Statistics with Julia:
Fundamentals for Data Science, Machine Learning and Artificial Intelligence.

D R A F T

Hayden Klok, Yoni Nazarathy

January 20, 2020

Preface to this DRAFT version

This DRAFT version of our book includes a complete structure of the contents and an almost-complete code-base using Julia 1.3. Note that most of the content of Chapters 8 and 9 is omitted and will be available through the publisher's release. We hope you find this draft useful. Please let us know of any feedback you have. What has helped you? What more you would like to see? What parts do you think can be improved?

Hayden Klok
Yoni Nazarathy,
January, 2020.

Preface

The journey of this book began at the end of 2016 when preparing material for a statistics course for The University of Queensland. At the time, the Julia language was already showing itself as a powerful new and applicable tool, even though it was only at version 0.5. For this reason, we chose Julia for use in the course, since, by exposing students to statistics with Julia early on, they would be able to employ Julia for data science, numerical computation and machine learning tasks later in their careers. This choice was not without some resistance from students and colleagues, since back then, as is still now in 2020, in terms of volume, the R-language dominates the world of statistics, in the same way that Python dominates the world of machine-learning. So why Julia?

There were three main reasons: performance, simplicity and flexibility. Julia is quickly becoming a major contending language in the world of data science, statistics, machine learning, artificial intelligence and general scientific computing. It is easy to use like R, Python and Matlab, but due to its type system and just-in-time compilation, it performs computations much more efficiently. This enables it to be fast, not just in terms of run time, but also in terms of development time. In addition, there are many different Julia packages. These include advanced methods for the data-scientist, statistician, or machine learning practitioner. Hence the language has a broad scope of application.

Our goal in writing this book was to create a resource for understanding the fundamental concepts of statistics needed for mastering machine learning, data science and artificial intelligence. This is with a view of introducing the reader to Julia through the use of it as a computational tool. The book also aims to serve as a reference for the data scientist, machine learning practitioner, bio-statistician, finance professional, or engineer, who has either studied statistics before, or wishes to fill gaps in their understanding. In today's world, such students, professionals, or researchers often use advanced methods and techniques. However, one is often required to take a step back and explore or revisit fundamental concepts. Revisiting these concepts with the aid of a programming language such as Julia immediately makes the concepts concrete.

Now, 3 years since we embarked on this book writing journey, Julia has matured beyond V1.0, and the book has matured along with it. Julia can be easily deployed by anyone who wishes to use it. However, currently many of Julia's users are hard-core developers that contribute to the language's standard libraries, and to the extensive package eco-system that surrounds it. Therefore, much of the Julia material available at present is aimed at other developers rather than end users. This is where our book comes in, as it has been written with the end-user in mind. The code examples have been deliberately written in a simple format, sometimes at the expense of efficiency and generality, but with the advantage of being easily readable. Each of the code examples aims to convey a specific statistical point, while covering Julia programming concepts in parallel. In a way, the code examples are reminiscent of examples that a lecturer may use in a lecture to illustrate concepts. The content of the book is written in a manner that does not assume any prior statistical knowledge, and in fact only assumes some basic programming experience and a basic understanding of mathematical notation.

The book contains a total of 10 chapters and 3 appendices. The content may be read continuously, or accessed in an ad-hoc manner. The structure is as follows:

Chapter 1 is an introduction to Julia, including its setup, package manager and the main packages

used in the book. The reader is introduced to some basic syntax, and programmatic structure through code examples that aim to illustrate some of the language's features.

Chapter 2 explores basic probability, with a focus on events, outcomes, independence and conditional probability concepts. Several typical probability examples are presented, along with exploratory simulation code.

Chapter 3 explores random variables and probability distributions, with a focus on the use of Julia's `Distributions` package. Discrete, continuous, univariate and multi-variate probability distributions are introduced and explored as an insightful and pedagogical task. This is done through both simulation and explicit analysis, along with the graphing of associated functions of distributions, such as the PMF, PDF, CDF etc.

Chapter 4 momentarily departs from probabilistic notions to focus on data processing, data summary and data visualizations. The concept of the `DataFrame` is introduced as a mechanism for storing heterogeneous data types with the possibility of missing values. Data frames play an integral component of data science and statistics in Julia, just as they do in R and Python. A brief summary of classic descriptive statistics and their application in Julia is also introduced. This is augmented by the inclusion of concepts such as Kernel Density Estimation and the empirical cumulative distribution function. The chapter closes with some basic functionality for working with files.

Chapter 5 introduces general statistical inference ideas. The sampling distributions of the sample mean and sample variance are presented through simulation and analytic examples, illustrating the central limit theorem and related results. Then general concepts of statistical estimation are explored, including basic examples of the method of moments and maximum likelihood estimation, followed by simple confidence bounds. Basic notions of statistical hypothesis testing are introduced, and finally the chapter is closed by touching basic ideas of Bayesian statistics.

Chapter 6 covers a variety of practical confidence intervals for both one and two samples. The chapter starts with standard confidence intervals for means, and then progresses to the more modern bootstrap method and prediction intervals. The chapter also serves as an entry point for investigating the effects of model assumptions on inference.

Chapter 7 focuses on hypothesis testing. The chapter begins with standard t-tests for population means, and then covers hypothesis tests for the comparison of two means. Then, Analysis of Variance (ANOVA) is covered, along with hypothesis tests for checking independence and goodness of fit. The reader is then introduced to power curves. The chapter closes by touching on a seldom looked at property, the distribution of the p-value.

Chapter 8 covers least squares and statistical linear regression models. It begins by covering least squares and then moves onto the linear regression statistical model, including hypothesis tests and confidence bands. Additional concepts of regression are also explored. These include assumption checking, model selection, interactions and more.

