typeof(identity), op::typeof(Base.add_sum), A::Vector{…}, ifirs
│││││││││││││ runtime dispatch detected: op::typeof(Base.add_sum)(%69::Any, %71::Any)::Any
││││││││││││└────────────────────
```

```
||||||||||┌ _mapreduce(f::typeof(identity), op::typeof(Base.add_sum), ::IndexLinear, A::Vector{
||||||||||│ runtime dispatch detected: Base.mapreduce_first(f::typeof(identity), op::typeof(Base
||||||||||└────────────────────
||||||||||┌ _mapreduce(f::typeof(identity), op::typeof(Base.add_sum), ::IndexLinear, A::Vector{
||||||||||│ runtime dispatch detected: op::typeof(Base.add_sum)(%15::Any, %16::Any)::Any
||||||||||└────────────────────
||||||||||┌ _mapreduce(f::typeof(identity), op::typeof(Base.add_sum), ::IndexLinear, A::Vector{
||||||||||│ runtime dispatch detected: op::typeof(Base.add_sum)(%18::Any, %25::Any)::Any
||||||||||└────────────────────
```

```
[ Info: tracking Base
```

> **Tip**
>
> The Julia extension features a static linter, and runtime diagnostics with JET can be automated to run periodically on your codebase and show any problems detected.

Cthulhu.jl exposes the `@descend` macro which can be used to interactively "step through" lines of the corresponding typed code, and "descend" into a particular line if needed. This is akin to repeatedly calling `@code_warntype` deeper and deeper into your functions ("slowly succumbing to the madness..." of type instability).

### 23.5.2. Fixing instabilities

The Julia manual has a collection of tips to improve type inference.

> **Tip**
>
> To be more forceful about ensuring type stabiltiy in your code, one approach is to error whenever a type instability occurs: the macro `@stable` from DispatchDoctor.jl allows exactly that.

## 23.6. Memory management

After ensuring type stability, one should try to reduce the number of heap allocations a program makes. Again, the Julia manual has a series of tricks related to arrays and allocations which you should take a look at. In particular, try to modify existing arrays instead of allocating new objects and try to access arrays in the right order for Julia, i.e. accessing data down columns instead of across rows.

Alternatively, to ensure that non-allocating functions never regress in future versions of your code, you can write a test set to check allocations by providing the function and a concrete type-signature.

```
julia AllocCheck @testset "non-allocating" begin    @test isempty(AllocCheck.check_allocs(my_fun
(Float64, Float64))) end
```

## 23.7. Compilation

A number of tools allow you to reduce Julia's latency, also referred to as TTFX (time to first X, where X was historically plotting a graph).

### 23.7.1. Precompilation

PrecompileTools.jl reduces the amount of time taken to run functions loaded from a package or local module that you wrote. It allows module authors to specify methods to precompile when a module is loaded for the first time. The methods chosen should represent those which would be used during typical user workflows. These methods then have the same latency as if they had already been run by the end user. This adds upfront pre-compilation time when installing a package version for the first time, but then subsequent uses will be much quicker.

Here's an example of precompilation, adapted from the package's documentation:

```
module MyPackage

using PrecompileTools: @compile_workload

struct MyType
    x::Int
end

myfunction(a::Vector) = a[1].x

@compile_workload begin
    a = [MyType(1)]
    myfunction(a)
end

end
```

Note that every method that is called will be compiled, no matter how far down the call stack or which module it comes from. To see if the intended calls were compiled correctly or diagnose other problems related to precompilation, use SnoopCompile.jl. This is especially important for writers of registered Julia packages, as it allows you to diagnose recompilation that happens due to invalidation.

### 23.7.2. Package compilation

To reduce the time that packages take to load, you can use PackageCompiler.jl to generate a custom version of Julia, called a **sysimage** (system image), with its own custom standard library. As packages in the standard library are already compiled, any `using` or `import` statement involving them is almost instant.

Once PackageCompiler.jl is added to your global environment, activate a local environment for which you want to generate a sysimage, ensure all of the packages you want to compile are in its `Project.toml`, and run `create_sysimage` as in the example below. The filetype of `sysimage_path` differs by operating system: Linux has `.so`, MacOS has `.dylib`, and Windows has `.dll`.

```
using PackageCompiler # installed in global environment
packages_to_compile = ["Makie", "DifferentialEquations"]
create_sysimage(packages_to_compile; sysimage_path="MySysimage.so")
```

Once a sysimage is generated, it can be used with the command line flag: `julia --sysimage=path/to/sysimage`.

> **Tip**
>
> The generation and loading of sysimages can be streamlined with VSCode. By default, the command sequence `Task: Run Build Task` followed by `Julia: Build custom sysimage for current environment` will compile a sysimage containing all packages in the current environment, but additional details can be specified in a `/.vscode/JuliaSysimage.toml` file. To automatically detect and use a custom sysimage, set `useCustomSysimage` to `true` in the application settings.

### 23.7.3. Static compilation

PackageCompiler.jl also facilitates the creation of apps and libraries that can be shared to and run on machines that don't have Julia installed.

At a basic level, all that's required to turn a Julia module `MyModule` into an app is a function `julia_main()::Cint` that returns `0` upon successful completion. Then, with PackageCompiler.jl loaded, run `create_app("MyModule", "MyAppCompiled")`.

Command line arguments to the resulting app are assigned to the global variable `ARGS::Array{ASCIIString}`, the handling of which can be made easier by ArgParse.jl.

In Julia, a library is just a sysimage with some extras that enable external programs to interact with it. Any functions in a module marked with `Base.@ccallable`, and whose type signature involves C-conforming types e.g. `Cint`, `Cstring`, and `Cvoid`, can be compiled into an externally callable library with `create_library`, similarly to `create_app`. Unfortunately, the process of compiling and sharing a standalone executable or callable library must take relocability into account, which is beyond the scope of this blog.

> **Note**
>
> An alternative way to compile a shareable app or library that doesn't need to compile a sysimage, and therefore results in smaller binaries, is to use StaticCompiler.jl and its sister package StaticTools.jl. The biggest tradeoff of not compiling a sysimage, is that Julia's garbage collector is no longer included, so all heap allocations must be managed manually, and all code compiled *must* be type-stable. To get around this limitation, you can use static equivalents of dynamic types, such as a `StaticArray` (StaticArrays.jl) instead of an `Array` or a `StaticString` (StaticTools.jl), use `malloc` and `free` from StaticTools.jl directly, or use arena allocators with Bumper.jl. The README of StaticCompiler.jl contains a more detailed guide on how to prepare code to be compiled.

> **Note**
>
> Starting with Julia 1.12, it is anticipated that there will be a new way to compile Julia to small, static binaries with a tool called `juliac`.

## 23.8. Parallelism

Code can be made to run faster through parallel execution with multithreading (shared-memory parallelism) or multiprocessing / distributed computing. Parallelism was covered in a whole chapter (Chapter 10), but this section provides insight into recommended pacakges and patterns specific to Julia.

Many common operations such as maps and reductions can be trivially parallelised through either method by using their respective Julia packages (e.g `pmap` from Distributed.jl and `tmap` from OhMyThreads.jl). Multithreading is available on almost all modern hardware, whereas distributed computing is most useful to users of high-performance computing clusters.

### 23.8.1. Multithreading

To enable multithreading with the built-in `Threads` library, use one of the following equivalent command line flags, and give either an integer or `auto`:

```
julia --threads 4
julia -t auto
```

Once Julia is running, you can check if this was successful by calling `Threads.nthreads()`.

> **Tip**
>
> The default number of threads can be edited by adding `"julia.NumThreads": 4,` to your settings. This will be applied to the integrated terminal.

> **Note**
>
> Linear algebra code calls the low-level libraries BLAS and LAPACK. These libraries manage their own pool of threads, so single-threaded Julia processes can still make use of multiple threads, and multi-threaded Julia processes that call these libraries may run into performance issues due to the limited number of threads available in a single core. In this case, once `LinearAlgebra` is loaded, BLAS can be set to use only one thread by calling `BLAS.set_num_threads(1)`. For more information see the docs on multithreading and linear algebra.

Regardless of the number of threads, you can parallelise a for loop with the macro `Threads.@threads`. The macros `@spawn` and `@async` function similarly, but require more manual management of tasks and their results. For this reason `@threads` is recommended for those who do not wish to use third-party packages.

When designing multithreaded code, you should generally try to write to shared memory as rarely as possible. Where it cannot be avoided, you need to be careful to avoid "race conditions", i.e. situations when competing threads try to write different things to the same memory location. It is usually a good idea to separate memory accesses with loop indices, as in the example below:

```julia
results = zeros(Int, 4)
Threads.@threads for i in 1:4
    results[i] = i^2
end
```

Almost always, it is **not** a good idea to use `threadid()`.

Even if you manage to avoid any race conditions in your multithreaded code, it is very easy to run into subtle performance issues (like false sharing). For these reasons, you might want to consider using a high-level package like OhMyThreads.jl, which provides a user-friendly alternative to `Threads` and makes managing threads and their memory use much easier. The helpful translation guide demonstrates how Base multi-threading loops can be translated into the OhMyThreads API.

If the latency of spinning up new threads becomes a bottleneck, check out Polyester.jl for very lightweight threads that are quicker to start.

If you're on Linux, you should consider using ThreadPinning.jl to pin your Julia threads to CPU cores to obtain stable and optimal performance. The package can also be used to visualize where the Julia threads are running on your system (see `threadinfo()`).

### 23.8.2. Distributed computing

Julia's multiprocessing and distributed relies on the standard library `Distributed`. The main difference with multi-threading is that data isn't shared between worker processes. Once Julia is started, processes can be added with `addprocs`, and their can be queried with `nworkers`.

The macro `Distributed.@distributed` is a *syntactic* equivalent for `Threads.@threads`. Hence, we can use `@distributed` to parallelise a for loop as before, but we have to additionally deal with sharing and recombining the `results` array. We can delegate this responsibility to the standard library `SharedArrays`. However, in order for all workers to know about a function or module, we have to load it `@everywhere`:

```julia
using Distributed

# Add additional workers then load code on the workers
addprocs(3)
@everywhere using SharedArrays
@everywhere f(x) = 3x^2

results = SharedArray{Int}(4)
@sync @distributed for i in 1:4
    results[i] = f(i)
end
results
```

```
4-element SharedVector{Int64}:
  3
 12
 27
 48
```

Note that `@distributed` does not force the main process to wait for other workers, so we must use `@sync` to block execution until all computations are done.

One feature `@distributed` has over `@threads` is the possibility to specify a reduction function (an associative binary operator) which combines the results of each worker. In this case `@sync` is implied, as the reduction cannot happen unless all of the workers have finished.

```
using Distributed  # hide

@distributed (+) for i in 1:4
    i^2
end
```

```
30
```

Alternately, the convenience function `pmap` can be used to easily parallelise a `map`, both in a distributed and multi-threaded way.

```
results = pmap(f, 1:100; distributed=true, batch_size=25, on_error=ex->0)
```

```
100-element Vector{Int64}:
     3
    12
    27
    48
    75
   108
   147
   192
   243
   300
   363
   432
   507
     ⋮
 23763
 24300
 24843
 25392
 25947
 26508
 27075
 27648
```

```
28227
28812
29403
30000
```

For more functionalities related to higher-order functions, Transducers.jl and Folds.jl are the way to go.

> **Tip**
>
> MPI.jl implements the Message Passing Interface standard, which is heavily used in high-performance computing beyond Julia. The C library that MPI.jl wraps is *highly* optimized, so Julia code that needs to be scaled up to a large number of cores, such as an HPC cluster, will typically run faster with MPI than with plain `Distributed`.
> Elemental.jl is a package for distributed dense and sparse linear algebra which wraps the Elemental library written in C++, itself using MPI under the hood.

### 23.8.3. GPU programming

GPUs are specialised in executing instructions in parallel over a large number of threads. While they were originally designed for accelerating graphics rendering, more recently they have been used to train and evaluate machine learning models.

Julia's GPU ecosystem is managed by the JuliaGPU organisation, which provides individual packages for directly working with each GPU vendor's instruction set. The most popular one is CUDA.jl, which also simplifies installation of CUDA drivers for NVIDIA GPUs. Through KernelAbstractions.jl, you can easily write code that is agnostic to the type of GPU where it will run.

### 23.8.4. SIMD instructions

In the Single Instruction, Multiple Data paradigm, several processing units perform the same instruction at the same time, differing only in their inputs. The range of operations that can be parallelised (or "vectorized") like this is more limited than in the previous sections, and slightly harder to control. Julia can automatically vectorize repeated numerical operations (such as those found in loops) provided a few conditions are met:

1. Reordering operations must not change the result of the computation.
2. There must be no control flow or branches in the core computation.
3. All array accesses must follow some linear pattern.

While this may seem straightforward, there are a number of important caveats which prevent code from being vectorized. Performance annotations like `@simd` or `@inbounds` help enable vectorization in some cases, as does replacing control flow with `ifelse`.

If this isn't enough, SIMD.jl allows users to force the use of SIMD instructions and bypass the check for whether this is possible. One particular use-case for this is for vectorising non-contiguous memory reads and writes through `SIMD.vgather` and `SIMD.vscatter` respectively.

> **Tip**
>
> You can detect whether the optimizations have occurred by inspecting the output of `@code_llvm` or `@code_native` and looking for vectorised registers, types, instructions. Note that the exact things you're looking for will vary between code and CPU instruction set, an example of what to look for can be seen in this blog post by Kristoffer Carlsson.

## 23.9. Efficient types

Using an efficient data structure is a tried and true way of improving the performance. While users can write their own efficient implementations through officially documented interfaces, a number of packages containing common use cases are more tightly integrated into the Julia ecosystem.

### 23.9.1. Static arrays

Using StaticArrays.jl, you can construct arrays which contain size information in their type. Through multiple dispatch, statically sized arrays give rise to specialised, efficient methods for certain algorithms like linear algebra. In addition, the `SArray`, `SMatrix` and `SVector` types are immutable, so the array does not need to be garbage collected as it can be stack-allocated. Creating a new `SArrays` comes at almost no extra cost, compared to directly editing the data of a mutable object. With `MArray`, `MMatrix`, and `MVector`, data remains mutable as in normal arrays.

To handle mutable and immutable data structures with the same syntax, you can use Accessors.jl:

```julia
using StaticArrays, Accessors

sx = SA[1, 2, 3] # SA constructs an SArray
@set sx[1] = 3 # Returns a copy, does not update the variable
@reset sx[1] = 4 # Replaces the original
```

```
[ Info: Precompiling Accessors [7d9f7c33-5ae7-4f3b-8dc6-eff91059b697]
WARNING: Method definition getfields(LinearAlgebra.Tridiagonal{T, V} where V<:AbstractArray{T, 1
ERROR: Method overwriting is not permitted during Module precompilation. Use `__precompile__(fals
┌ Error: Error during loading of extension ConstructionBaseLinearAlgebraExt of ConstructionBase, 
│   exception =
│    1-element ExceptionStack:
│   Declaring __precompile__(false) is not allowed in files that are being precompiled.
│    Stacktrace:
│      [1] _require(pkg::Base.PkgId, env::Nothing)
│        @ Base ./loading.jl:1999
│      [2] __require_prelocked(uuidkey::Base.PkgId, env::Nothing)
│        @ Base ./loading.jl:1812
│      [3] #invoke_in_world#3
│        @ ./essentials.jl:926 [inlined]
│      [4] invoke_in_world
│        @ ./essentials.jl:923 [inlined]
│      [5] _require_prelocked
│        @ ./loading.jl:1803 [inlined]
│      [6] _require_prelocked
│        @ ./loading.jl:1802 [inlined]
│      [7] run_extension_callbacks(extid::Base.ExtensionId)
│        @ Base ./loading.jl:1295
│      [8] run_extension_callbacks(pkgid::Base.PkgId)
│        @ Base ./loading.jl:1330
│      [9] run_package_callbacks(modkey::Base.PkgId)
│        @ Base ./loading.jl:1164
│     [10] __require_prelocked(uuidkey::Base.PkgId, env::String)
│        @ Base ./loading.jl:1819
│     [11] #invoke_in_world#3
│        @ ./essentials.jl:926 [inlined]
│     [12] invoke_in_world
│        @ ./essentials.jl:923 [inlined]
│     [13] _require_prelocked(uuidkey::Base.PkgId, env::String)
│        @ Base ./loading.jl:1803
│     [14] macro expansion
│        @ ./loading.jl:1790 [inlined]
│     [15] macro expansion
│        @ ./lock.jl:267 [inlined]
│     [16] __require(into::Module, mod::Symbol)
│        @ Base ./loading.jl:1753
│     [17] #invoke_in_world#3
│        @ ./essentials.jl:926 [inlined]
│     [18] invoke_in_world
```

```
│         @ ./essentials.jl:923 [inlined]
│      [19] require(into::Module, mod::Symbol)
│         @ Base ./loading.jl:1746
│      [20] include(mod::Module, _path::String)
│         @ Base ./Base.jl:495
│      [21] include(x::String)
│         @ Accessors ~/.julia/packages/Accessors/C574d/src/Accessors.jl:1
│      [22] top-level scope
│         @ ~/.julia/packages/Accessors/C574d/src/Accessors.jl:13
│      [23] include
│         @ ./Base.jl:495 [inlined]
│    [24] include_package_for_output(pkg::Base.PkgId, input::String, depot_path::Vector{String},
│         @ Base ./loading.jl:2222
│      [25] top-level scope
│         @ stdin:3
│      [26] eval
│         @ ./boot.jl:385 [inlined]
│    [27] include_string(mapexpr::typeof(identity), mod::Module, code::String, filename::String)
│         @ Base ./loading.jl:2076
│      [28] include_string
│         @ ./loading.jl:2086 [inlined]
│      [29] exec_options(opts::Base.JLOptions)
│         @ Base ./client.jl:316
│      [30] _start()
│         @ Base ./client.jl:552
└ @ Base loading.jl:1301
Precompiling AccessorsDatesExt
  ✓ Accessors → AccessorsDatesExt
  ✓ Accessors → AccessorsTestExt
  ✓ Accessors → AccessorsStaticArraysExt
  3 dependencies successfully precompiled in 2 seconds. 15 already precompiled.
[ Info: Precompiling AccessorsDatesExt [308068fd-b978-57fd-b78f-2a52fb63dba4]
┌ Warning: Module Accessors with build ID fafbfcfd-200f-5437-0004-5913cbacb27a is missing from the
│ This may mean Accessors [7d9f7c33-5ae7-4f3b-8dc6-eff91059b697] does not support precompilation
└ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing AccessorsDatesExt [308068f
[ Info: Precompiling AccessorsIntervalSetsExt [727f68c9-d1d4-5b40-b284-36502e629768]
┌ Warning: Module Accessors with build ID fafbfcfd-200f-5437-0004-5913cbacb27a is missing from the
│ This may mean Accessors [7d9f7c33-5ae7-4f3b-8dc6-eff91059b697] does not support precompilation
└ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing AccessorsIntervalSetsExt [
[ Info: Precompiling AccessorsStructArraysExt [deedf894-762e-575a-ad8b-1df4bba63293]
┌ Warning: Module Accessors with build ID fafbfcfd-200f-5437-0004-5913cbacb27a is missing from the
```

```
│ This may mean Accessors [7d9f7c33-5ae7-4f3b-8dc6-eff91059b697] does not support precompilation b
└ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing AccessorsStructArraysExt [
[ Info: Precompiling AccessorsStaticArraysExt [91548973-bbcf-5128-ac3c-c8b871e934a5]
┌ Warning: Module Accessors with build ID fafbfcfd-200f-5437-0004-5913cbacb27a is missing from the
│ This may mean Accessors [7d9f7c33-5ae7-4f3b-8dc6-eff91059b697] does not support precompilation b
└ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing AccessorsStaticArraysExt [
[ Info: Precompiling AccessorsTestExt [3f8c54d2-55ce-559e-9981-66f7a1691a45]
┌ Warning: Module Accessors with build ID fafbfcfd-200f-5437-0004-5913cbacb27a is missing from the
│ This may mean Accessors [7d9f7c33-5ae7-4f3b-8dc6-eff91059b697] does not support precompilation b
└ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing AccessorsTestExt [3f8c54d2
[ Info: Precompiling AccessorsUnitfulExt [0f33c9ce-b40b-5f58-839e-64dee873ac84]
┌ Warning: Module Accessors with build ID fafbfcfd-200f-5437-0004-5913cbacb27a is missing from the
│ This may mean Accessors [7d9f7c33-5ae7-4f3b-8dc6-eff91059b697] does not support precompilation b
└ @ Base loading.jl:1948
[ Info: Skipping precompilation since __precompile__(false). Importing AccessorsUnitfulExt [0f33c
```

```
3-element SVector{3, Int64} with indices SOneTo(3):
 4
 2
 3
```

### 23.9.2. Classic data structures

All but the most obscure data structures can be found in the packages from the Julia-Collections organization, especially DataStructures.jl which has all the standards from the computer science courses (stacks, queues, heaps, trees and the like). Iteration and memoization utilities are also provided by packages like IterTools.jl and Memoize.jl.

### 23.9.3. Bits types

When you create custome `structs`, keeping the fields as simple, concrete types makes it more likely that the compiler will be able to allocate these objects on the stack instead of hte heap. An example of this was shown in Section 12.4.3.

# Part VI.

# Applied Financial Modeling Techniques

Here we focus on practical implementation of financial models. This section provides concrete examples and strategies for building effective models across various applications. We'll discuss model design, optimization, and validation, with an emphasis on real-world usage. The goal is to bridge theory and practice, giving you hands-on knowledge to apply advanced techniques to actual financial challenges.

# 24. Stochastic Mortality Projections

## 24.1. 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.

## 24.2. Introduction

Monte Carlo simulation is common in risk and valuation contexts. This worked example will create a population of term life insurance policies and simulate the associated claims stochastically. For this chapter, the focus is not so much on the outcomes of the model, but instead *how* and *why* the model was chosen to be setup in the way that it is.

The general structure of the example is:

1. Define the datatypes and sample data
2. Define the core logic that governs the projected outcomes for the modeled policies
3. Evaluate a few ways to structure the simulation, including:
4. allocating and non-allocating approaches
5. single threaded and mutli-threaded approaches

As will be shown, the number of simulations able to be completed on modern CPU hardware is really remarkable!

```julia
using CSV, DataFrames
using MortalityTables, FinanceCore
using Dates
using ThreadsX
using BenchmarkTools
using Random
using CairoMakie
```

## 24.3. Data and Types

### 24.3.1. `@enums` and the `Policy` Type

Our core unit of simulation with be a single life insurance `Policy`. Important characteristics include: the age a policy was issued at, the sex of the insured, and risk class to determine the assumed mortality rate. To make the example more realistic and demonstrate how it might look for a real block of inforce policies, additional fields have been included such as ID (not really used, but a common identifier) and `COLA` which is a cost-of-living-adjustment, used to modify the policy benefit through time. To be clear: the `Policy` type has more fields than will actually be used in the calculations, with the purpose to show how typical fields used in practice might be defined.

Before we define the core `Policy` type, there's a couple of types we might consider defining that would subsequently get used by `Policy`: types representing sex and risk class. A typical approach might be to simply define associated types, like this:

```
abstract type Sex end

struct Male <: Sex end
struct Female <: Sex end
```

Then, if we were to include a sex field in Policy, we could write it like this:

```
struct Policy
    # ...
    sex::Sex
    # ...
end
```

This would be a totally valid and logical approach! However, high performance is a top priority for this simulation, and therefore this approach would sacrifice being able to keep `Policy` data on the stack instead of the heap. This is because `Sex` is of unknown concrete type, which could be as simple as our definition above (with no fields in `Male` and `Female`) or someone could add a new `Alien` subtype of `Sex` with a number of associated data fields! This is why Julia cannot assume that subtypes of `Sex` will always be a simple Singletons with no associated data.

Instead, we can utilize Enums, which are a sort of lightweight type where the only thing that matters with it is distinguishing between categories. Enums in Base Julia are basically a set of constants grouped together that reference an associated integer.

```
@enum Sex Female = 1 Male = 2
@enum Risk Standard = 1 Preferred = 2
@enum PaymentMode Annual = 1 Quarterly = 4 Monthly = 12
```

Enums are convenient because it lets us use human-meaningful names for integer-based categories. Julia will also keep track of valid options: we cannot now use anything other than `Female` or `Male` where we have said a `Sex` must be specified.

> **Note**
>
> There exist Julia packages which are more powerful versions of Enums, essentially leaning into the ability to use the type system instead of just nice names for categorical variables.

Moving on to the definition of the policy itself, here's what that looks like. Note that every field has a type annotation associated with it.

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

The benefit of the way we have defined it here, using simple bits-types for each field is that our new composite `Policy` type is also a bitstype:

```
let
    p = Policy(1, Male, 1e6, 0.02, Annual, today(), 50, Standard)

    isbits(p)
end
```

```
true
```

> **Note**
>
> Type annotations are optional, but providing them is able to coerce the values to be all plain bits (i.e. simple, non-referenced values like arrays are) when the type is constructed. We could have defined `Policy` without the types specified:
>
> ```
> struct Policy
>     id
>     sex
> ```

```
    ...
    risk
end
```

Leaving out the annotations here forces Julia to assume that `Any` type could be used for the given field. Having the field be of type `Any` makes the instantiated struct data be stored in the heap, since Julia can't know the size of `Policy` in bits in advance.
We would also find that the un-annotated type is about 50 times slower than the one with annotation due to the need to utilize runtime lookup and reference memory on the heap instead of the stack.

### 24.3.2. The Data

To partially illustrate a common workflow, we'll pretend that the data we are interested in comes from a CSV file, which will be defined inline using an `IOBuffer` so that the structure of the source data is clear to the reader. Only two policies will be listed for brevity, but we will duplicate them for simulation purposes later on.

```
sample_csv_data =
    IOBuffer(
        raw"id,sex,benefit_base,COLA,mode,issue_date,issue_age,risk
         1,M,100000.0,0.03,12,1999-12-05,30,Std
         2,F,200000.0,0.03,12,1999-12-05,30,Pref"
    )
```

```
IOBuffer(data=UInt8[...], readable=true, writable=false, seekable=true, append=false, size=152,
```

We will not load the sample data using a common pattern:

1. Load the source file into a `DataFrame`
2. `map` over each row of the dataframe, and return an instantiated `Policy` object
3. Within the map, apply basic data parsing and translation logic as needed.

```
policies = let

    # read CSV directly into a dataframe
    df = CSV.read(sample_csv_data, DataFrame)                          ①

    # map over each row and construct an array of Policy objects
    map(eachrow(df)) do row
        Policy(
```

```
            row.id,
            row.sex == "M" ? Male : Female,
            row.benefit_base,
            row.COLA,
            PaymentMode(row.mode),
            row.issue_date,
            row.issue_age,
            row.risk == "Std" ? Standard : Preferred,
        )
    end


end
```

① CSV.read("sample_inforce.csv",DataFrame) could be used if the data really was in a CSV file named sample_inforce.csv instead of our demonstration IOBuffer.

```
2-element Vector{Policy}:
 Policy(1, Male, 100000.0, 0.03, Monthly, Date("1999-12-05"), 30, Standard)
 Policy(2, Female, 200000.0, 0.03, Monthly, Date("1999-12-05"), 30, Preferred)
```

## 24.4. Model Assumptions

### 24.4.1. Mortality Assumption

MortalityTables.jl provides common life insurance industry tables, and we will use two tables: one each for male and female policies respectively.

```
mort = Dict(
    Male => MortalityTables.table(988).ultimate,          ①
    Female => MortalityTables.table(992).ultimate,
)

function mortality(pol::Policy, params)
    return params.mortality[pol.sex]
end
```

① ulitmate refers to not differentiating the mortality by a 'select' underwriting period, which is common but uncessary for the deomstration in this chapter.

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

## 24.5. Model Struture and Mechanics

### 24.5.1. Core Model Behavior

The overall flow of the model loop will be as follows:

1. Determine some initialized values for each policy at the start of the projection.
2. Step through annual timesteps and simulate whether a death has occurred.
    1. If a death has occurred, log the benefit paid out.
    2. If a death has not occurred keep track of the remaining lives inforce and increment policy values.

The code is shown first and then discussion will follow it:

```
function pol_project!(out, policy, params)
    # some starting values for the given policy
    dur = length(policy.issue_date:Year(1):params.val_date) + 1
    start_age = policy.issue_age + dur - 1
    COLA_factor = (1 + policy.COLA)
    cur_benefit = policy.benefit_base * COLA_factor^(dur - 1)

    # get the right mortality vector
    qs = mortality(policy, params)

    # grab the current thread's id to write to results container
    # without conflicting with other threads
    tid = Threads.threadid()

    ω = lastindex(qs)

    @inbounds for t in 1:min(params.proj_length, ω - start_age)          ①

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

        if (rand() < q)
            # if dead then just return and don't increment the results anymore
            out.benefits[t, tid] += cur_benefit                          ②
            return
        else
            # pay benefit, add a life to the output count, and increment the benefit for next
            out.lives[t, tid] += 1
            cur_benefit *= COLA_factor
        end
```

```
    end
end
```

① inbounds turns off bounds-checking, which makes hot loops faster but first write loop without it to ensure you don't create an error (will crash if you have the error without bounds checking)
② Note that the loop is iterating down a column (i.e. across rows) for efficiency (since Julia is column-major).

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

### 24.5.2. Inputs and Outputs

#### 24.5.2.1. Inputs

The general approach for non-allocating model runs is to provide a previously instantiated container for the function to write the results to. Here, the incoming argument `out`(put) will be a named tuple with associated matrices as the `lives` and `benefits` fields. We know how many periods the model will simulate for and can therefore size the array appropriately at creation.

Other inputs include: `params` which defines some global-like parameters and `policy` which is a single `Policy` object.

> **Note**
>
> Note that the unit of the core model logic is a single policy. This simplifies the logic and reduces the chance for error due to needing to code for entire arrays of policies at a single time (as would be the case for array oriented programming style, as described in Section 5.5).

### 24.5.3. Threading

The simulations is using a threaded parallelism approach where it could be operating on any of the computer's available threads. Multi-processor (multi-machine) or GPU-based computation would require some modifications see (**?@sec-parallelism**). For the scale and complexity of this example, thread-based parallelism on a single CPU is all one should need for compute.

The threads are handled by distributed the work across threads (this is done by `ThreadsX` in the `foreach` loop below), but we need to write the appropriate place in the matrix so that threads do not compete for the same column in the output data.

Therefore, when we create the `lives` and `benefits` matrices we need to have them sized so that the number of rows is the number of projection periods and the number of columns is the number of threads.

### 24.5.4. Simulation Control

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.OffsetVecto
```

Having defined the model behavior at the unit of the policy above, we now need to define how the model should iterate over the entire population of interest. Given a vector of `Policy`s in the `policies` argument, the `project` function will:

1. Create output containers, keeping in mind the projection length and number of threads being used.
2. Loop over each policy, letting `ThreadsX.foreach` distribute the work across different threads.
3. Sum up the results across threads via `reduce`.

```
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)                                    ①
end
```

① `vec` turns the result into a 1D vector instead of a 1D matrix for later convenience.

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

## 24.6. Running the projection

Example of a single projection:

```
project(repeat(policies, 100_000), params) # <!>
```

```
(benefits = [1.2593455527498674e9, 1.3703065071649613e9, 1.437587926500566e9, 1.507241636886389e
```

1. `repeat` creates a vector that repeats the two demonstration policies many times.

### 24.6.1. Stochastic Projection

This defines a loop to calculate the results `n` times (this is only running the two policies in the sample data `n` times). This is emulating running our population of policies through $n$ stochastic scenarios, similar to what might be done for a risk or pricing exercise.

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

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

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

#### 24.6.1.1. Demonstration

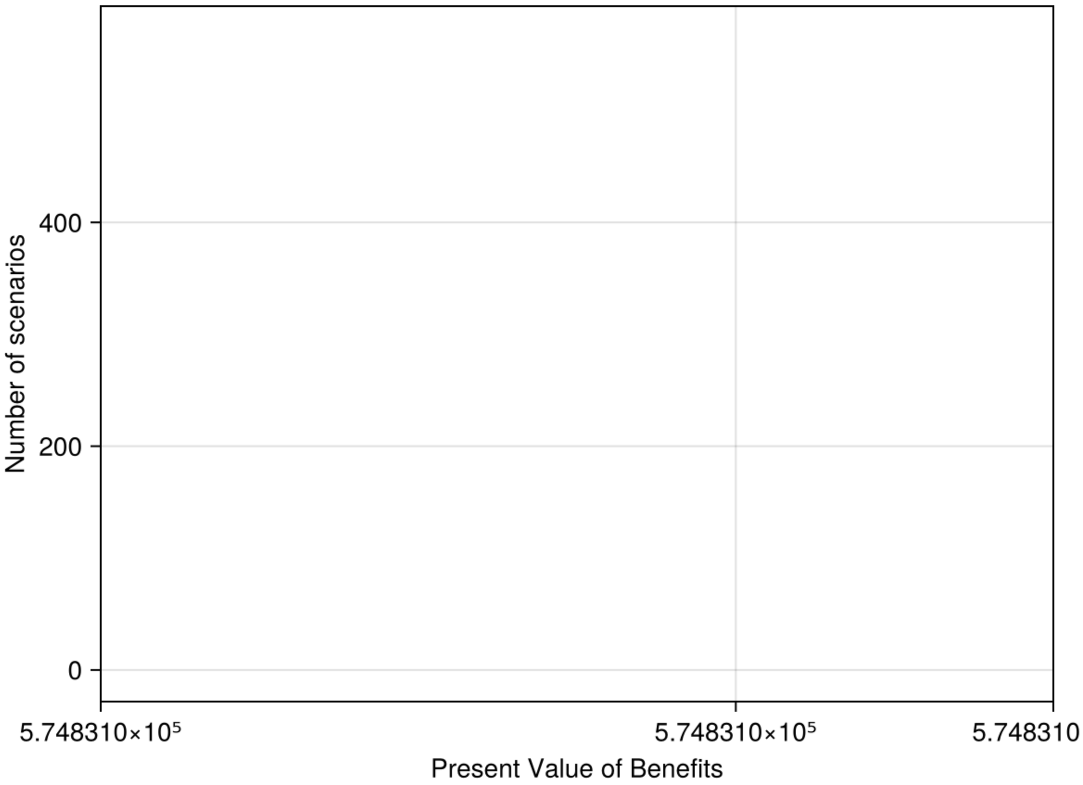
We'll simulate the two policies' outcomes 1,000 times and visualize the resulting distribution of claims value:

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

```
1000-element Vector{@NamedTuple{benefits::Vector{Float64}, lives::Vector{Int64}}}:
 (benefits = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 (benefits = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 250008.03453253524, 0.0  …  0.0, 0.0, 0.0, 0.0
 (benefits = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 515016.5511370226  …  0.0, 0.0, 0.0, 0.0,
 (benefits = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 (benefits = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 242726.24711896625, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.0
 (benefits = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 (benefits = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 (benefits = [0.0, 0.0, 209377.79296542163, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.0
```

```
(benefits = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 471313.1012018762, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.0,
(benefits = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 772524.8267055339  …  0.0, 0.0, 0.0, 0.0,
(benefits = [0.0, 0.0, 209377.79296542163, 0.0, 0.0, 0.0, 471313.1012018762, 0.0, 0.0, 0.0  …  0.0
(benefits = [0.0, 0.0, 0.0, 0.0, 0.0, 457585.5351474526, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.0,
(benefits = [197358.6511126606, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.0,
 ⋮
(benefits = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
(benefits = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
(benefits = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 250008.03453253524, 515016.5511370226  …  0.0
(benefits = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
(benefits = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
(benefits = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
(benefits = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
(benefits = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
(benefits = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
(benefits = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
(benefits = [0.0, 0.0, 209377.79296542163, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.0
(benefits = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
```

```julia
let
    v = [pv(0.03, s.benefits) for s in stoch]
    hist(v,
        bins=15,
        axis=(
            xlabel="Present Value of Benefits",
            ylabel="Number of scenarios"
        )
    )
end
```




## 24.7. Benchmarking

Using a 2024 Macbook Air M3 laptop, about 45 million policies able to be stochastically projected per second:

```
policies_to_benchmark = 4_500_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)
```

```
markTools.Trial: 24 samples with 1 evaluation.
e (min … max):  178.000 ms … 222.260 ms  ┊ GC (min … max): 0.00% … 0.00%
 (median):     181.389 ms              ┊ GC (median):    0.00%
 (mean ± σ):   184.970 ms ±  10.493 ms  ┊ GC (mean ± σ):  0.00% ± 0.00%

 ms           Histogram: frequency by time         222 ms <

ry estimate: 28.67 KiB, allocs estimate: 217.
```

## 24.8. Conclusion

This example has worked through a recommended pattern of setting up and running a stochastic simulation using a threaded approach to parallelism. The results show that quite a bit of simulation power is available using even consumer laptop hardware!