Chapter 9 provides an overview of several more advanced machine learning concepts. At onset, the machine learning paradigm of investigating data is introduced. This includes, training, validation and testing. Then the concept of bias and variance in the context of machine learning is introduced. This goes together with presenting ideas of regularization, applied to linear models. The chapter

then moves onto logistic regression and the generalized linear model. Then further supervised learning methods are introduced, including linear classification, random forests, support vector machines and deep neural networks. Then some unsupervised methods are introduced, including k-means and Principal Component Analysis (PCA). The chapter closes with a brief exploration of Markov decision processes and reinforcement learning.

Chapter 10 moves on to stochastic models in applied probability, giving the reader an indication of the strength of stochastic modeling and Monte-Carlo simulation. It focuses on dynamic systems, where Markov chains, discrete event simulation, and reliability analysis are explored, along with several aspects dealing with random number generation.

Appendix A contains a list of many useful items detailing "how to perform ... in Julia", where the reader is directed to specific code examples that detail directly with these items.

Appendix B lists additional language features of the Julia language that were not used by the code examples in this book.

Appendix C lists additional Julia packages dealing with statistics, machine learning, data science and artificial intelligence that were not used in this book.

Whether you are professional, a student, an educator, a researcher or an enthusiast, we hope that you find this book useful. We hope it can expand your knowledge in fundamentals of statistics with a view towards machine learning, artificial intelligence and data science. We further hope that the integration of Julia code and the content that we present help you quickly apply Julia for such purposes.

We would like to thank colleagues, family members and friends for their feedback, comments and suggestions. These include, Julianna Forbes, Milan Bouchet-Valat, Heidi Dixon, Jaco Du Plessis, Vaughan Evans, Liam Hodgkinson, Bogumił Kamiński, Dirk Kroese, Benoit Liquet, Ruth Luscombe, Geoff McLachlan, Moshe Nazarathy, Robert Salomone, Vincent Tam, Sérgio Bacelar, Alex Stenlake, James Tanton and others. In particular, we thank Vektor Dewanto for detailed feedback, and for catching dozens of typos and errors. Yoni Nazarathy would also like like to acknowledge the Australian Research Council (ARC) for supporting part of this work via Discovery Project grant DP180101602.

Hayden Klok and Yoni Nazarathy.

Contents

Chapter 1

Introducing Julia - DRAFT

Programming goes hand in hand with mathematics, statistics, data science and many other fields. Scientists, engineers, data scientists and statisticians often need to automate computation that would otherwise take too long or be infeasible to carry out. This is for the purpose of prediction, planning, analysis, design, control, visualization or as an aid for theoretical research. Often, general programming languages such as Fortran, C/C++, Java, Swift, C#, Go, JavaScript or Python are used. In other cases, more mathematical/statistical programming languages such as Mathematica, Matlab/Octave, R, or Maple are employed. The process typically involves analyzing the problem at hand, writing code, analyzing behavior and output, re-factoring, iterating and improving the model. At the end of the day, a critical component is speed, specifically, the speed it takes to reach a solution - whatever it may be.

When trying to quantify speed, the answer is not simple. On the one hand, speed can be quantified in terms of how fast a piece of computer code runs, namely *runtime speed*. On the other hand, speed can be quantified in terms of how fast it takes to code, debug and re-factor computer code, namely *development speed*. Within the realm of *scientific computing* and *statistical computing*, compiled low-level languages such as Fortran, C/C++ and the like generally yield fast runtime performance, however require more care in creation of the code. Hence they are generally fast in terms of runtime, yet slow in terms of development time. On the opposite side of the spectrum are mathematically specialized languages such as Mathematica, R, Matlab as well as Python. These typically allow for more flexibility when creating code, hence generally yield quicker development times. However, runtimes are typically significantly slower than what can be achieved with a low-level language. In fact, many of the efficient statistical and scientific computing packages incorporated in these languages are written in low-level languages, such as Fortran or C/C++, which allow for faster runtimes when applied as closed modules.

A practitioner wanting to use a computer for statistical and mathematical analysis often faces a trade-off between runtime and development time. While speed (both development and runtime) is hard to fully and fairly quantify, Figure 1.1 illustrates a schematic view showing general speed trade-offs between languages. As is postulated by this figure, there is a type of a *Pareto optimal frontier* ranging from the C language on one end to the R language on the other. The location of each language on this figure cannot be determined exactly. However, few would disagree that "R is generally faster to code than C" and "C generally runs faster than R". So, what about Julia?

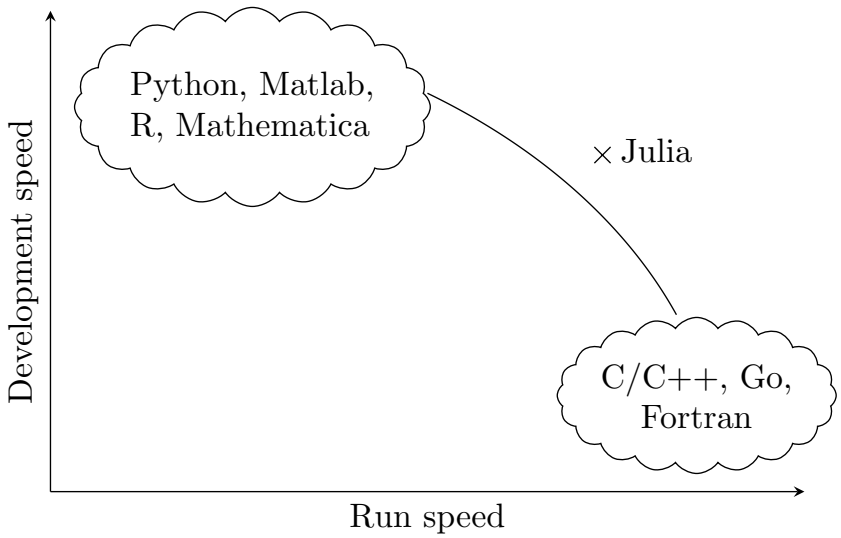

Figure 1.1: A schematic of run speed vs. development speed.
Observe the Pareto-optimal frontier existing prior to Julia.

The *Julia language and framework* developed in the last several years makes use of a variety of advances in compilation, computer languages, scientific computation and performance optimization. It is a language designed with a view of improving on the previous Pareto-optimal frontier depicted in Figure 1.1. With syntax and style somewhat similar to R, Python and Matlab/Octave, and with performance comparable to that of C/C++ and Fortran, Julia attempts to break the so called *two-language problem*. That is, it is postulated that practitioners may quickly create code in Julia, which also runs quickly. Further, re-factoring, improving, iterating and optimizing code can be done in Julia, and does not require the code to be ported to C/C++ or Fortran, since the Julia standard libraries, and almost all of Julia base are written in Julia.

Following this discussion about development speed and runtime speed, we make a rather sharp turn. We focus on *learning speed*. In this context, we focus on learning how to use Julia and in the same process learning and/or strengthening statistical knowledge. In this respect, with the exception of some minor discussions in Section 1.1, "runtime speed and performance" is seldom mentioned in the book. It is rather axiomatically obtained by using Julia. Similarly, coding and complex project development speed is not our focus. Again, the fact that Julia feels like a high-level language, very similar to Python, immediately suggests it is practical to code complex projects quickly in the language. Our focus is on learning quickly.

By following the code examples in this book (there are over 175), we allow you to learn how to use the basics of Julia quickly and efficiently. In the same go, we believe that this book will strengthen your understanding of statistics. In fact, the book contains a self contained overview of elementary probability and statistics, taking the reader through a tour of many concepts, illustrated via Julia code examples. Even if you are a seasoned statistician, data-scientist or probabilist, we are confident that you will find some of our discussions and examples interesting and gain further insight into statistics, machine learning, artificial intelligence and data science as you explore the basics of Julia.

Question: Do I need to have any statistics or probability knowledge to read this book?
Answer: Statistics or probability knowledge is not pre-assumed, however some general mathematics knowledge is assumed. Hence, this book is also a self-contained guide for the core principles of probability and statistics. It is ideally suited for a data-scientist wishing to strengthen their core probability and statistics knowledge while exploring the Julia language.

Question: What experience in programming is needed in-order to use this book?
Answer: While this book is not an introductory programming book, it does not assume that the reader is a professional software developer. Any reader that has coded in some other language (even if only minimally) will be able to follow the code examples in this book and their descriptions.

Question: How to read the book?
Answer: You may either read the book sequentially, or explore ideas and code examples in an ad-hoc manner. In any case, feel free to use the code-repository on GitHub:

`https://github.com/h-Klok/StatsWithJuliaBook`

As you do so, you may want to modify the code in the examples to experiment with various aspects of the statistical phenomena being presented. You may often modify numerical parameters and see what effect your modification has on the output. You may also find the "How-to in Julia" index (Appendix A) useful. This index (also available online) directs you to individual code listings that contain specific examples of "how to".

The remainder of this chapter is structured as follows: In Section 1.1 we present a brief overview of the Julia language. In Section 1.2, we describe some options for setting up a Julia working environment presenting the REPL and JuliaBox. Then in Section 1.3 we dive into Julia code examples designed to highlight basic powerful language features. We continue in Section 1.4 where we present code examples for plotting and graphics. Then in Section 1.5 we overview random number generation and the Monte Carlo method, used throughout the book. We close with Section 1.6 where we illustrate how other languages such as Python, R and C can be easily integrated with your Julia code.

If you are a newcomer to statistics or data-science, then it is possible that many of the examples covered in the first chapter are based on ideas that you have not previously touched. The purpose of the examples is to illustrate key aspects of the Julia language in this context. Hence, if you find the examples of the first chapter overwhelming, feel free to advance to the next chapter where probability is introduced from first principles. The content builds up from there gradually.

1.1 Language Overview

We now embark on a very quick tour of Julia. We start by overviewing language features in broad terms and continue with several basic code examples. This section is in no way a comprehensive description of the programming language and its features. Rather, it aims to overview a few select language features and introduce minimal basics. As a Julia learning resource, the reader may use this book by following through its examples, beginning in Section 1.3, and continuing through to Chapter 10.

About Julia

Julia is first and foremost a *scientific programming language*. It is perfectly suited for statistics, machine learning, data science, as well as for light and heavy numerical computational tasks. It can also be integrated in user-level applications, however one would not typically use it for front-end interfaces, or game creation. It is an open-source language and platform, and the Julia community brings together contributors from the scientific computing, statistics, and data-science worlds. This puts the Julia language and package system in a good place for combining mainstream statistical methods with methods and trends of the scientific computing world. Coupled with programmatic simplicity similar to Python, and with speed similar to C, Julia is taking an active part of the data-science revolution. In fact, some believe it may overtake Python and R to become the primary language of data-science in the future. Visit `https://julialang.org/` for more details.

We now discuss a few of the languages main features. If you are relativity new to programming, you may want to skip this discussion, and move to the subsection below which deals with a few basic commands. A key distinction between Julia and other high-level scientific computing languages is that Julia is *strongly typed*. This means that every variable or object has a distinct type that can either explicitly or implicitly be defined by the programmer. This allows the Julia system to work efficiently and integrates well with Julia's *just-in-time (JIT)* compiler. However, in contrast to low level strongly-typed languages, Julia alleviates the user from having to be "type-aware" whenever possible. In fact, many of the code examples in this book, do not explicitly specify types. That is, Julia features *optional typing*, and when coupled with Julia's *multiple dispatch* and *type inference*, Julia's JIT compilation system creates fast running code (compiled to *LLVM*), that is also very easy to program and understand.