# 25. Scenario Generation

## 25.1. In This Chapter

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

## 25.2. 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.

### 25.2.1. Common PRNGs

#### 25.2.1.1. 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.

#### 25.2.1.2. 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.

#### 25.2.1.3. 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.

### 25.2.2. 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.

```julia
using Random

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

```
2×3 Matrix{Int64}:
 -2367992497871477668  -3370321129578830604   4031102505194876165
  1238044661032492679  -2301345960144502295  -4190379019896447863
```

```
using Random

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

```
2×3 Matrix{Float64}:
 0.763821  0.27558   0.661525
 0.467776  0.358406  0.46836
```

```
using Random

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

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

## 25.3. Common Economic Scenario Generation Approaches

Economic scenario generation involves the development of plausible future economic scenarios to assess the potential impact on financial portfolios, investments, or decision-making processes. Various approaches are used to generate economic scenarios, including stochastic differential equations (SDEs) and Monte Carlo simulations.

### 25.3.1. Interest Rate Models

#### 25.3.1.1. Vasicek and Cox Ingersoll Ross (CIR)

The Vasicek model is a one-factor model commonly used for simulating interest rate scenarios. It describes the dynamics of short-term interest rates using a stochastic differential equation (SDE). In a Monte Carlo simulation, we can use the Vasicek model to generate multiple interest rate paths. The CIR model is an extension of the Vasicek model with non-constant volatility. It 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 $t$.
- $\kappa$ is the speed of mean reversion, representing how quickly the interest rate reverts to its long-term mean.
- $\theta$ is the long-term mean or equilibrium level of the interest rate.
- $\sigma$ 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

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

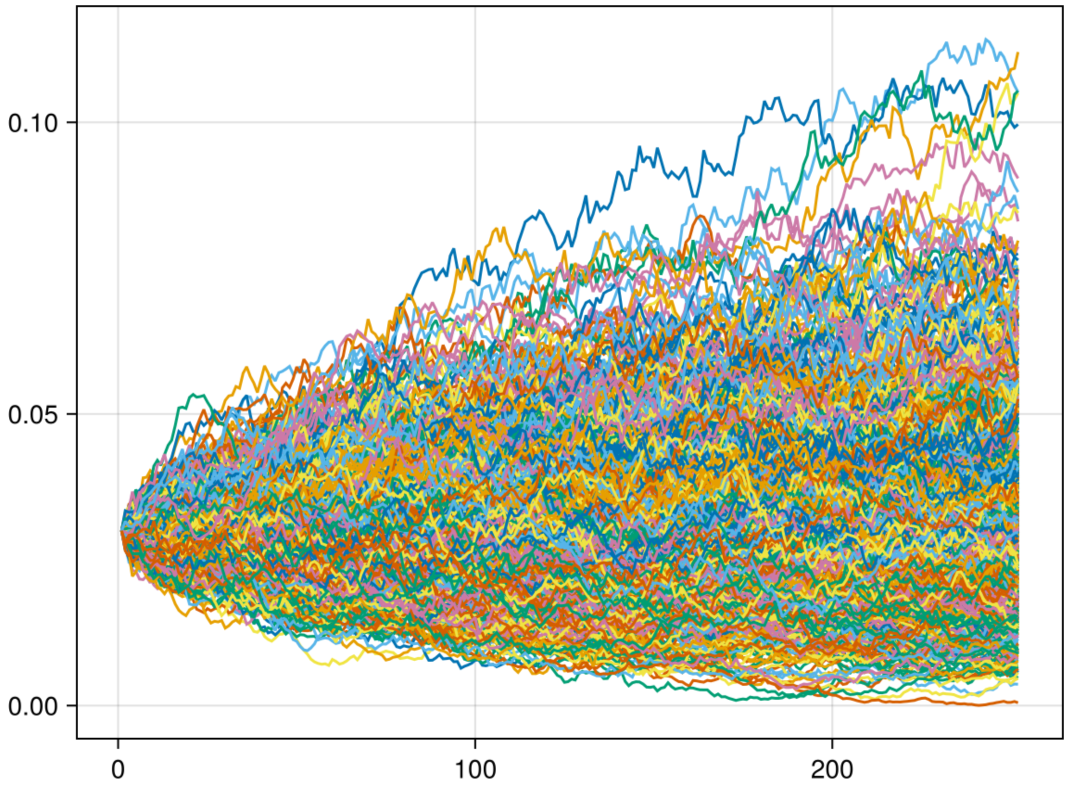
where

- $r(t)$ is the short-term interest rate at time $t$.
- $\kappa$ is the speed of mean reversion, representing how quickly the interest rate reverts to its long-term mean.
- $\theta$ is the long-term mean or equilibrium level of the interest rate.
- $\sigma$ is the volatility of the interest rate.
- $dW(t)$ is a Wiener process or Brownian motion, representing a random shock.

The following code shows a simplified implementation of a CIR model. The specification of $dr$ can be changed to make it a Vasicek model.

```
using Random, CairoMakie

# 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 * dW
            dr = κ * (θ - interest_rate_paths[i-1, j]) * Δt + σ * sqrt(interest_rate_paths[i-1
            interest_rate_paths[i, j] = max(interest_rate_paths[i-1, j] + dr, 0)  # Ensure non
        end
    end
    return interest_rate_paths
end

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

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




#### 25.3.1.2. Hull White

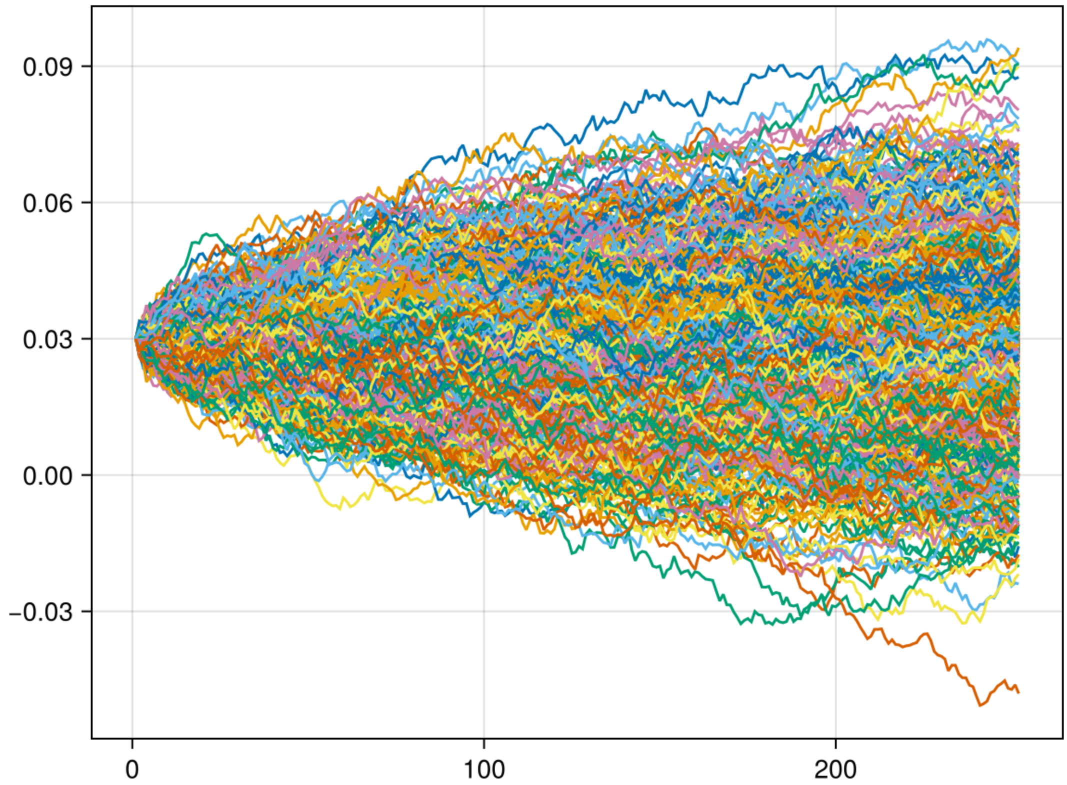
The Hull-White model is a one-factor model that extends the Vasicek model by allowing the mean reversion and volatility parameters to be time-dependent. It is commonly used for pricing interest rate derivatives. Brace-Gatarek-Musiela (BGM) Model extends the Hull-White model to incorporate more factors. 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 $t$.
- $\theta$ 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.

```julia
using Random, CairoMakie

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

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

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

# Time increment
Δt = 1 / 252

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

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

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




### 25.3.2. Stock Models

#### 25.3.2.1. Geometric Brownian Motion (GBM)

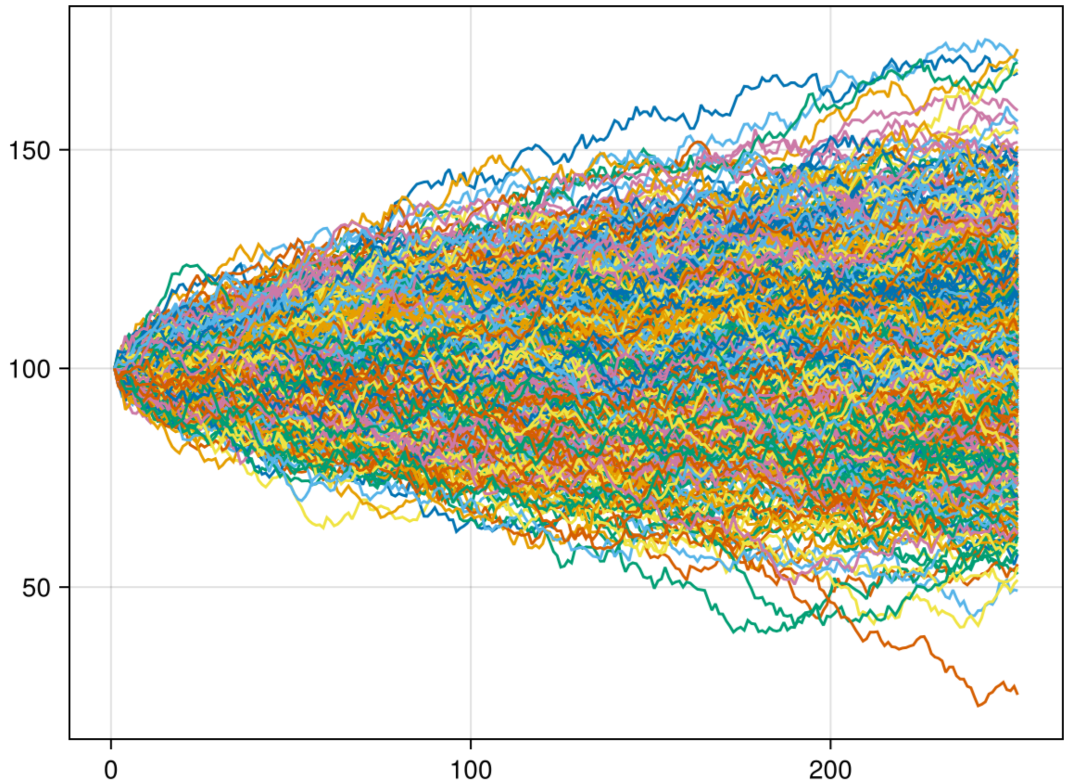
GBM is a stochastic process commonly used to model the price movement of financial instruments, including stocks. It assumes constant volatility and is characterized by a log-normal distribution. It is defined as

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

where

- $S(t)$ is the stock price at time $t$.
- $\mu$ is the drift coefficient (expected return).
- $\sigma$ is the volatility coefficient.
- $dW(t)$ is a Wiener process or Brownian motion, representing a random shock.

```julia
using Random, CairoMakie

# 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] + dS
        end
    end
    return stock_price_paths
end

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

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




The above can be restructured to use GPU instead.

```
using Random, CairoMakie

# 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 in gpu
function gbm_simulation_gpu(μ, σ, S₀, Δt, num_steps, num_simulations)
    stock_price_paths = cu(zeros(num_steps, num_simulations))
    stock_price_paths .= S₀
    dW = cu(randn(num_steps - 1, num_simulations) .* sqrt(Δt))
    dS = μ * S₀ * Δt .+ σ .* S₀ .* dW
    stock_price_paths[2:end, :] .+= cumsum(dS, dims=1)
    return stock_price_paths
end

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

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

#### 25.3.2.2. Generalized Autoregressive Conditional Heteroskedasticity (GARCH)

GARCH models capture time-varying volatility. They are often used in conjunction with other models to forecast volatility. It is defined as

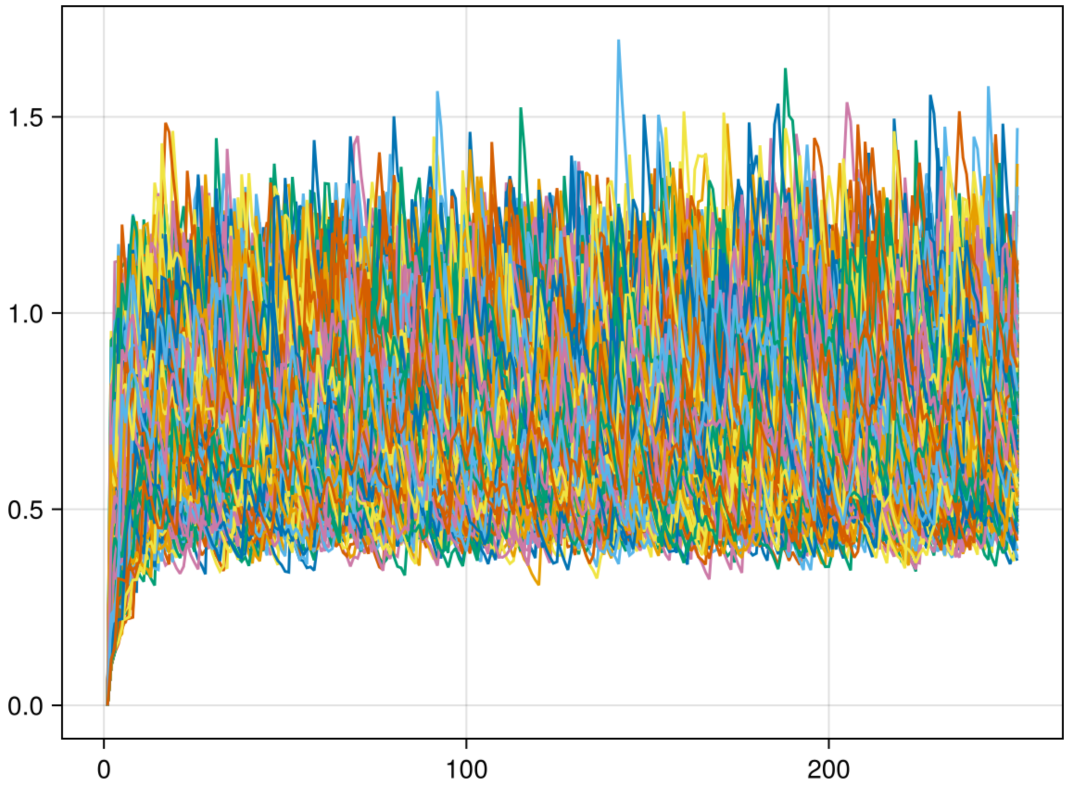
$$\sigma_t^2 = \omega + \alpha_1 r_{t-1}^2 + \beta_1 \sigma_{t-1}^2$$

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

- $\sigma_t^2$ is the conditional variance at time $t$
- $r_t$ is the return at time $t$
- $\varepsilon_t$ is a white noise or innovation process
- $\omega$, $\alpha_1$, $\beta_1$ are model parameters