The core Julia language imposes very little, and in fact the standard Julia libraries, and almost all of Julia `Base`, is written in Julia itself. Even primitive operations such as integer arithmetic are written in Julia. The language features a variety of additional packages, some of which are used in this book. All of these packages, including the language and system itself, are free and open source (MIT licensed). There are dozens of features of the language that can be mentioned. While it is possible, there is no need to vectorize code for performance. There is efficient support for *Unicode*, including but not limited to UTF-8. C can be called directly from Julia. There are even Lisp-like macros, and other metaprogramming facilities.

Julia development started in 2009 by Jeff Bezanson, Stefan Karpinski, Viral Shah and Alan Edelman. The language was launched in 2012 and has grown significantly since then, with the current version 1.3 as of the end of 2019. While the language and implementation are open source,

the commercial company *Julia Computing* provides services and support for schools, universities, business and enterprises that wish to use Julia. One of their services is *JuliaBox* which allows one to run Julia via *Jupyter* notebooks remotely.

A Few Basic Commands

Julia is a complete programming language supporting various programming paradigms including *procedural programming*, *object oriented programming*, *meta-programming* and *functional programming*. It is useful for *numerical computations*, *data processing*, *visualization*, *parallel computing*, *network input and output*, and much more.

As with any programming language you need to start somewhere. We start with an extended "Hello world". Look at the code listing below, and the output that follows. If you've programmed previously, you can probably figure out what each of the code lines do. We've also added a few comments to this code example, using #. Read the code below, and look at the output that follows:

Listing 1.1: Hello world and perfect squares

```
println("There is more than one way to say hello:")

# This is an array consisting of three strings
helloArray = ["Hello","G'day","Shalom"]

for i in 1:3
    println("\t", helloArray[i], " World!")
end

println("\nThese squares are just perfect:")

# This construct is called a 'comprehension' (or 'list comprehension')
squares = [i^2 for i in 0:10]

# You can loop on elements of arrays without having to use indexing
for s in squares
    print("  ",s)
end

# The last line of every code snippet is also evaluated as output (in addition to
# any figures and printing output generated previously).
sqrt.(squares)
```

```
There is more than one way to say hello:
        Hello World!
        G'day World!
        Shalom World!

These squares are just perfect:
  0  1  4  9  16  25  36  49  64  81  100
11-element Array{Float64,1}:
  0.0
  1.0
  2.0
  3.0
  4.0
```

```
 5.0
 6.0
 7.0
 8.0
 9.0
10.0
```

Most of the book contains code listings such as Listing 1.1 above. For brevity, we generally omit comments from code examples. Instead most listings are followed by minor comments as seen below.

The `println()` function is used for strings such as `"There is...hello:"`. In line 4 we define an array consisting of 3 strings. The `for` *loop* in lines 6-8 executes three times, with the variable `i` incremented on each iteration. Line 7, is the body of the loop where `println()` is used to print several arguments. The first, `"\t"` is a tab spacing. The second is the `i`-th entry of `helloArray` (in Julia array indexing begins with index 1), and the third is an additional string. In line 10 the `"\n"` character is used within the string to signify printing a new line. In line 13, a *comprehension* is defined. It consists of the elements, $\{i^2 \ : \ i \in \{0,\ldots,10\}\}$. We cover comprehensions further in Listing 1.2. Lines 16-18 illustrate that loops may be performed on all elements of an array. In this case, the loop changes the value of the variable `s` to another value of the array `squares` in each iteration. Note the use of the `print()` function to print without a newline. Line 22, the last line of the code block applies the `sqrt()` function on each element of the array `squares` by using the '`.`' broadcast operator. The expression of the last line of every code block, unless terminated by a "`;`", is presented as output. In this case, it is an `11-element` array of the numbers $0,\ldots,10$. The type of the output expression is also presented. It is `Array{Float64,1}`.

When exploring statistics and other forms of numerical computation, it is often useful to use a *comprehension* as a basic programming construct. As explained above, a typical form of a comprehension is:

```
[f(x) for x in A]
```

Here, `A` is some array, or more generally a collection of objects. Such a comprehension creates an array of elements, where each element `x` of `A` is transformed via `f(x)`. Comprehensions are ubiquitous in the code examples we present in this book. We often use them due to their expressiveness and simplicity. We now present a simple additional example:

Listing 1.2: Using a comprehension

```
1  array1 = [(2n+1)^2 for n in 1:5]
2  array2 = [sqrt(i) for i in array1]
3  println(typeof(1:5), "  ", typeof(array1), "  ", typeof(array2))
4  1:5, array1, array2
```

```
UnitRange{Int64}  Array{Int64,1}  Array{Float64,1}
(1:5, [9, 25, 49, 81, 121], [3.0, 5.0, 7.0, 9.0, 11.0])
```

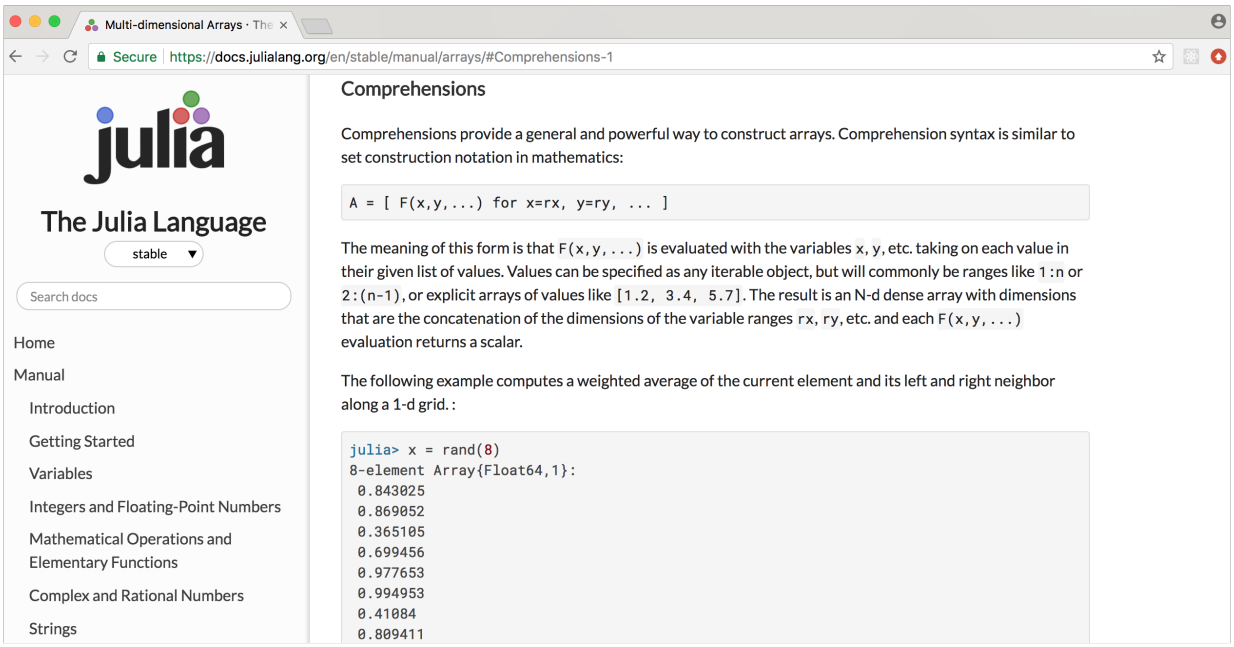

Figure 1.2: Visit `https://docs.julialang.org` for official language documentation.

The array `array1`, is created in line 1 with the elements $\{(2n+1)^2 \ : \ n \in \{1,\ldots,5\}\}$, in order. Note that while mathematical sets are not ordered, comprehensions generate ordered arrays. Observe the literal `2` in the multiplication `2n`, without explicit use of the `*` symbol. In the next line, `array2` is created. An alternative would be to use `sqrt.(array1)`. In line 3, we print the `typeof()` three expressions. The type of `1:5` (used to create `array1`) is a `UnitRange` of `Int64`. It is a special type of object that encodes the integers $1,\ldots,5$ without explicitly allocating memory. Then the types of both `array1` and `array2` are `Array` types, and they contain values of types `Int64` and `Float64` respectively. In line 4, a tuple of values is created through the use of a comma between `1:5`, `array1` and `array2`. As it is the last line of the code, it is printed as output. Observe that in the output, the values of the second element of the tuple are printed as integers (no decimal point) while the values of the third element are printed as floating point numbers.

Getting Help

You may consult the official Julia documentation, `https://docs.julialang.org/` for help. The documentation strikes a balance between precision and readability. See Figure 1.2.

While using Julia, help may be obtained through the use of `?`. For example try, `?sqrt` and you will see output similar to Figure 1.3.

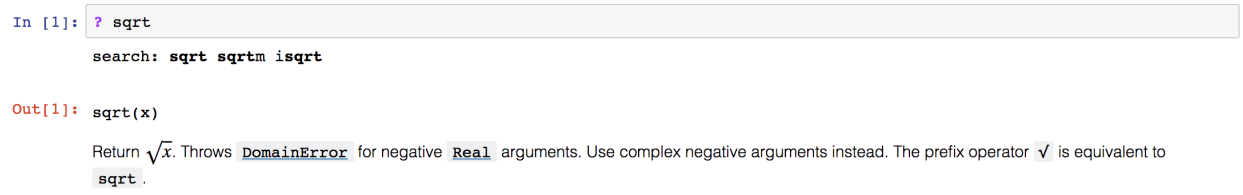

Figure 1.3: Snapshot from a Julia Jupyter notebook: Keying in `?sqrt` presents help for the `sqrt()` function.

You may also find it useful to apply the `methods()` function. Try, `methods(sqrt)`. You will see output that contains lines of this sort:

```
...
sqrt(x::Float32) at math.jl:426
sqrt(x::Float64) at math.jl:425
sqrt(z::Complex{#s45} where #s45<:AbstractFloat) at complex.jl:392
sqrt(z::Complex) at complex.jl:416
sqrt(x::Real) at math.jl:434
sqrt{T<:Number}(x::AbstractArray{T,N} where N) at deprecated.jl:56
...
```

This presents different *Julia methods* implementation for the function `sqrt()`. In Julia, a given function may be implemented in different ways depending on different input arguments with each different implementation being a *method*. This is called *multiple dispatch*. Here, the various methods of `sqrt()` are shown for different types of input arguments.

Runtime Speed and Performance

While Julia is fast and efficient, for most of this book we don't explicitly focus on runtime speed and performance. Rather, our aim is to help the reader learn how to use Julia while enhancing knowledge of probability and statistics. Nevertheless, we now briefly discuss runtime speed and performance.

From a user perspective, Julia feels like an *interpreted language* as opposed to a *compiled language*. With Julia, you are not required to explicitly compile your code before it is run. However, as you use Julia, behind the scenes, the system's JIT compiler compiles every new function and code snippet as it is needed. This often means that on a first execution of a function, runtime is much slower than the second, or subsequent runs. From a user perspective, this is apparent when using other packages (as the example in Listing 1.3 below illustrates, this is often done by the `using` command). On a first call (during a session) to the `using` command of a given package, you may sometimes wait a few seconds for the package to compile. However, afterwards, no such wait is needed.