```
using Random, CairoMakie

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

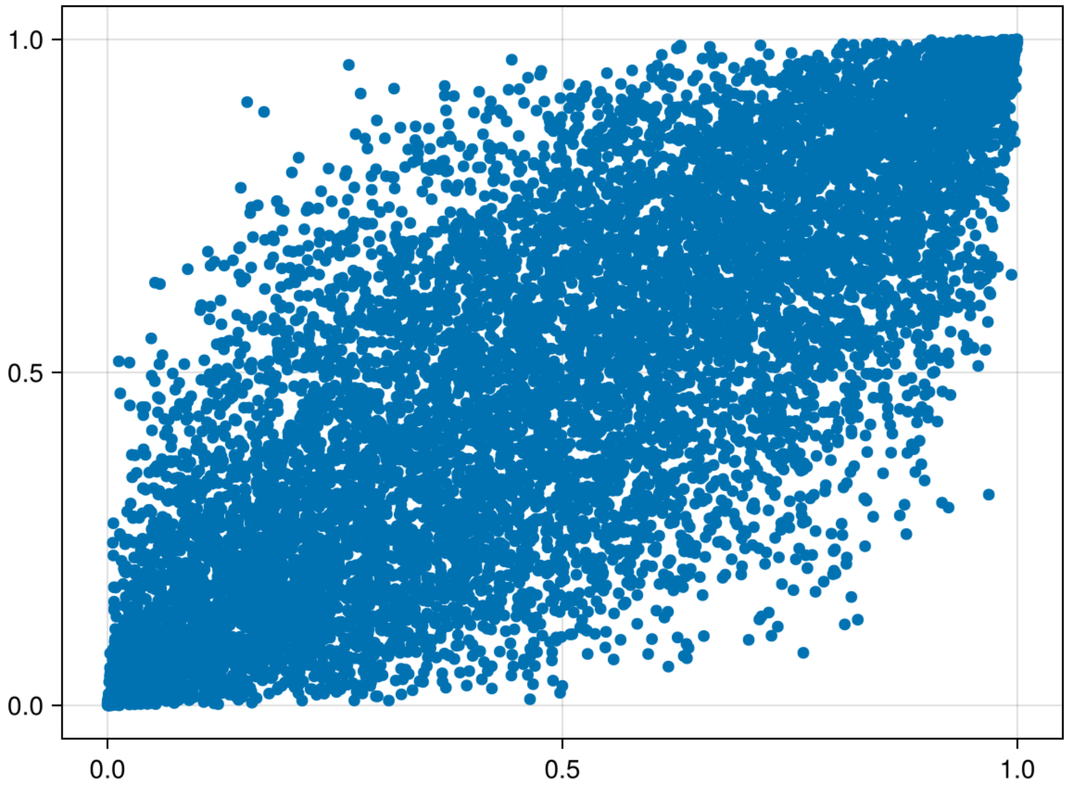
```
# GARCH(1,1) parameters
α₀ = 0.01      # Constant term
α₁ = 0.1       # Coefficient for lagged squared returns
β₁ = 0.8       # Coefficient for lagged conditional volatility

# 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_returns[i-1]
            volatility_paths[i, j] = sqrt(squared_returns[i])
        end
    end
    return volatility_paths
end

# Run GARCH simulation
garch_paths = garch_simulation(α₀, α₁, β₁, num_steps, num_simulations)

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




### 25.3.3. Copulas

Simulating data using copulas involves generating multivariate samples with specified marginal distributions and a copula structure.

```
using Random, CairoMakie, BivariateCopulas

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

# Generate a Gaussian copula
gaussian_copula = Gaussian(0.8)

# Show simulated copula
f = scatter(rand(gaussian_copula, 10^4))
f
```




Copulas can also be used to infer combined distributions from data samples.

```
using Copulas, Distributions, Random

X₁ = Gamma(2, 3)
X₂ = Pareto()
X₃ = LogNormal(0, 1)
C = ClaytonCopula(3, 0.7) # A 3-variate Clayton Copula with θ = 0.7
D = SklarDist(C, (X₁, X₂, X₃)) # The final distribution

# Generate a dataset
simu = rand(D, 1000)
# We may estimate a copula, or get parameters of underlying distributions, using the `fit` fun
D̂ = fit(SklarDist{ClaytonCopula,Tuple{Gamma,Normal,LogNormal}}, simu)
```

```
SklarDist{ClaytonCopula{3, Float64}, Tuple{Gamma{Float64}, Normal{Float64}, LogNormal{Float64}}
C: ClaytonCopula{3, Float64}(
G: Copulas.ClaytonGenerator{Float64}(0.7255762179151387)
)
```

```
m: (Gamma{Float64}(α=1.9509359315325794, θ=3.0668504198367565), Normal{Float64}(μ=6.95876479629
)
```

# 26. Similarity Analysis

## 26.1. 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.

## 26.2. 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=true, append=false, size=278,
```

```
using CSV, DataFrames, TableTransforms

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 |

```
using StatsBase

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

```
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.

## 26.3. Common Similarity Measures

The following measures are commonly used to calculate similarities.

### 26.3.1. Euclidean Distance (L2 norm)

Euclidean distance, also known as the L2 norm, is defined as

$$d = \sqrt{\sum_{i=1}^{n} (w_i - v_i)^2}$$

The distance is usually meaningful when applied to numerical data. The following Julia code shows the Euclidean distance for the first two rows in df_num.

```
using LinearAlgebra

#d₁₂ = √(∑((Array(df_num[1, :]) .- Array(df_num[2, :])) .* (Array(df_num[1, :]) .- Array(df_nu
d₁₂ = LinearAlgebra.norm(Array(df_num[1, :]) .- Array(df_num[2, :]))
```

```
0.5590169943749475
```

### 26.3.2. 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.

```
using LinearAlgebra

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

```
0.75
```

### 26.3.3. Cosine Similarity

Cosine similarity is defined as

$$d = \frac{\sum_{i=1}^{n} w_i \cdot v_i}{\sqrt{\sum_{i=1}^{n} w_i^2} \cdot \sqrt{\sum_{i=1}^{n} v_i^2}}$$

The 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.

```
using LinearAlgebra

d₁₂ = (Array(df_num[1, :]) ⋅ Array(df_num[2, :])) / norm(df_num[1, :]) / norm(df_num[2, :])
```

```
0.7071067811865475
```

The following Julia code shows the cosine similarity for the first and the third rows in arr_cat.

```
using LinearAlgebra

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

```
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.

### 26.3.4. 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
```

### 26.3.5. 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
```

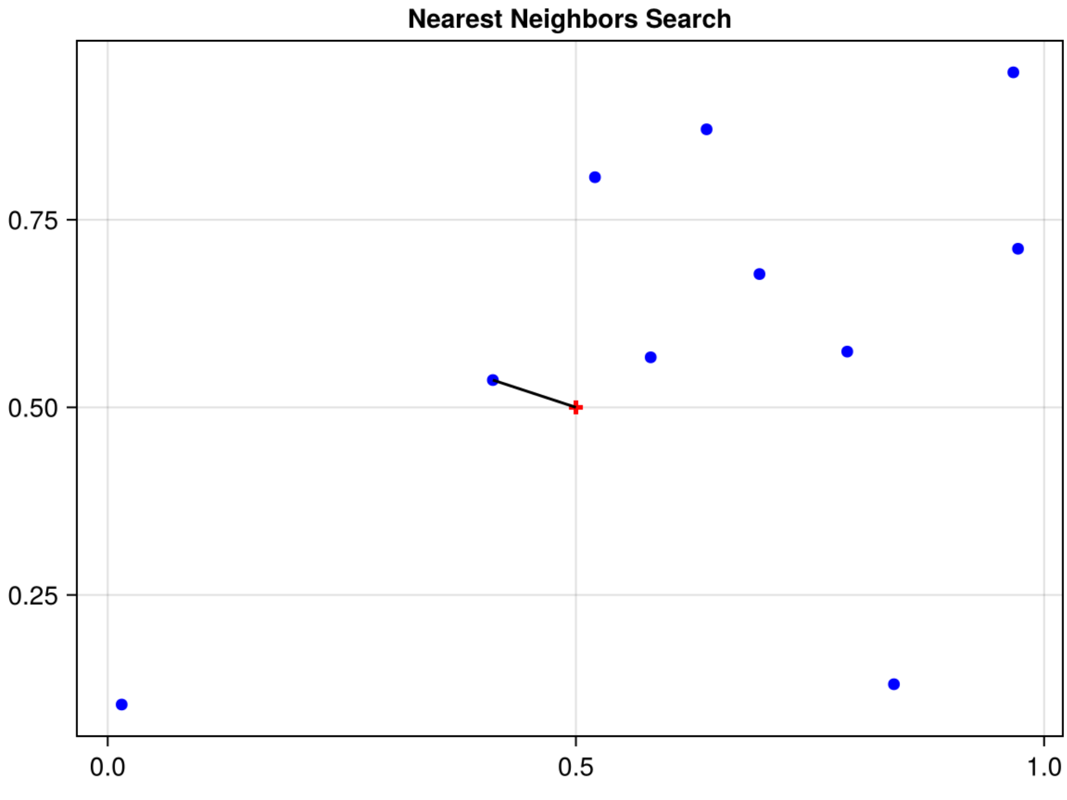
## 26.4. k-Nearest Neighbor (kNN) Clustering

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.

```
using Random, NearestNeighbors, CairoMakie

# Step 1: Generate synthetic data
Random.seed!(1234)  # For reproducibility
data = rand(10, 2)  # 10 points with 2 dimensions
println("Dataset:\n", data)

# Step 2: Create a KD-tree for efficient nearest neighbor search
kdtree = KDTree(data)

# Step 3: Define a query point (for which we want to find nearest neighbors)
query_point = [0.5, 0.5]

# Step 4: Find the nearest neighbors
# Specify how many neighbors to find
k = 1
indices, distances = knn(kdtree, query_point, k)

# Step 5: Display the results
println("\nQuery Point: ", query_point)
println("Indices of Nearest Neighbors: ", indices)
println("Distances to Nearest Neighbors: ", distances)
```

```
# Step 6: Visualize the points and the query
f = Figure()
axis = Axis(f[1, 1], title="Nearest Neighbors Search")
scatter!(data[:, 1], data[:, 2], label="Data Points", color=:blue)
scatter!([query_point[1]], [query_point[2]], label="Query Point", color=:red, marker=:cross, m

# Highlight nearest neighbors
for idx in indices
    lines!([query_point[1], data[idx, 1]], [query_point[2], data[idx, 2]], color=:black)
end
f
```

```
Dataset:
[0.5798621201341324 0.5667043025437501; 0.4112941179498505 0.5363685687293304; 0.97213608245546

Query Point: [0.5, 0.5]
Indices of Nearest Neighbors: [2]
Distances to Nearest Neighbors: [0.07597457975710152]
```



# 27. Sensitivity Analysis

## 27.1. 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.

## 27.2. Setup

Let's assume there are certain insurance policies throughout the chapter.

```
using Dates

@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
    prem::Float64
    pp::Int
    issue_date::Date
    issue_age::Int
    risk::Risk
end
```

## 27.3. The Data

```
using MortalityTables
sample_csv_data =
```

```
    IOBuffer(
        raw"id,sex,benefit_base,COLA,mode,prem,pp,issue_date,issue_age,risk
         1,M,100000.0,0.03,1,1000.0,3,1999-12-05,30,Std"
    )

mort = Dict(
    Male => MortalityTables.table(988).ultimate,
    Female => MortalityTables.table(992).ultimate,
)
```

```
Dict{Sex, OffsetArrays.OffsetVector{Float64, Vector{Float64}}} with 2 entries:
  Male   => [0.022571, 0.022571, 0.022571, 0.022571, 0.022571, 0.022571, 0.0225…
  Female => [0.00745, 0.00745, 0.00745, 0.00745, 0.00745, 0.00745, 0.00745, 0.0…
```

```
using CSV, DataFrames

policies = let

    # read CSV directly into a dataframe
    # df = CSV.read("sample_inforce.csv",DataFrame) # use local string for notebook
    df = CSV.read(sample_csv_data, DataFrame)

    # map over each row and construct an array of Policy objects
    map(eachrow(df)) do row
        Policy(
            row.id,
            row.sex == "M" ? Male : Female,
            row.benefit_base,
            row.COLA,
            row.mode,
            row.prem,
            row.pp,
            row.issue_date,
            row.issue_age,
            row.risk == "Std" ? Standard : Preferred,
        )
    end

end
```

```
1-element Vector{Policy}:
 Policy(1, Male, 100000.0, 0.03, 1, 1000.0, 3, 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
  - $p_{y-1} = 1$,
  - $p_x = p_{x-1} * (1 - mort_{age+y})/(1 + COLA)$ 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)
    s = 0.0
    if y >= 1 && y <= 120 - pol.issue_age
        for t in (pol.issue_age+y):120
            prem = 0.0
            if y <= pol.pp
                prem = pol.prem
            end
            s += sur(t - pol.issue_age - y, pol) * mort[pol.sex][t] * pol.benefit_base - prem
        end
    end
```

```
    s
end
```

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

## 27.4. Common Sensitivity Analysis Methodologies

### 27.4.1. 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, r::Float64, h=1e-3)
    p₊, p₋ = deepcopy(pol), deepcopy(pol)
    p₊.COLA, p₋.COLA = r + h, r - h
    (res(y, p₋) - res(y, p₊)) / (2res(y, pol))
end

res_wrt_r_fd(2, policies[1], 0.03) # changes in reserve at year 2 when the interest rate at 3%
```

```
0.021366520936389077
```

### 27.4.2. Regression Analyses

```
using GlobalSensitivity

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

# reserve @ year 1/2, interest rate @ 0.03 ± 0.01, prem @ 1000.0 ± 0.1
reg_anal = gsa(r1_wrt_r, RegressionGSA(), [[1, 2], [0.029, 0.031], [999.9, 1000.1]], samples=1
@show reg_anal.pearson
```

```
[ Info: Precompiling GlobalSensitivity [af5da776-676b-467e-8baf-acd8249e4f0f] (cache misses: inc
```

```
reg_anal.pearson = [0.002266826033036182 -0.9999603237826435 -0.003255946696849575]
```

```
1×3 Matrix{Float64}:
 0.00226683  -0.99996  -0.00325595
```

The Pearson Spearman coefficients show the correlation coefficient matrix between inputs and outputs.

### 27.4.3. 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.

$$Y = f_0 + \sum_{i=1}^{d} f_i(X_i) + \sum_{i<j}^{d} f_{ij}(X_i, X_j) + ... + f_{1,2,...,d}(X_1, X_2, ..., X_d)$$

$$Var(Y) = \sum_{i=1}^{d} V_i + \sum_{i<j}^{d} V_{ij} + ... + V_{1,2,...,d}$$

The Sobol Indices are "ordered", the first order indices given by $S_i = \frac{V_i}{Var(Y)}$, the contribution to the output variance of the main effect of $X_i$. Therefore, it measures the effect of varying $X_i$ alone, but averaged over variations in other input parameters. It is standardized by the total variance to provide a fractional contribution. Higher-order interaction indices $S_{ij}, S_{ijk}$ and so on can be formed by dividing other terms in the variance decomposition by $Var(Y)$.

```
using QuasiMonteCarlo, GlobalSensitivity

# reserve @ year 1/2, interest rate @ 0.03 ± 0.01, prem @ 1000.0 ± 0.1
L, U = QuasiMonteCarlo.generate_design_matrices(1000, [1, 0.029, 999.9], [2, 0.031, 1000.1], S
s = gsa(r1_wrt_r, Sobol(), L, U)
@show s.S1
@show s.ST
```

```
┌ Warning: The `generate_design_matrices(n, d, sampler, R = NoRand(), num_mats)` method does not p
│   Prefer using randomization methods such as `R = Shift()`, `R = MatousekScrambling()`, etc., se
└ @ QuasiMonteCarlo ~/.julia/packages/QuasiMonteCarlo/KvLfb/src/RandomizedQuasiMonteCarlo/itera
```

```
s.S1 = [0.0, 1.2308143537488936, -0.00010730838653271882]
s.ST = [0.0, 1.0014202549551285, 5.83151055643934e-6]
```

```
3-element Vector{Float64}:
 0.0
 1.0014202549551285
 5.83151055643934e-6
```

The output shows the first order and total order of variations in different input parameters.

### 27.4.4. Morris Method

The Morris method also known as Morris's OAT method where OAT stands for One At a Time can be described in the following steps:

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

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.