For day to day statistics and scientific computing needs, you often don't need to give much thought to performance and run speed with Julia, since Julia is inherently fast. For instance, as we do in dozens of examples in this book, simple Monte Carlo simulations involving 10^6 random variables typically run in less than a second, and are very easy to code. However, as you progress

into more complicated projects, many repetitions of the same code block may merit profiling and optimization of the code in question. Hence, you may wish to carry out basic profiling.

For basic profiling of performance the `@time` *macro* is useful. Wrapping code blocks with it (via `begin` and `end`) causes Julia to profile the performance of the block. In Listings 1.3 and 1.4, we carry out such profiling. In both listings, we populate an array, called `data`, containing 10^6 values, where each value is a mean of 500 random numbers. Hence, both listings handle half a billion numbers. However, Listing 1.3 is a much slower implementation.

Listing 1.3: Slow code example

```
using Statistics

@time begin
        data = Float64[]
            for _ in 1:10^6
                group = Float64[]
                    for _ in 1:5*10^2
                        push!(group,rand())
                    end
                push!(data,mean(group))
            end
        println("98% of the means lie in the estimated range: ",
                        (quantile(data,0.01),quantile(data,0.99)) )
end
```

```
98% of the means lie in the estimated range: (0.4699623580817418, 0.5299937027991253)
 11.587458 seconds (10.00 M allocations: 8.034 GiB, 4.69% gc time)
```

The actual output of the code gives a range, in this case approximately 0.47 to 0.53 where 98% of the sample means (averages) lie. We cover more on this type of statistical analysis in the chapters that follow.

The second line of output, generated by `@time`, states that it took about 11.6 seconds for the code to execute. There is also further information indicating how many memory allocations took place, in this case about 10 million, totaling just over 8 Gigabytes (in other words, Julia writes a little bit, then clears, and repeats this process many times over). This constant read-write is what slows our processing time.

Now, look at Listing 1.4 and its output.

Listing 1.4: Fast code example

```
using Statistics

@time begin
        data = [mean(rand(5*10^2)) for _ in 1:10^6]
        println("98% of the means lie in the estimated range: ",
                        (quantile(data,0.01),quantile(data,0.99)) )
end
```

```
98% of the means lie in the estimated range: (0.469999864362845, 0.5300834606858865)
  1.705009 seconds (1.01 M allocations: 3.897 GiB, 10.76% gc time)
```

As can be seen, the output gives the same estimate for the interval containing 98% of the means. However, in terms of performance, the output of `@time` indicates that this code is clearly superior. It took about 1.7 seconds (compare with 11.6 seconds for Listing 1.3). In this case, the code is much faster because far fewer memory allocations are made. Note that '`gc time`' stands for "garbage collection" and quantifies what percentage of the running time Julia was busy with internal memory management.

Here are some comments for both code-listings 1.3 and 1.4:

In both listings we use the `Statistics` package, required for the `mean()` function. Line 4 (Listing 1.3) creates an empty array of type `Float64`, `data`. Line 6 creates an empty array, `group`. Then lines 7-9 loop 500 times, each time pushing to the array, `group`, a new random value generated from `rand()`. The `push!()` function here uses the naming convention of having an exclamation mark when the function modifies the argument. This is not part of the Julia language, but rather decorates the name of the function. In this case, it modifies `group` by appending another new element. Here is one point where the code is inefficient. The Julia compiler has no direct way of knowing how much memory to allocate for `group` initially, hence some of the calls to `push!()` imply reallocation of the array and copying. Line 10 is of a similar nature. The composition of `push!()` and `mean()` imply that the new mean (average of 500 values) is pushed into `data`. However, some of these calls to `push!()` imply a reallocation. At some point the allocated space of `data` will suddenly run out, and at this point the system will need to internally allocate new memory, and copy all values to the new location. This is a big cause of inefficiency in our example. Line 13 creates a tuple within `println()`, using `(,)`. The two elements of the tuple are return values from the `quantile()` function which compute the 0.01 and 0.99 quantiles of `data`. Quantiles are covered further in Chapter 4. The lines of Listing 1.4 are relatively simpler and in this case performance is better. All of the computation is carried out in the comprehension in Line 4, within the square brackets `[]`. Writing the code in this way allows the Julia compiler to pre-allocate 10^6 memory spaces for `data`. Then, applying `rand()` with an argument of `5*10^2`, indicating the number of desired random values, allows for faster operation. The functionality of `rand()` is covered in Section 1.5.

Julia is inherently fast, even if you don't give it much thought as a programmer. However, in order to create truly optimized code, one needs to understand the inner workings of the system a bit better. There are some general guidelines that you may follow. A key is to think about memory usage and allocation as in the examples above. Other issues involve allowing Julia to carry out type inference efficiently. Nevertheless, for simplicity, the majority of the code examples of this book ignore types as much as possible and don't focus on performance.

Types and Multiple Dispatch

Functions in Julia are invoked via *multiple dispatch*. This means the way a function is executed, i.e. its *method*, is based on the *type* of its inputs, i.e. its *argument* types. Indeed functions can have multiple methods of execution, which can be checked using the `methods()` command.

Julia has a powerful type system which allows for *user-defined-types*. One can check the type of a variable using the `typeof()` function, while the functions `subtype()` and `supertype()` return the *subtype* and *supertype* of a particular type respectively. As an example, `Bool` is a subtype of `Integer`, while `Real` is the supertype of Integer. This is illustrated in Figure 1.4, which shows the type hierarchy of numbers in Julia.

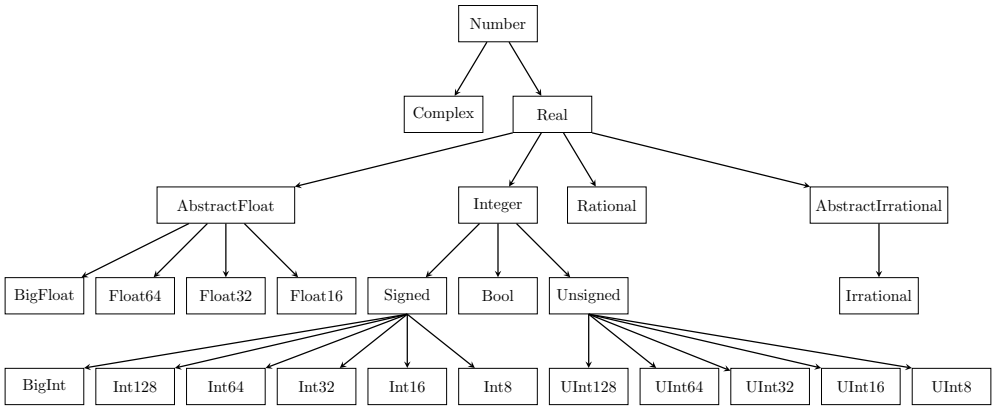

Figure 1.4: Type hierarchy for Julia numbers.

One aspect of Julia is that if the user does not specify all variable types in a given piece of code, Julia will attempt to infer what types the unspecified variables should be, and will then attempt to execute the code using these types. This is known as *type inference*, and relies on a type inference algorithm. This makes Julia somewhat forgiving when it comes to those new to coding, and also allows one to quickly mock-up fast working code. It should be noted however that if one wants the fastest possible code, then it is good to specify the types involved. This also helps to prevent *type instability* during code execution.

1.2 Setup and Interface

There are multiple ways to run Julia. Here we introduce two ways, the *REPL command line interface* and *Jupyter notebooks*. We first describe these two alternatives, and then describe the *package manager* which allows one to extend Julia's basic functionality by installing additional packages.

No matter how you run Julia, there is an instance of a Julia *kernel* running. The running kernel contains all of the currently compiled Julia functions, loaded packages, defined variables, and objects. You may even run multiple kernels, sometimes in a distributed manner.

REPL Command Line Interface

The *Read Evaluate Print Loop (REPL)* command line interface is a simple and straight forward way of using Julia. It can be downloaded directly from: `https://julialang.org/downloads/`. Downloading it implies downloading the Julia Kernel as well.

Once installed locally, Julia can be launched and the Julia REPL will appear, within which Julia commands can be entered and executed. For example, in Figure 1.5 the code `1+2` was entered, followed by the enter key. Note that if Julia is launched as its own stand alone application, a new Julia instance will appear. However, if you are working in a shell/command-line environment, the REPL can also be launched from within the current environment.

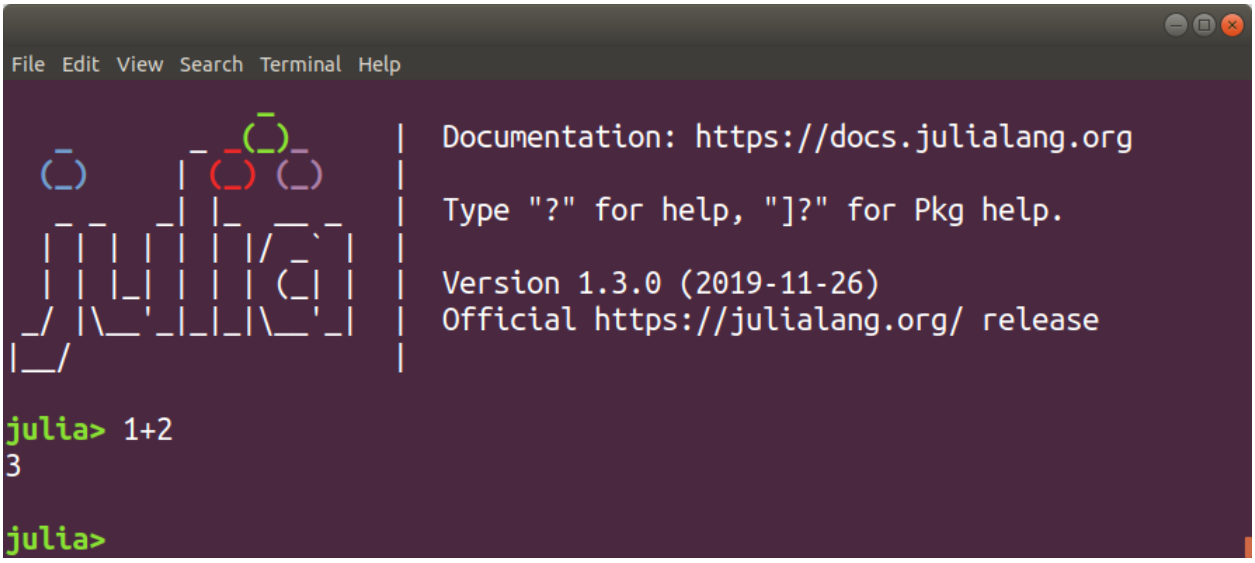

Figure 1.5: Julia's REPL interface.

When using the REPL, typically one will also work with Julia files which have the `.jl` extension. In fact, every code listing in this book is stored such a file. These files are available on the book's GitHub.

Jupyter Notebooks

An alternative to using the REPL is to use a *Jupyter Notebook* as presented in Figure 1.6. It is a browser based interface in which one can type and execute Julia code, as well as Python, R, and other languages. Jupyter notebooks are easy to use and allow one to combine code, output, visuals, and markdown formatting all together in one document. A Jupyter notebook is both a means of presentation and execution.