```julia
using GlobalSensitivity

# reserve @ year 1/2, interest rate @ 0.03 ± 0.01, prem @ 1000.0 ± 0.1
m = gsa(r1_wrt_r, Morris(), [[1, 2], [0.029, 0.031], [999.9, 1000.1]])
@show m.means
@show m.variances
```

```
m.means = [0.0 -719176.0096029512 -17.24314733076375]
m.variances = [0.0 2.4208765409777492e8 0.015651580821343488]
```

```
1×3 Matrix{Float64}:
 0.0  2.42088e8  0.0156516
```

From the means it can be observed which variables are more important, and the variances imply higher degree of nonlinearity or interactions with other variables.

### 27.4.5. Fourier Amplitude Sensitivity Tests

FAST offers a robust, especially at low sample size, and computationally efficient procedure to get the first and total order indices as discussed in Sobol. It utilizes monodimensional Fourier decomposition along a curve, exploring the parameter space. The curve is defined by a set of parametric equations,

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

where $s$ is a scalar variable varying over the range $-\infty < s < +\infty$, $G_i$ are transformation functions and $w_i, \forall i = 1, 2, ..., N$ is a set of different (angular) frequencies, to be properly selected, associated with each factor for all $N$ (samples) number of parameter sets.

```julia
using GlobalSensitivity

# reserve @ year 1/2, interest rate @ 0.03 ± 0.01, prem @ 1000.0 ± 0.1
fast = gsa(r1_wrt_r, eFAST(), [[1, 2], [0.029, 0.031], [999.9, 1000.1]], samples=1000)
@show fast.S1
@show fast.ST
```

```
fast.S1 = [4.996422587017407e-12 0.997681142730249 5.794912537635325e-6]
fast.ST = [7.174666488696246e-7 0.9999935931694235 0.0023094980773166185]
```

```
1×3 Matrix{Float64}:
 7.17467e-7  0.999994  0.0023095
```

The output shows the first order and total order of variations in different input parameters.

### 27.4.6. Automatic Differentiation

By applying the chain rule repeatedly on elementary operations of computer calculations, automatic differentiation can be applied to measure impacts of small differences. More details in Chapter 14 on automatic differentiation.

### 27.4.7. Scenario Analyses

Scenarios can be generated following scenario generation methodologies to evaluate impacts. More details in Chapter 25 on scenario generation.

When scenarios are generated to evaluate sensitivities, one may need to take the following into consideration.

- Reverse stress testing. Reverse stress testing in scenario analysis involves identifying extreme scenarios that could potentially lead to catastrophic outcomes for a financial institution or a system. Unlike traditional sensitivity testing to simulate the impact of adverse events on the system, reverse stress testing starts with a catastrophic outcome and works backwards to determine the combination of events or circumstances that could lead to such an outcome.

One typically follows these steps to do reverse stress testing. – Define a critical failure point (e.g., bankruptcy, system outage, regulatory breach). – Analyze the combinations of events or variables that could cause the failure. – Model the path from normal conditions to the adverse outcome.

Potential benefits that reverse stress testing could bring include: – Focusing on Vulnerabilities: Highlights specific scenarios to avoid at all costs. – Enhancing Resilience: Strengthens systems against extreme risks. – Regulatory Compliance: Often required in highly regulated industries like banking and energy.

- Stylistic scenarios. Developing stylistic scenarios in scenario analysis involves creating narratives or storylines that describe plausible future states or situations. These scenarios are crafted to capture key uncertainties, trends, and factors that could significantly impact the organization, industry, or environment under study.
- Backtesting against historical data. Backtesting in scenario analysis involves an iterative process of using past data to validate the effectiveness and accuracy of scenarios developed for forecasting future outcomes. Scenarios are first defined and applied on selective historical data, and refined after any discrepancies of scenario outcomes versus historical results are identified.

One typically follows these steps to do backtesting against historical data. – Define Scenarios: Establish hypothetical scenarios (e.g., market crashes, changes in interest rates, or operational disruptions), and ensure scenarios cover a range of possibilities, such as best-case, base-case, and worst-case scenarios. – Collect Historical Data: Gather relevant historical data for key variables (e.g., stock prices, interest rates, production metrics), and ensure data spans periods where similar events occurred in the past. – Model Scenario Impacts: Use historical data to simulate the impacts of the scenarios on key metrics or performance indicators. – Compare Results: Compare the modeled results of the scenarios with the actual historical outcomes, and assess how well the scenarios predict or explain the observed data. – Adjust and Refine: If the scenarios do not align with historical outcomes, refine the assumptions or parameters in the scenario models, and incorporate lessons learned from historical patterns to improve future scenario analyses.

Some considerations in incorporating historical data. – Data Quality: Ensure historical data is accurate, complete, and relevant to the scenarios being tested. – Model Limitations: Scenario models are based on assumptions that might not fully capture real-world complexities. – Overfitting: Avoid fine-tuning scenarios to perfectly match historical outcomes, as this reduces their applicability to future events. – Changing Dynamics: Historical events may not fully represent future possibilities due to changes in market conditions, regulations, or technology.

# 28. Portfolio Optimization

## 28.1. In This Chapter

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

## 28.2. The Data

```
μ = [0.1, 0.15, 0.12] # returns
ρ = [0.1 0.05 0.03;
    0.05 0.12 0.04;
    0.03 0.04 0.08] # covariances
nₐ = length(μ) # number of assets
```

```
3
```

## 28.3. Theory

Harry Markowitz introduced the modern portfolio theory in 1952. The main idea is that investors are pursuing to maximize their expected return of a portfolio given a certain amount of risk. By definition any portfolio yielding a higher return must have higher amount of risk, so there is a trade-off between desired expected returns and allowable risks. The risk versus maximized expected return relationship can be plotted out as a curve, a.k.a. the efficient frontier.

## 28.4. Mathematical tools

### 28.4.1. Mean-variance optimization model

Mean-variance optimization is a mathematical framework that seeks to maximize expected returns while minimizing portfolio variance (or standard deviation). It involves calculating the expected return and risk of individual assets and finding the optimal combination of assets to achieve the desired risk-return tradeoff.

$$\begin{aligned} \text{minimize} \quad & w^T \Sigma w \\ \text{subject to} \quad & R^T \geq \mu_{\text{target}} \\ & 1^T w = 1 \\ & w \geq 0 \end{aligned}$$

```julia
using JuMP, Ipopt

# Create an optimization model
model = Model(optimizer_with_attributes(Ipopt.Optimizer, "print_level" => 0))
# Set up weights as variables to optimize
@variable(model, w[1:nₐ] >= zero(0.0))
# Objective: minimize portfolio variance
@objective(model, Min, sum(w[i] * ρ[i, j] * w[j] for i in 1:nₐ, j in 1:nₐ))
# Constraints: Sum of portfolio weights should equal to 1, and all weights should be zero or p
@constraint(model, sum(w) == 1)
# May also add additional constraints
# target_return = 0.1
# @constraint(model, dot(μ, w) >= target_return)
# Solve the optimization problem
optimize!(model)
# Print results
@show "Optimal Portfolio Weights:"
for i = 1:nₐ
    @show ("Asset ", i, ": ", value.(w)[i])
end
```

```
******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
 Ipopt is released as open source code under the Eclipse Public License (EPL).
         For more information visit https://github.com/coin-or/Ipopt
******************************************************************************

"Optimal Portfolio Weights:" = "Optimal Portfolio Weights:"
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 1, ": ", 0.3333333012309821)
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 2, ": ", 0.16666675086886984)
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 3, ": ", 0.4999999479001481)
```

In mean-variance portfolio optimization, incorporating a cost of risk-based capital on assets is a practical consideration that reflects the additional capital required to support riskier assets in a portfolio. This approach ensures that the optimization process not only maximizes returns relative to risk but also considers the regulatory or internal cost implications associated with holding riskier assets.

$$\begin{aligned} \text{maximize} \quad & w^T R_{adj} \\ \text{subject to} \quad & w^T \Sigma w \leq \sigma^2_{max} \\ & 1^T w = 1 \\ & w \geq 0 \end{aligned}$$

where $R_{adj} = [(\mu_1 - \lambda_1), (\mu_2 - \lambda_2), ..., (\mu_N - \lambda_N)]$ is the adjusted expected returns.

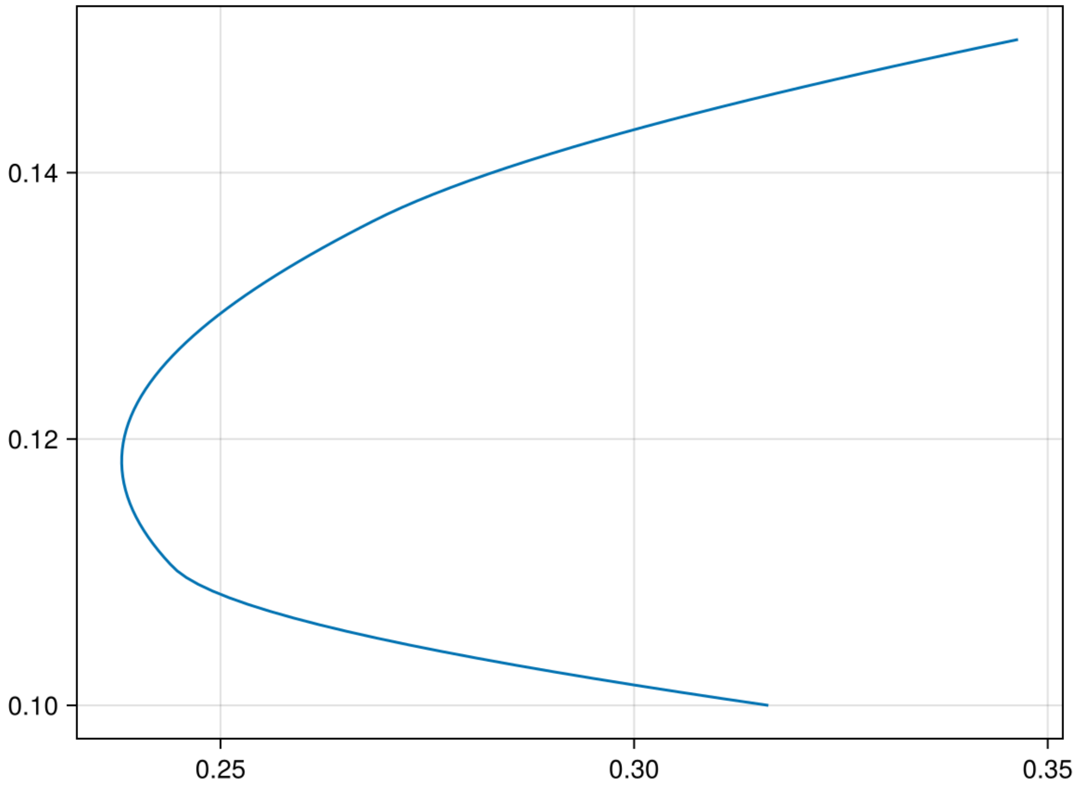
```
using JuMP, Ipopt

# Create an optimization model
model = Model(optimizer_with_attributes(Ipopt.Optimizer, "print_level" => 0))
r = μ .- [0.01, 0.02, 0.05] # risk adjusted returns
σ²_max = 0.1 # maximum portfolio variance
# Set up weights as variables to optimize
@variable(model, w[1:nₐ] >= zero(0.0))
# Objective: minimize portfolio variance
@objective(model, Max, sum(w[i] * r[i] for i in 1:nₐ))
# Constraints: Sum of portfolio weights should equal to 1, and all weights should be zero or p
@constraint(model, sum(w) == 1)
# Constraints: Sum of allowable portfolio variance is limited
@constraint(model, sum(w[i] * ρ[i, j] * w[j] for i in 1:nₐ, j in 1:nₐ) <= σ²_max)
# May also add additional constraints
# target_return = 0.1
# @constraint(model, dot(μ, w) >= target_return)
# Solve the optimization problem
optimize!(model)
# Print results
@show "Optimal Portfolio Weights:"
for i = 1:nₐ
    @show ("Asset ", i, ": ", value.(w)[i])
end
```

```
"Optimal Portfolio Weights:" = "Optimal Portfolio Weights:"
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 1, ": ", 0.16666454953827514)
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 2, ": ", 0.8333339906581219)
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 3, ": ", 1.4598036030472386e-6)
```

### 28.4.2. Efficient frontier analysis

The efficient frontier represents the set of portfolios that offer the highest expected return for a given level of risk or the lowest risk for a given level of return. Efficient frontier analysis involves plotting risk-return combinations for different portfolios and identifying the optimal portfolio on the frontier.

```julia
using JuMP, Ipopt, CairoMakie, LinearAlgebra

# Create an optimization model
model = Model(optimizer_with_attributes(Ipopt.Optimizer, "print_level" => 0))
# Set up weights as variables to optimize
@variable(model, w[1:nₐ] >= zero(0.0))
# Define objective function: minimize portfolio variance
portfolio_variance = w'ρ * w
@objective(model, Min, portfolio_variance)
# Constraints: Sum of portfolio weights should equal to 1, and all weights should be zero or p
@constraint(model, sum(w) == 1)
# Generate a range of target returns
points = 100
target_returns = range(minimum(μ), maximum(μ), length=points)

efficient_frontier = []
for target_return in target_returns
    # Add additional constraint for target return
    @constraint(model, c, dot(μ, w) == target_return)
    # Solve the problem
    optimize!(model)
    # Show solution
    if termination_status(model) == MOI.LOCALLY_SOLVED
        push!(efficient_frontier, (sqrt(objective_value(model)), target_return))
    end
    unregister(model, :c)
    delete(model, c)
end
# Plot Efficient Frontier
fig = Figure()
Axis(fig[1, 1])
lines!(map(x -> x[1], efficient_frontier), map(x -> x[2], efficient_frontier))
fig
```




### 28.4.3. Black-Litterman

The Black-Litterman model combines the views of investors with market equilibrium assumptions to generate optimal portfolios. It starts with a market equilibrium portfolio and adjusts it based on investor views and confidence levels. The model incorporates subjective opinions while maintaining diversification and risk management principles.

$$
\begin{aligned}
\text{maximize} \quad & \mu^T w - \lambda \cdot \frac{1}{2} w^T \Sigma w \\
\text{subject to} \quad & \sum_{i=1}^{N} w_i = 1 \\
& w_i \geq 0, \quad \forall i
\end{aligned}
$$

```
using JuMP, Ipopt

λ = 2.5 # risk aversion
```

```
rfr = 0.02 # risk free rate
# Market equilibrium parameters (prior)
μ_market = [0.08, 0.08, 0.08] # Market equilibrium return
Σ_market = ρ # Market equilibrium covariance matrix
# Investor views
Q = μ # Expected returns on assets according to investor views
P = [1 0 0; 0 1 0; 0 0 1]     # Pick matrix specifying which assets views are on
Ω = [0.001^2 0.0 0.0; 0.0 0.002^2 0.0; 0.0 0.0 0.003^2]  # Views uncertainty (covariance matri

# Create an optimization model
model = Model(optimizer_with_attributes(Ipopt.Optimizer, "print_level" => 0))
# Set up weights as variables to optimize
@variable(model, w[1:nₐ] >= zero(0.0))
# Black-Litterman expected return adjustment
Σ_prior_inv = inv(Σ_market)
τ = 0.05 # Scaling factor
# Calculate the posterior expected returns
μ_posterior = Σ_prior_inv * (τ * Σ_market * (Σ_prior_inv + P' * inv(Ω) * P)) \
              (τ * Σ_market * (Σ_prior_inv * μ_market + P' * inv(Ω) * Q) + Σ_prior_inv * μ_mar
# Objective: maximize sharpe ratio
sr = (w' * μ_posterior - rfr) / (λ / 2 * w' * Σ_market * w)
@objective(model, Max, sr)
# Constraints: Sum of portfolio weights should equal to 1, and all weights should be zero or p
@constraint(model, sum(w) == 1)
# Solve the optimization problem
optimize!(model)
# Print results
v = sqrt(value.(w)' * Σ_market * value.(w))
@show "Optimal Portfolio Weights, Expected Portfolio Return, Portfolio Volatility:", v
for i = 1:nₐ
    @show ("Asset ", i, ": ", value.(w)[i], value.(w)[i] * μ_posterior[i])
end
```

```
("Optimal Portfolio Weights, Expected Portfolio Return, Portfolio Volatility:", v) = ("Optimal Por
("Asset ", i, ": ", value.(w)[i], value.(w)[i] * μ_posterior[i]) = ("Asset ", 1, ": ", 2.0824146208
("Asset ", i, ": ", value.(w)[i], value.(w)[i] * μ_posterior[i]) = ("Asset ", 2, ": ", 0.2311617288
("Asset ", i, ": ", value.(w)[i], value.(w)[i] * μ_posterior[i]) = ("Asset ", 3, ": ", 0.7688382502
```

### 28.4.4. Risk Parity

Risk parity is an asset allocation strategy that allocates capital based on risk rather than traditional measures such as market capitalization or asset prices. It aims to balance risk contributions across different assets or asset classes to achieve a more stable portfolio. Risk parity portfolios often include assets with different risk profiles, such as stocks, bonds, and commodities.

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{N} (w_i \cdot \sqrt{\sigma_i})^2 \\
\text{subject to} \quad & \sum_{i=1}^{N} w_i = 1 \\
& w_i \geq 0, \quad \forall i
\end{aligned}
$$

```
using JuMP, Ipopt

# Create an optimization model
model = Model(optimizer_with_attributes(Ipopt.Optimizer, "print_level" => 0))
# Set up weights as variables to optimize
@variable(model, w[1:nₐ] >= zero(0.0))
# Objective: minimize portfolio variance
portfolio_variance = w'ρ * w
margin = (ρ * w ./ sqrt(portfolio_variance)) .* w
risk_contributions = margin ./ sum(margin)
target = repeat([1.0 / nₐ], nₐ)
@objective(model, Max, sum((risk_contributions .- target) .^ 2))
# Constraints: Sum of portfolio weights should equal to 1, and all weights should be zero or p
@constraint(model, sum(w) == 1)
# Solve the optimization problem
optimize!(model)
# Print results
@show "Optimal Portfolio Weights:"
for i = 1:nₐ
    @show ("Asset ", i, ": ", value.(w)[i])
end
```

```
"Optimal Portfolio Weights:" = "Optimal Portfolio Weights:"
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 1, ": ", -6.957484531612737e-9)
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 2, ": ", 1.0000000131375544)
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 3, ": ", -6.180069741122123e-9)
```

### 28.4.5. Sharpe Ratio Maximization

The Sharpe ratio measures the risk-adjusted return of a portfolio and is calculated as the ratio of excess return to volatility. Maximizing the Sharpe ratio involves finding the portfolio allocation that offers the highest risk-adjusted return. This approach focuses on achieving the best tradeoff between risk and return.

$$
\begin{aligned}
\text{maximize} \quad & \frac{E[R_p] - R_f}{\sigma_p} \\
\text{subject to} \quad & \sum_{i=1}^{N} w_i = 1 \\
& w_i \geq 0, \quad \forall i
\end{aligned}
$$

```julia
using JuMP, Ipopt

# Create an optimization model
model = Model(optimizer_with_attributes(Ipopt.Optimizer, "print_level" => 0))
# Set up weights as variables to optimize
@variable(model, w[1:nₐ] >= zero(0.0))
# Objective: minimize portfolio variance
rfr = 0.05 # risk free rate
@objective(model, Max, (dot(μ, w) - rfr) / sqrt(sum(w[i] * ρ[i, j] * w[j] for i in 1:nₐ, j in
# Constraints: Sum of portfolio weights should equal to 1, and all weights should be zero or p
@constraint(model, sum(w) == 1)
# Solve the optimization problem
optimize!(model)
# Print results
@show "Optimal Portfolio Weights:"
for i = 1:nₐ
    @show ("Asset ", i, ": ", value.(w)[i])
end
```

```
"Optimal Portfolio Weights:" = "Optimal Portfolio Weights:"
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 1, ": ", 0.010841995514843134)
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 2, ": ", 0.5352292318109132)
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 3, ": ", 0.45392877267424375)
```

### 28.4.6. Robust Optimization

Robust optimization techniques aim to create portfolios that are resilient to uncertainties and fluctuations in market conditions. These techniques consider a range of possible scenarios and optimize portfolios to perform well across different market environments. A robust parameter in robust portfolio optimization is typically chosen to ensure the portfolio's performance remains stable and satisfactory under different market conditions or variations in input data. Robust optimization may involve incorporating stress tests, scenario analysis, or robust risk measures into the portfolio construction process.

$$\begin{aligned} \text{minimize} \quad & w^T \Sigma w + \gamma \|w - w_0\|_2^2 \\ \text{subject to} \quad & \sum_{i=1}^{N} w_i = 1 \\ & w_i \geq 0, \quad \forall i \\ & \|(\Sigma^{1/2}(w - w_0))\|_2 \leq \epsilon \end{aligned}$$

```
using JuMP, Ipopt

# Create an optimization model
model = Model(optimizer_with_attributes(Ipopt.Optimizer, "print_level" => 0))
# Set up weights as variables to optimize
@variable(model, w[1:nₐ] >= zero(0.0))
# Objective: minimize portfolio variance
ε = 0.05  # Uncertainty level
γ = 0.1  # Robustness parameter
w₀ = [0.3, 0.4, 0.3] # expected weights
@objective(model, Min, dot(w, ρ * w) + γ * sum((w[i] - w₀[i])^2 for i in 1:nₐ))
# Constraints: Sum of portfolio weights should equal to 1, and all weights should be zero or p
@constraint(model, sum(w) == 1)
@constraint(model, sum((ρ[i, j] * (w[i] - w₀[i]) * (w[j] - w₀[j])) for i in 1:nₐ, j in 1:nₐ) <
# Solve the optimization problem
optimize!(model)
# Print results
@show "Optimal Portfolio Weights:"
for i = 1:nₐ
    @show ("Asset ", i, ": ", value.(w)[i])
end
```

```
"Optimal Portfolio Weights:" = "Optimal Portfolio Weights:"
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 1, ": ", 0.31250000098314346)
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 2, ": ", 0.31250000376951037)
("Asset ", i, ": ", value.(w)[i]) = ("Asset ", 3, ": ", 0.37499999524734623)
```

### 28.4.7. Asset weights from different methodologies

Table 28.1.: Optimized asset weights from different methodologies

| Methodology | Asset weights |
|---|---|
| Standard mean variance | [0.33, 0.17, 0.50] |
| (with RBC costs) | [0.17, 0.83, 0.00] |
| Black-Litterman | [0.00, 0.23, 0.77] |
| Risk parity | [0.00, 1.00, 0.00] |
| Sharpe ratio | [0.01, 0.54, 0.45] |
| Robust | [0.31, 0.31, 0.38] |

Seeing from the asset weights from a standard mean variance approach, due to RBC costs asset weights shifted to ones with higher yields. Asset weights from Sharpe raio approach aligns with the Sharpe ratio for each asset. Those from Robust approach seek results not far away from expected weights under different conditions.

## 28.5. Practical considerations

### 28.5.1. Fractional purchases of assets

In traditional portfolio optimization, fractional purchases of assets refer to the ability to allocate fractions or percentages of capital to individual assets. However, in certain contexts or practical implementations, fractional purchases may not be allowed or considered.

- Practical constraints. Some investment vehicles or platforms may restrict investors from purchasing fractions of shares or assets. For instance, certain mutual funds, exchange-traded funds (ETFs), or other investment products may require whole units of shares to be purchased.

- Simplicity and cost-effectiveness. Handling fractional shares can add complexity and operational costs to portfolio management, especially in terms of transaction fees, administrative overhead, and reconciliation processes.

- Market liquidity. Some assets may have limited liquidity or trading volumes, making it impractical or difficult to execute fractional purchases without significantly impacting market prices or transaction costs.

- Regulatory considerations. Regulations in certain jurisdictions may impose restrictions on fractional share trading or ownership, potentially limiting the ability to include fractional purchases in portfolio optimization strategies.

### 28.5.2. Large number of assets

In portfolio optimization, a penalty factor for a large volume of assets typically refers to a mechanism or adjustment applied to the optimization process to mitigate the potential biases or challenges that arise when dealing with a large number of assets. This concept is particularly relevant in the context of mean-variance optimization and other optimization frameworks where computational efficiency and practical portfolio management considerations come into play. Too many assets may have the following issues.

- Dimensionality. As the number of assets (or dimensions) increases in a portfolio, traditional optimization methods may become computationally intensive or prone to overfitting. This is because the complexity of the optimization problem grows exponentially with the number of assets.
- Sparsity and concentration. In practice, not all assets may contribute equally to portfolio performance. Some assets may have negligible impact on the overall portfolio characteristics (such as risk or return) due to low weights or correlations with other assets.
- Penalizing excessive complexity. A penalty factor can be introduced to penalize portfolios that overly diversify or allocate small weights to a large number of assets. This encourages the optimization process to focus on more significant assets or reduce the complexity of the portfolio structure.

There are various ways to implement a penalty factor for a large volume of assets:

- Regularization techniques. Techniques like Lasso (L1 regularization) or Ridge (L2 regularization) regression can penalize small weights or excessive diversification by adding a penalty term to the objective function.
- Subset selection. Methods that explicitly select a subset of assets based on their contribution to portfolio performance, rather than including all assets indiscriminately.
- Heuristic adjustments. Introducing heuristic rules or adjustments based on practical portfolio management principles or empirical observations.

# 29. Bayesian Mortality Modeling

> "After a year of intense mental struggle, however, [Arthur Bailey] realized to his consternation that actuarial sledgehammering worked. He even preferred [the Bayesian underpinnings of credibility theory] to the elegance of frequentism. He positively liked formulae that described 'actual data. . . . I realized that the hard-shelled underwriters were recognizing certain facts of life neglected by the statistical theorists.' He wanted to give more weight to a large volume of data than to the frequentists' small sample; doing so felt surprisingly 'logical and reasonable.' He concluded that only a 'suicidal' actuary would use Fisher's method of maximum likelihood, which assigned a zero probability to nonevents." - Excerpt From The Theory That Would Not Die Sharon Bertsch McGrayne

## 29.1. In This Chapter

An example of using a Bayesian MCMC approach to fitting a mortality curve to sample data, with multi-level models and censored data.

## 29.2. Generating fake data

The problem of interest is to look at mortality rates, which are given in terms of exposures (whether or not a life experienced a death in a given year).

We'll grab some example rates from an insurance table, which has a "selection" component: When someone enters observation, say at age 50, their mortality is path dependent (so for someone who started being observed at 50 will have a different risk/mortality rate at age 55 than someone who started being observed at 45).

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 above.

```
using MortalityTables
using Turing
using DataFramesMeta
using MCMCChains
using LinearAlgebra
using CairoMakie
using StatsBase

n = 10_000
inforce = map(1:n) do i
    (
        issue_age=rand(30:70),
        risk_level=rand(1:3),
    )

end
```

```
10000-element Vector{@NamedTuple{issue_age::Int64, risk_level::Int64}}:
 (issue_age = 42, risk_level = 2)
 (issue_age = 42, risk_level = 3)
 (issue_age = 66, risk_level = 3)
 (issue_age = 61, risk_level = 2)
 (issue_age = 38, risk_level = 1)
 (issue_age = 31, risk_level = 3)
 (issue_age = 65, risk_level = 3)
 (issue_age = 53, risk_level = 1)
 (issue_age = 55, risk_level = 1)
 (issue_age = 46, risk_level = 3)
 (issue_age = 57, risk_level = 1)
 (issue_age = 47, risk_level = 1)
 (issue_age = 65, risk_level = 3)
 ⋮
 (issue_age = 49, risk_level = 3)
 (issue_age = 35, risk_level = 2)
 (issue_age = 48, risk_level = 2)
 (issue_age = 56, risk_level = 2)
 (issue_age = 54, risk_level = 2)
 (issue_age = 48, risk_level = 3)
 (issue_age = 57, risk_level = 3)
 (issue_age = 46, risk_level = 1)
 (issue_age = 53, risk_level = 1)
 (issue_age = 70, risk_level = 2)
```

```
 (issue_age = 59, risk_level = 1)
 (issue_age = 51, risk_level = 3)

base_table = MortalityTables.table("2001 VBT Residual Standard Select and Ultimate - Male Nons

function tabular_mortality(params, issue_age, att_age, risk_level)
    q = params.ultimate[att_age]
    if risk_level == 1
        q *= 0.7
    elseif risk_level == 2
        q = q
    else
        q *= 1.5
    end
end
```

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

```
function model_outcomes(inforce, assumption, assumption_params; n_years=5)

    outcomes = map(inforce) do pol
        alive = 1
        sim = map(1:n_years) do t
            att_age = pol.issue_age + t - 1
            q = assumption(
                assumption_params,
                pol.issue_age,
                att_age,
                pol.risk_level
            )
            if rand() < q
                out = (att_age=att_age, exposures=alive, death=1)
                alive = 0
                out
            else
                (att_age=att_age, exposures=alive, death=0)
            end
        end
        filter!(x -> x.exposures == 1, sim)

    end


    df = DataFrame(inforce)
```

```
    df.outcomes = outcomes
    df = flatten(df, :outcomes)

    df.att_age = [x.att_age for x in df.outcomes]
    df.death = [x.death for x in df.outcomes]
    df.exposures = [x.exposures for x in df.outcomes]
    select!(df, Not(:outcomes))


end

exposures = model_outcomes(inforce, tabular_mortality, base_table)
data = combine(groupby(exposures, [:issue_age, :att_age])) do subdf
    (exposures=nrow(subdf),
        deaths=sum(subdf.death),
        fraction=sum(subdf.death) / nrow(subdf))
end


data2 = combine(groupby(exposures, [:issue_age, :att_age, :risk_level])) do subdf
    (exposures=nrow(subdf),
        deaths=sum(subdf.death),
        fraction=sum(subdf.death) / nrow(subdf))
end
```

| | issue_age | att_age | risk_level | exposures | deaths | fraction |
|---|---|---|---|---|---|---|
| | Int64 | Int64 | Int64 | Int64 | Int64 | Float64 |
| 1 | 30 | 30 | 1 | 79 | 0 | 0.0 |
| 2 | 30 | 30 | 2 | 55 | 0 | 0.0 |
| 3 | 30 | 30 | 3 | 71 | 0 | 0.0 |
| 4 | 30 | 31 | 1 | 79 | 0 | 0.0 |
| 5 | 30 | 31 | 2 | 55 | 0 | 0.0 |
| 6 | 30 | 31 | 3 | 71 | 0 | 0.0 |
| 7 | 30 | 32 | 1 | 79 | 0 | 0.0 |
| 8 | 30 | 32 | 2 | 55 | 0 | 0.0 |
| 9 | 30 | 32 | 3 | 71 | 1 | 0.0140845 |
| 10 | 30 | 33 | 1 | 79 | 0 | 0.0 |
| 11 | 30 | 33 | 2 | 55 | 0 | 0.0 |
| 12 | 30 | 33 | 3 | 70 | 0 | 0.0 |
| 13 | 30 | 34 | 1 | 79 | 0 | 0.0 |
| 14 | 30 | 34 | 2 | 55 | 0 | 0.0 |
| 15 | 30 | 34 | 3 | 70 | 0 | 0.0 |
| 16 | 31 | 31 | 1 | 78 | 0 | 0.0 |
| 17 | 31 | 31 | 2 | 85 | 0 | 0.0 |
| 18 | 31 | 31 | 3 | 66 | 0 | 0.0 |
| 19 | 31 | 32 | 1 | 78 | 0 | 0.0 |
| 20 | 31 | 32 | 2 | 85 | 0 | 0.0 |
| 21 | 31 | 32 | 3 | 66 | 0 | 0.0 |
| 22 | 31 | 33 | 1 | 78 | 0 | 0.0 |
| 23 | 31 | 33 | 2 | 85 | 0 | 0.0 |
| 24 | 31 | 33 | 3 | 66 | 0 | 0.0 |
| 25 | 31 | 34 | 1 | 78 | 0 | 0.0 |
| 26 | 31 | 34 | 2 | 85 | 0 | 0.0 |
| 27 | 31 | 34 | 3 | 66 | 0 | 0.0 |
| 28 | 31 | 35 | 1 | 78 | 0 | 0.0 |
| 29 | 31 | 35 | 2 | 85 | 0 | 0.0 |
| 30 | 31 | 35 | 3 | 66 | 0 | 0.0 |
| … | … | … | … | … | … | … |

## 29.3. 1: A single binomial parameter model

Estiamte $q$, the average mortality rate, not accounting for any variation within the population/sample. Our model is defines as:

$$q\ Beta(1,1)p(\text{death})\ \text{Binomial}(q)$$

```
@model function mortality(data, deaths)
    q ~ Beta(1, 1)
    for i = 1:nrow(data)
        deaths[i] ~ Binomial(data.exposures[i], q)
    end
end

m1 = mortality(data, data.deaths)
```

```
DynamicPPL.Model{typeof(mortality), (:data, :deaths), (), (), Tuple{DataFrame, Vector{Int64}}, T
 Row │ issue_age  att_age  exposures  deaths  fraction
     │ Int64      Int64    Int64      Int64   Float64
─────┼─────────────────────────────────────────────────────
   1 │        30       30        205       0  0.0
   2 │        30       31        205       0  0.0
   3 │        30       32        205       1  0.00487805
   4 │        30       33        204       0  0.0
   5 │        30       34        204       0  0.0
   6 │        31       31        229       0  0.0
   7 │        31       32        229       0  0.0
   8 │        31       33        229       0  0.0
   9 │        31       34        229       0  0.0
  10 │        31       35        229       0  0.0
  11 │        32       32        258       1  0.00387597
  ⋮  │     ⋮         ⋮         ⋮        ⋮         ⋮
 196 │        69       69        261       9  0.0344828
 197 │        69       70        252      10  0.0396825
 198 │        69       71        242       5  0.0206612
 199 │        69       72        237       3  0.0126582
 200 │        69       73        234       7  0.0299145
 201 │        70       70        235       6  0.0255319
 202 │        70       71        229       3  0.0131004
 203 │        70       72        226       9  0.039823
 204 │        70       73        217       7  0.0322581
 205 │        70       74        210       9  0.0428571
                                  184 rows omitted, deaths = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0 … 9, 10, 5, 3,
```

### 29.3.1. Sampling from the posterior

We use a No-U-Turn-Sampler (NUTS) technique to sample multiple chains at once:

```
num_chains = 4
chain = sample(m1, NUTS(), MCMCThreads(), 400, num_chains)
```

```
Chains MCMC chain (400×13×4 Array{Float64, 3}):

Iterations        = 201:1:600
Number of chains  = 4
Samples per chain = 400
Wall duration     = 2.2 seconds
Compute duration  = 8.76 seconds
parameters        = q
internals      = lp, n_steps, is_accept, acceptance_rate, log_density, hamiltonian_energy, hamil

Summary Statistics
  parameters      mean       std      mcse   ess_bulk    ess_tail      rhat   ⋯
      Symbol   Float64   Float64   Float64    Float64     Float64   Float64   ⋯

           q    0.0076    0.0004    0.0000   765.7448   1022.5374    1.0057   ⋯
                                                            1 column omitted

Quantiles
  parameters      2.5%     25.0%     50.0%     75.0%     97.5%
      Symbol   Float64   Float64   Float64   Float64   Float64

           q    0.0069    0.0074    0.0076    0.0079    0.0084
```

Here, we have asked for the outcomes to be modeled via a single parameter for the population. We see that the posterior distribution of $q$ is very close to the overall population mortality rate:

```
sum(data.deaths) / sum(data.exposures)
```

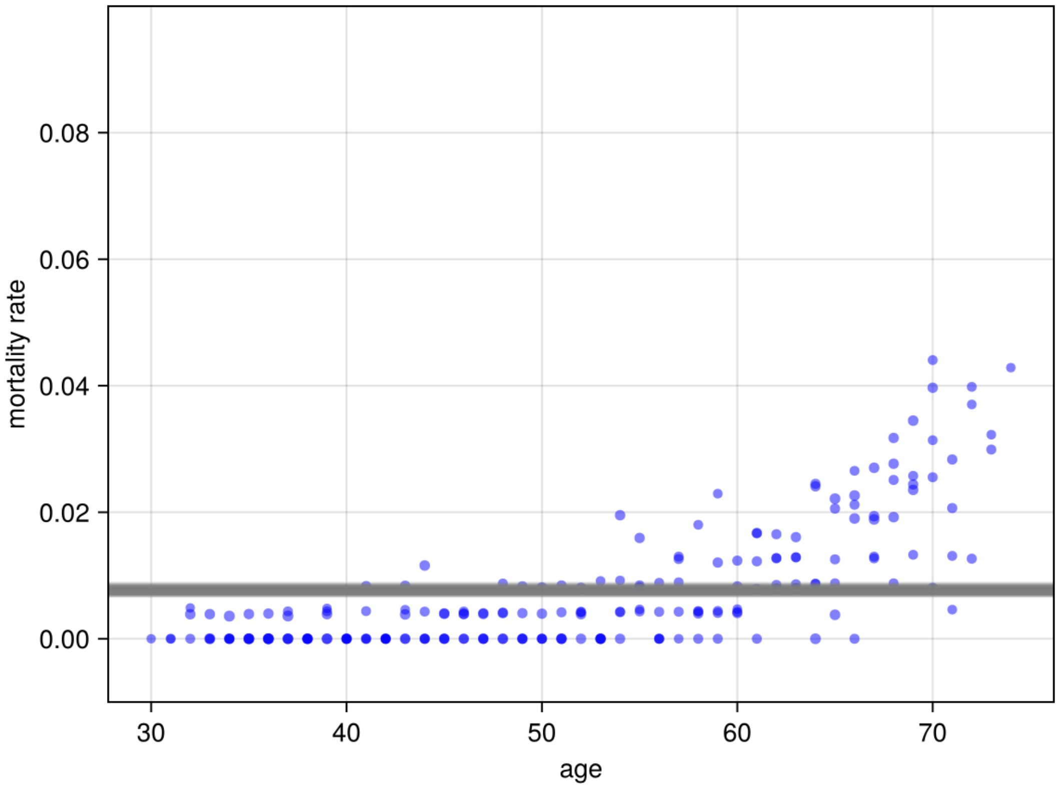
```
0.007628167413929521
```

However, We can see that the sampling of possible posterior parameters doesn't really fit the data very well since our model was so simplified. The lines represent the posterior binomial probability.

This is saying that for the observed data, if there really is just a single probability `p` that governs the true process that came up with the data, there's a pretty narrow range of values it could possibly be:

```
let
    data_weight = sqrt.(data.exposures) / 2
    f = Figure(title="Parametric Bayseian Mortality"
    )
```

```
    ax = Axis(f[1, 1],
        xlabel="age",
        ylabel="mortality rate",
        limits=(nothing, nothing, -0.01, 0.10),
    )
    scatter!(ax,
        data.att_age,
        data.fraction,
        markersize=data_weight,
        color=(:blue, 0.5),
        label="Experience data point (size indicates relative exposure quantity)",)

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

    q_posterior = sample(chain, n)[:q]


    for i in 1:n

        hlines!(ax, [q_posterior[i]], color=(:grey, 0.1))
    end

    # Need to simulate at indivudal level and then aggregate?


    sim05 = Float64[]
    sim95 = Float64[]
    for r in eachrow(data)
        outcomes = map(1:n) do i
            rand(Binomial(r.exposures, q_posterior[i]), 500)
        end
        push!(sim05, quantile(Iterators.flatten(outcomes), 0.05) / r.exposures)
        push!(sim95, quantile(Iterators.flatten(outcomes), 0.95) / r.exposures)


    end



    f
end
```




```
let
    n = 300
    q_posterior = sample(chain, n)[:q]


end
```

```
2-dimensional AxisArray{Float64,2,...} with axes:
    :iter, 1:300
    :chain, 1:1
And data, a 300×1 Matrix{Float64}:
 0.00806875696169308
 0.007432773043269552
 0.0076638643017949105
 0.0070768510806453406
 0.007936271287065663
 0.008053439825266849
 0.007862227138508178
 0.007388853405994968
```

```
0.007752246957323746
0.007666854115249467
0.007843812172816293
0.007347577862572877
0.007305832533032253
⋮
0.007796348440442385
0.008178364559182966
0.008498773585785204
0.007997322758132499
0.007969056393475586
0.00719535416209007
0.007465718725185073
0.008117885545775206
0.007822516805692927
0.007367531061215949
0.006886391285267214
0.00795573087272422
```

## 29.4. 2. Parametric model

In this example, we utilize a MakehamBeard parameterization because it's already very similar in form to a logistic function. This is important because our desired output is a probability (ie the probability of a death at a given age), so the value must be constrained to be in the interval between zero and one.

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

This isn't an ideal parameterization (e.g. we aren't including information about the select underwriting period), but is an example of utilizing Bayesian techniques on life experience data. "

```
@model function mortality2(data, deaths)
    a ~ Exponential(0.1)
    b ~ Exponential(0.1)
    c = 0.0
    k ~ truncated(Exponential(1), 1, Inf)

    # use the variables to create a parametric mortality model
    m = MortalityTables.MakehamBeard(; a, b, c, k)

    # loop through the rows of the dataframe to let Turing observe 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
```

```
mortality2 (generic function with 2 methods)
```

We combine the model with the data and sample from the posterior using a similar call as before:

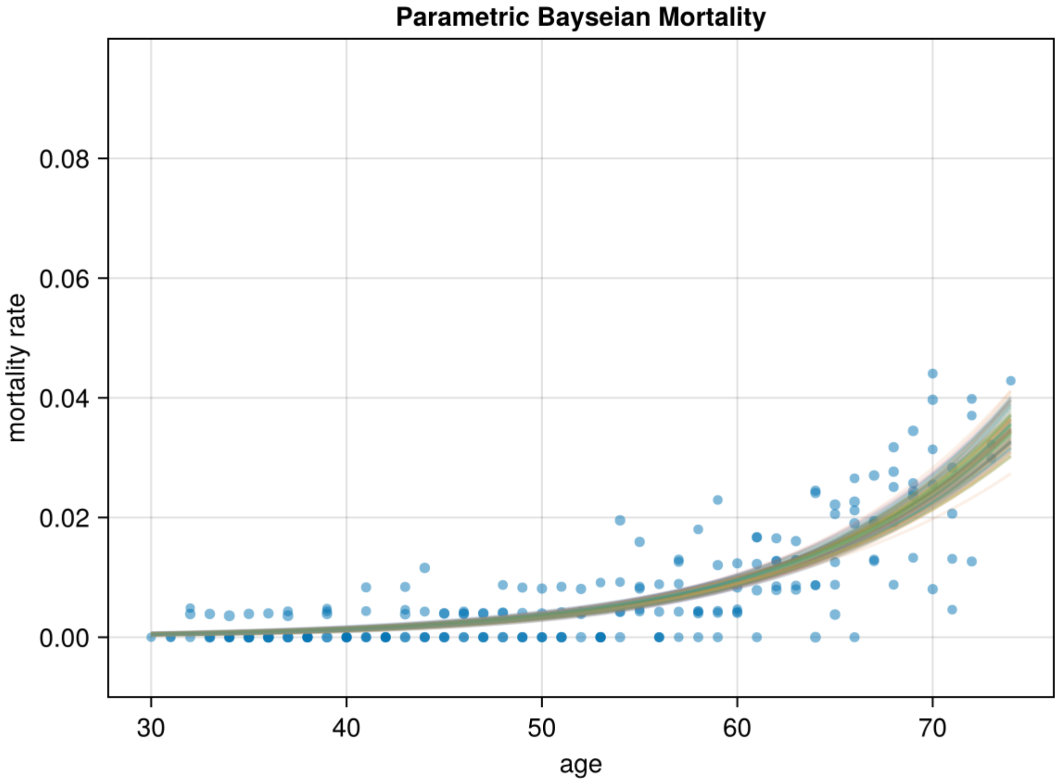
```
m2 = mortality2(data, data.deaths)

chain2 = sample(m2, NUTS(), MCMCThreads(), 400, num_chains)
```

```
Chains MCMC chain (400×15×4 Array{Float64, 3}):

Iterations        = 201:1:600
Number of chains  = 4
Samples per chain = 400
Wall duration     = 86.35 seconds
Compute duration  = 106.74 seconds
parameters        = a, b, k
internals        = lp, n_steps, is_accept, acceptance_rate, log_density, hamiltonian_energy, hamil

Summary Statistics
  parameters      mean       std      mcse   ess_bulk   ess_tail      rhat   e ⋯
      Symbol   Float64   Float64   Float64    Float64    Float64   Float64     ⋯

           a    0.1726    0.2991    0.1480     7.3681    12.8567    1.5545     ⋯
           b    1.6414    2.6746    1.3239     7.4961    31.0503    1.5285     ⋯
           k    3.1775    2.3284    1.0932     7.3811    17.5217    1.5195     ⋯
                                                                1 column omitted

Quantiles
  parameters      2.5%     25.0%     50.0%     75.0%     97.5%
      Symbol   Float64   Float64   Float64   Float64   Float64

           a    0.0000    0.0000    0.0000    0.1727    0.6905
           b    0.0870    0.0950    0.1003    1.6571    6.2725
           k    1.0263    1.3762    2.0203    6.9856    6.9857
```

### 29.4.1. Plotting samples from the posterior

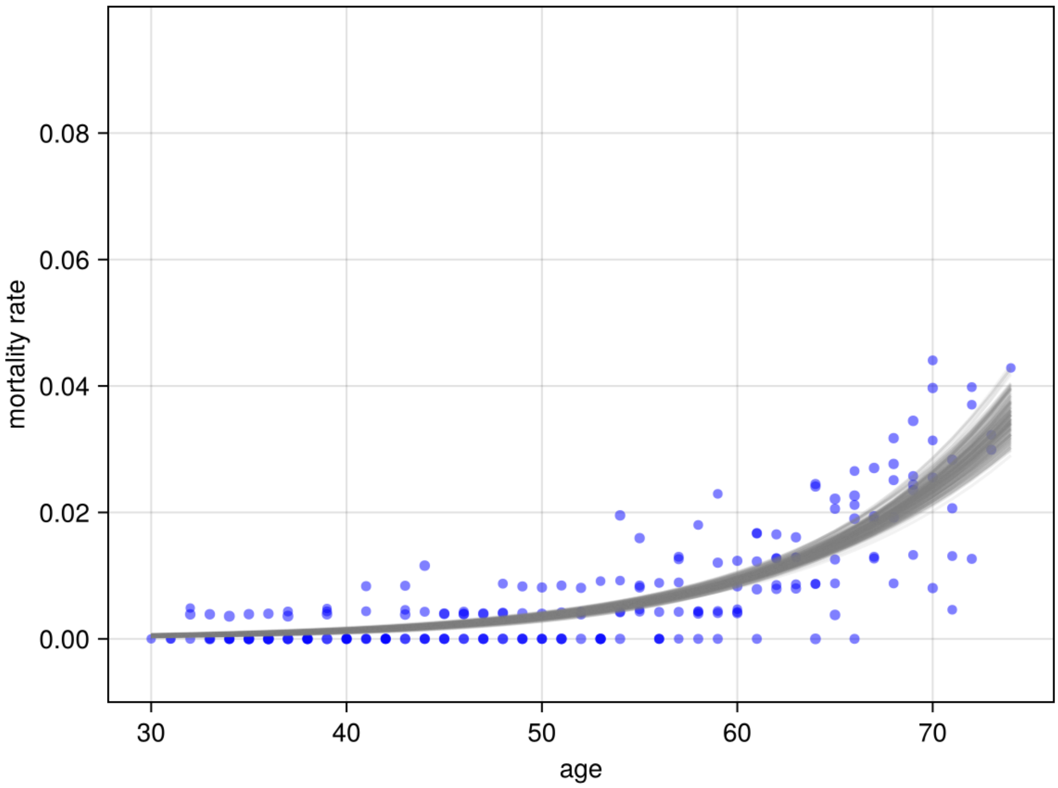
We can see that the sampling of possible posterior parameters fits the data well:

```
let
    data_weight = sqrt.(data.exposures) / 2

    p = scatter(
        data.att_age,
        data.fraction,
        markersize=data_weight,
        alpha=0.5,
        label="Experience data point (size indicates relative exposure quantity)",
        axis=(
            xlabel="age",
            limits=(nothing, nothing, -0.01, 0.10),
            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)
        lines!(ages, age -> MortalityTables.hazard(m, age), alpha=0.1, label="")
    end
    p
end
```




```
let
    data_weight = sqrt.(data.exposures) / 2
    f = Figure(title="Parametric Bayseian Mortality"
    )
    ax = Axis(f[1, 1],
        xlabel="age",
        ylabel="mortality rate",
        limits=(nothing, nothing, -0.01, 0.10),
    )
    scatter!(ax,
        data.att_age,
        data.fraction,
        markersize=data_weight,
        color=(:blue, 0.5),
        label="Experience data point (size indicates relative exposure quantity)",)

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

```
    for i in 1:n
        s = sample(chain2, 1)
        a = only(s[:a])
        b = only(s[:b])
        k = only(s[:k])
        c = 0
        m = MortalityTables.MakehamBeard(; a, b, c, k)
        qs = MortalityTables.hazard.(m, ages)
        lines!(ax, ages, qs, color=(:grey, 0.1))
    end
    f
end
```




Recall that the lines are not plotting the possible outcomes of the claims rates, but the *mean* claims rate for the given age.

## 29.5. 3. Multi-level model

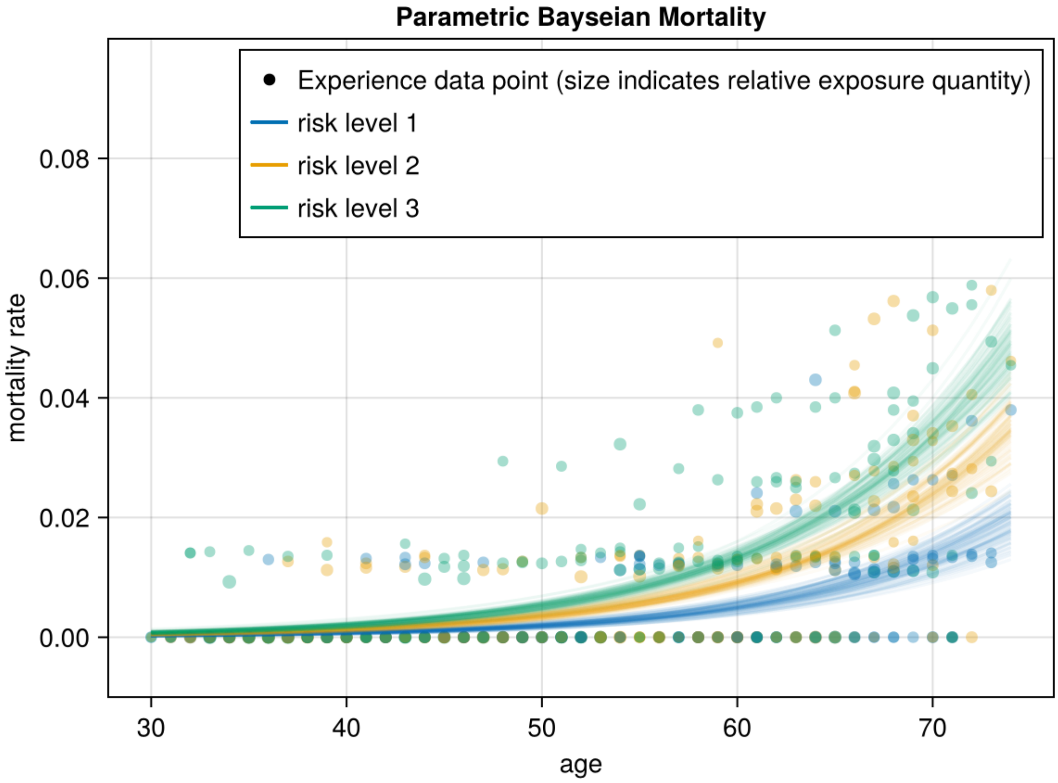
This model extends the prior to create a multi-level model. Each risk class (`risk_level`) gets its own $a$ paramater in the `MakhamBeard` model. The prior for $a_i$ is determined by the hyper-parameter $\bar{a}$.

```
@model function mortality3(data, deaths)
    risk_levels = length(levels(data.risk_level))
    b ~ Exponential(0.1)
    ā ~ Exponential(0.1)
    a ~ filldist(Exponential(ā), risk_levels)
    c = 0
    k ~ truncated(Exponential(1), 1, Inf)

    # use the variables to create a parametric mortality model

    # loop through the rows of the dataframe to let Turing observe the data
    # and how consistent the parameters are with the data
    for i = 1:nrow(data)
        risk = data.risk_level[i]

        m = MortalityTables.MakehamBeard(; a=a[risk], b, c, k)
        age = data.att_age[i]
        q = MortalityTables.hazard(m, age)
        deaths[i] ~ Binomial(data.exposures[i], q)
    end
end

m3 = mortality3(data2, data2.deaths)

chain3 = sample(m3, NUTS(), 1000)

summarize(chain3)
```

```
  parameters      mean       std      mcse    ess_bulk    ess_tail      rhat   e ⋯
      Symbol   Float64   Float64   Float64     Float64     Float64   Float64     ⋯

           b    0.0973    0.0070    0.0005    187.2359    213.6300    1.0046     ⋯
           ā    0.0001    0.0002    0.0000    369.2701    416.8890    1.0090     ⋯
        a[1]    0.0000    0.0000    0.0000    194.9829    188.0486    1.0041     ⋯
        a[2]    0.0000    0.0000    0.0000    192.0434    219.8672    1.0050     ⋯
        a[3]    0.0000    0.0000    0.0000    195.1524    206.5909    1.0060     ⋯
```

```
      k    1.9188    0.9644    0.0412   289.1036   284.4416    0.9998    ⋯
                                                       1 column omitted
```

```
let data = data2

    data_weight = sqrt.(data.exposures)
    color_i = data.risk_level
    cm = CairoMakie.Makie.wong_colors()

    p, ax, _ = scatter(
        data.att_age,
        data.fraction,
        markersize=data_weight,
        alpha=0.5,
        color=[(CairoMakie.Makie.wong_colors()[c], 0.7) for c in color_i],
        colormap=CairoMakie.Makie.wong_colors(),
        label="Experience data point (size indicates relative exposure quantity)",
        axis=(
            xlabel="age",
            limits=(nothing, nothing, -0.01, 0.10),
            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)
            lines!(ages, age -> MortalityTables.hazard(m, age), label="risk level $r", alpha=0
        end
    end
    axislegend(ax, merge=true)
    p
end
```




Again, the lines are not plotting the possible outcomes of the claims rates, but the *mean* claims rate for the given age and risk class.

## 29.6. Handling non-unit exposures

The key is to use the Poisson distribution, which is a continuous approximation to the Binomial distribution:

```
@model function mortality4(data, deaths)
    risk_levels = length(levels(data.risk_level))
    b ~ Exponential(0.1)
    ā ~ Exponential(0.1)
    a ~ filldist(Exponential(ā), risk_levels)
    c ~ Beta(4, 18)
    k ~ truncated(Exponential(1), 1, Inf)

    # use the variables to create a parametric mortality model
```

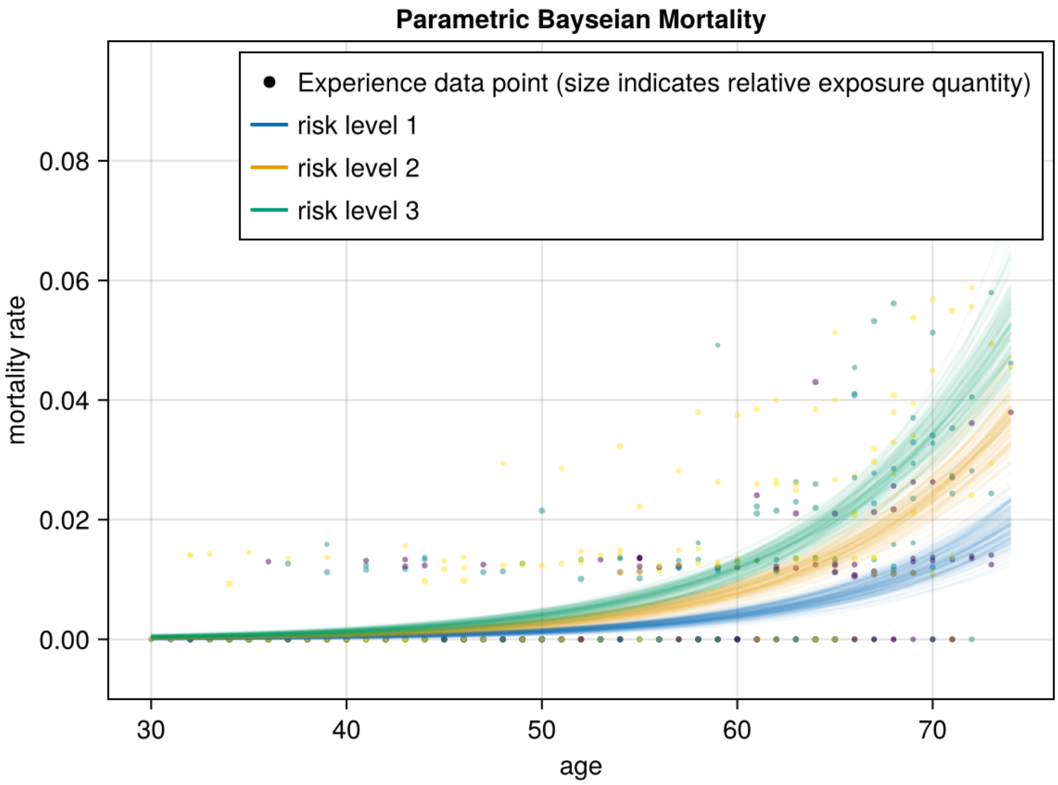
```
    # loop through the rows of the dataframe to let Turing observe the data
    # and how consistent the parameters are with the data
    for i = 1:nrow(data)
        risk = data.risk_level[i]

        m = MortalityTables.MakehamBeard(; a=a[risk], b, c, k)
        age = data.att_age[i]
        q = MortalityTables.hazard(m, age)
        deaths[i] ~ Poisson(data.exposures[i] * q)
    end
end

m4 = mortality4(data2, data2.deaths)

chain4 = sample(m4, NUTS(), 1000)
```

```
Chains MCMC chain (1000×19×1 Array{Float64, 3}):

Iterations        = 501:1:1500
Number of chains  = 1
Samples per chain = 1000
Wall duration     = 35.96 seconds
Compute duration  = 35.96 seconds
parameters        = b, ā, a[1], a[2], a[3], c, k
internals        = lp, n_steps, is_accept, acceptance_rate, log_density, hamiltonian_energy, hamil

Summary Statistics
  parameters      mean       std      mcse   ess_bulk   ess_tail      rhat   e ⋯
      Symbol   Float64   Float64   Float64    Float64    Float64   Float64     ⋯

           b    0.1133    0.0095    0.0008   135.2470   210.1857    1.0062     ⋯
           ā    0.0000    0.0000    0.0000   207.6479   354.5206    1.0039     ⋯
        a[1]    0.0000    0.0000    0.0000   148.3462   216.0662    1.0066     ⋯
        a[2]    0.0000    0.0000    0.0000   144.7798   216.0662    1.0062     ⋯
        a[3]    0.0000    0.0000    0.0000   130.7680   203.1803    1.0074     ⋯
           c    0.0008    0.0003    0.0000   338.3439   357.8855    0.9999     ⋯
           k    2.1141    1.1328    0.0392   551.9439   373.8888    1.0070     ⋯
                                                                1 column omitted

Quantiles
  parameters       2.5%     25.0%     50.0%     75.0%     97.5%
      Symbol    Float64   Float64   Float64   Float64   Float64
```

```
         b    0.0977    0.1066    0.1125    0.1192    0.1349
         ā    0.0000    0.0000    0.0000    0.0000    0.0001
      a[1]    0.0000    0.0000    0.0000    0.0000    0.0000
      a[2]    0.0000    0.0000    0.0000    0.0000    0.0000
      a[3]    0.0000    0.0000    0.0000    0.0000    0.0000
         c    0.0004    0.0006    0.0008    0.0010    0.0015
         k    1.0323    1.3020    1.7569    2.5064    5.1842
```

```
risk_factors4 = [mean(chain4[Symbol("a[$f]")]) for f in 1:3]

risk_factors4 ./ risk_factors4[2]

let data = data2

    data_weight = sqrt.(data.exposures) / 2
    color_i = data.risk_level

    p, ax, _ = scatter(
        data.att_age,
        data.fraction,
        markersize=data_weight,
        alpha=0.5,
        color=color_i,
        label="Experience data point (size indicates relative exposure quantity)",
        axis=(xlabel="age",
            limits=(nothing, nothing, -0.01, 0.10),
            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)
```

```
            lines!(ages, age -> MortalityTables.hazard(m, age), label="risk level $r", alpha=0
        end
    end
    axislegend(ax, merge=true)
    p
end
```




## 29.7. Model Predictions

We can generate predictive estimates by passing a vector of `missing` in place of the outcome variables and then calling `predict`.

We get a table of values where each row is the the prediction implied by the corresponding chain sample, and the columns are the predicted value for each of the outcomes in our original dataset.

```
preds = predict(mortality4(data2, fill(missing, length(data2.deaths))), chain4)
```

```
Chains MCMC chain (1000×615×1 Array{Float64, 3}):

Iterations        = 1:1:1000
Number of chains  = 1
Samples per chain = 1000
parameters        = deaths[1], deaths[2], deaths[3], deaths[4], deaths[5], deaths[6], deaths[7
internals         =

Summary Statistics
  parameters     mean      std     mcse   ess_bulk   ess_tail     rhat   ⋯
      Symbol  Float64  Float64  Float64    Float64    Float64  Float64   ⋯

   deaths[1]   0.0800   0.2823   0.0093   919.0958   922.2533   0.9991   ⋯
   deaths[2]   0.0800   0.2858   0.0095   956.6618  1004.0241   0.9995   ⋯
   deaths[3]   0.0980   0.3106   0.0097  1037.4826  1010.1399   1.0004   ⋯
   deaths[4]   0.0810   0.2838   0.0098   847.4833   872.5062   0.9991   ⋯
   deaths[5]   0.0590   0.2441   0.0082   874.4919   873.4523   0.9991   ⋯
   deaths[6]   0.0800   0.2787   0.0113   607.2429   611.1778   1.0139   ⋯
   deaths[7]   0.0730   0.2824   0.0088  1046.5170  1010.8263   0.9990   ⋯
   deaths[8]   0.0740   0.2768   0.0085  1069.8225  1004.0241   0.9990   ⋯
   deaths[9]   0.0770   0.2813   0.0093   933.3155   950.1473   1.0017   ⋯
  deaths[10]   0.0640   0.2529   0.0079  1033.4026  1004.0241   1.0004   ⋯
  deaths[11]   0.0590   0.2441   0.0082   865.6357   848.3052   0.9990   ⋯
  deaths[12]   0.0950   0.3132   0.0102   961.6908   980.3998   0.9998   ⋯
  deaths[13]   0.0920   0.3060   0.0093  1081.5253  1009.4549   1.0000   ⋯
  deaths[14]   0.0790   0.2843   0.0086  1088.4212  1008.0890   1.0010   ⋯
  deaths[15]   0.1040   0.3276   0.0103  1024.1093  1018.1688   0.9990   ⋯
  deaths[16]   0.0740   0.2804   0.0087  1036.9695  1009.4549   0.9990   ⋯
  deaths[17]   0.1020   0.3189   0.0100  1014.9344   734.8371   1.0004   ⋯
  deaths[18]   0.0870   0.2958   0.0093  1010.6150  1001.4290   0.9990   ⋯
  deaths[19]   0.0750   0.2710   0.0086   984.6411   978.9252   1.0011   ⋯
  deaths[20]   0.1180   0.3496   0.0108  1052.3176  1015.7701   0.9995   ⋯
  deaths[21]   0.0940   0.3021   0.0102   877.3250   870.9919   0.9990   ⋯
  deaths[22]   0.0850   0.2998   0.0094   997.6864   982.4658   0.9996   ⋯
  deaths[23]   0.1080   0.3413   0.0113   935.2134   883.7717   1.0007   ⋯
      ⋮           ⋮        ⋮        ⋮         ⋮          ⋮          ⋮     ⋱
                                                 1 column and 592 rows omitted

Quantiles
  parameters      2.5%     25.0%     50.0%     75.0%     97.5%
      Symbol   Float64   Float64   Float64   Float64   Float64

   deaths[1]    0.0000    0.0000    0.0000    0.0000    1.0000
```

```
   deaths[2]    0.0000    0.0000    0.0000    0.0000    1.0000
   deaths[3]    0.0000    0.0000    0.0000    0.0000    1.0000
   deaths[4]    0.0000    0.0000    0.0000    0.0000    1.0000
   deaths[5]    0.0000    0.0000    0.0000    0.0000    1.0000
   deaths[6]    0.0000    0.0000    0.0000    0.0000    1.0000
   deaths[7]    0.0000    0.0000    0.0000    0.0000    1.0000
   deaths[8]    0.0000    0.0000    0.0000    0.0000    1.0000
   deaths[9]    0.0000    0.0000    0.0000    0.0000    1.0000
  deaths[10]    0.0000    0.0000    0.0000    0.0000    1.0000
  deaths[11]    0.0000    0.0000    0.0000    0.0000    1.0000
  deaths[12]    0.0000    0.0000    0.0000    0.0000    1.0000
  deaths[13]    0.0000    0.0000    0.0000    0.0000    1.0000
  deaths[14]    0.0000    0.0000    0.0000    0.0000    1.0000
  deaths[15]    0.0000    0.0000    0.0000    0.0000    1.0000
  deaths[16]    0.0000    0.0000    0.0000    0.0000    1.0000
  deaths[17]    0.0000    0.0000    0.0000    0.0000    1.0000
  deaths[18]    0.0000    0.0000    0.0000    0.0000    1.0000
  deaths[19]    0.0000    0.0000    0.0000    0.0000    1.0000
  deaths[20]    0.0000    0.0000    0.0000    0.0000    1.0000
  deaths[21]    0.0000    0.0000    0.0000    0.0000    1.0000
  deaths[22]    0.0000    0.0000    0.0000    0.0000    1.0000
  deaths[23]    0.0000    0.0000    0.0000    0.0000    1.0000
      ⋮            ⋮         ⋮         ⋮         ⋮         ⋮
                                              592 rows omitted
```

# 30. Other Useful Techniques

## 30.1. 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. Also covered are elements of practical review such as static and dynamic validations, and implied rate analysis.

## 30.2. Conceptual Techniques

### 30.2.1. Taking things to the Extreme

Consider what happens if something is taken to an extreme. For example, what happens in the model if we input negative rates? Where should negative rates be allowed and can the model handle them?

### 30.2.2. Range Bounding

Sometimes you just need to know that an outcome is within a certain range - if you can develop a "high" and "low" estimate by making assumptions that you know are outside of feasible ranges, then you can determine whether something is reasonable or within tolerances.

To take an example from the pages of interview questions: say you need to determine if a mortgaged property's value is greater than the amount of the outstanding loan (say \$100,000). You don't have an appraisal, but know that it's in reasonable condition and that (1) a comparable house with many more issues sold for \$100 per square foot. You also don't know the square footage of the house, but know from 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.

## 30.3. Modeling Techniques

### 30.3.1. Serialization

# Part VII.

# Appendices

# 31. 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[1]

### 31.0.1. 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.

[1](**time?**) is a simple, built-in function. For true benchmarking purposes, see **?@sec-benchmarking**.

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.

### 31.0.2. 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.

### 31.0.3. 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.

### 31.0.4. 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.

### 31.0.5. 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 = 0.05);

julia> policy_av(pol)        # the ending account value
4048.08

julia> policy_av'(pol)       # the derivative of the account value with respect to the inputs
(annual_premium = 6.802, face = -0.0275, credit_rate = 10972.52)
```

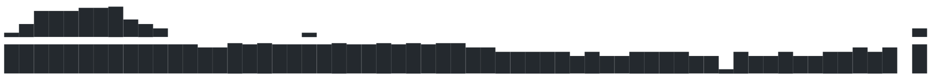
When executing the code above, Julia isn't just adding a small amount and calculating the finite difference. Differentiation is applied to entire programs through extensive use of basic derivatives and the chain rule. **Automatic differentiation**, has uses in optimization, machine learning, sensitivity testing, and risk analysis. You can read more about Julia's autodiff ecosystem here.

### 31.0.6. 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.

### 31.0.7. 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/.

### 31.0.8. 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.

Murphy, Hannah, and Cristina Criddle. 2024. "Meta AI Chief Says Large Language Models Will Not Reach Human Intelligence." https://www.ft.com/content/23fab126-f1d3-4add-a457-207a25730ad9.