Each notebook consists of a series of cells, in which code can be typed and run. Cells can be of different type. *Code cells* allow Julia code to be entered and executed, while *markdown cells* allow for formatting of the document in *Markdown*, which is a simple formatting language that also incorporates hyperlinks, images, and *LaTeX formatting* for formulas.

Jupyter notebook files have the `.ipynb` extension. The content of notebooks can also be exported as PDF and other formats. A common way to run Jupyter for Julia is using the *Anaconda* Python distribution which installs a *Jupyter notebook server* locally. A technical note is that the `IJulia` (Julia) package is required for Julia to work within Jupyter notebooks. More on packages below.

Another advantage of Jupyter notebooks is that because they are browser based, they can be configured to run over a remote connection. In fact an on-line product by Julia Computing that allows you to use Julia with cloud storage and compute, and without requiring any local installation is *JuliaBox*, available at `https://juliabox.com/`. Note that JuliaBox is a paid service while running a Jupyter notebook on your own local machine is free.

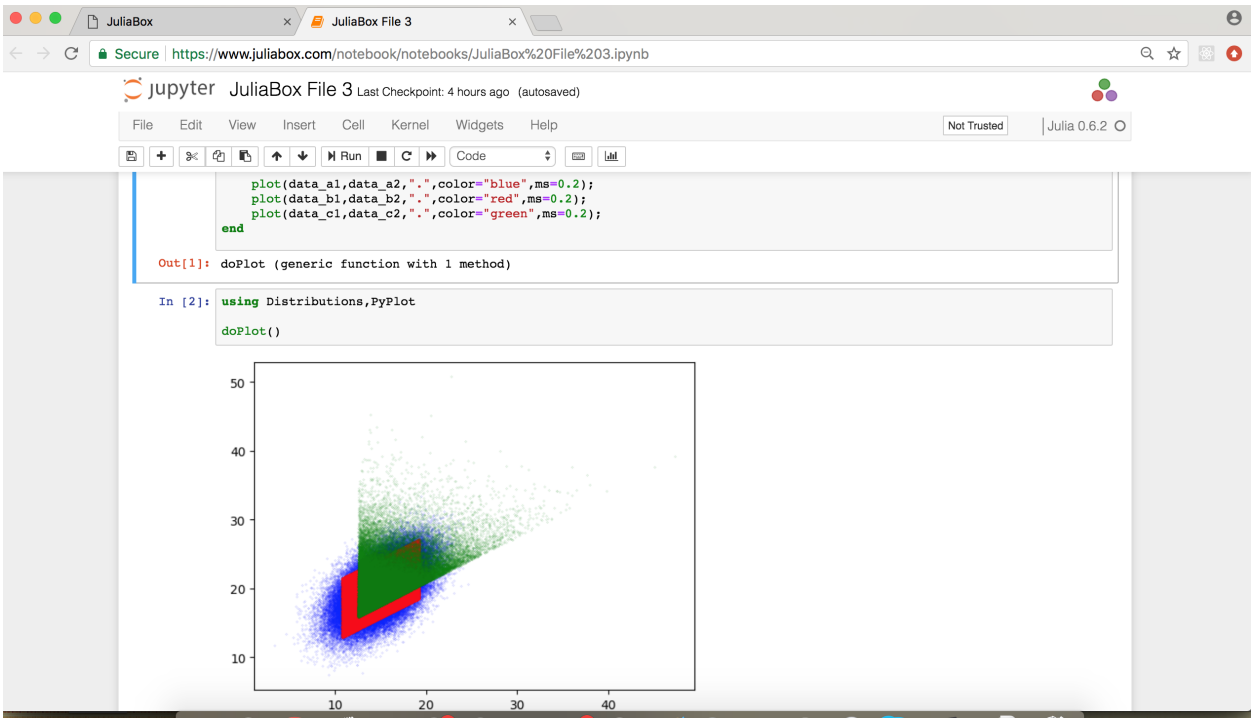

Figure 1.6: An example of a Jupyter notebook accessed via JuliaBox.

The user interface for using Jupyter notebooks is easy to learn. When starting, note that there are two input modes. *Edit mode* allows code/text to be entered into a cell, while *command mode* allows keyboard-activated actions, such as toggling line numbering, copying cells, and deleting cells. Cells can be executed by first selecting the cell and then pressing `ctrl-enter` or `shift-enter`. In command mode, additional cells can be created by pressing `a` or `b` to create cells above or below respectively.

The Package Manager

Although Julia comes with many built-in features, the core system can be extended. This is done by installing packages, which can be added to Julia at your discretion. This allows users to customize their Julia installation depending on their needs, and at the same time offers support for developers who wish to create their own packages, enriching the Julia ecosystem. Note that packages may be either *registered*, meaning that they are part of the Julia package repository, or *unregistered*, meaning they are not. A list of currently registered packages is available at: `https://pkg.julialang.org/`.

When using the REPL you can enter the *package manager mode* by typing "`]`". This mode can be exited by hitting the backspace key. In this mode, packages can be installed, updated, or removed. The following lists a few of the many useful commands available:

`] add Foo.jl`	adds package `Foo` to the current Julia build.
`] status`	lists what packages and versions are currently installed.

`] update` updates existing packages.

`] remove Foo.jl` removes package `Foo` from the current Julia build.

As you study the code examples in this book, you will notice that most start with the `using` command, followed by a package name. This is how Julia packages are loaded into the current namespace of the kernel, so that the package's functions, objects, and types can be used. Note that writing `using` does not imply installing a package. Installation of a package is a one-time operation which must be performed before the package can be used. In comparison, typing the keyword `using` is required every time functionality of a package is required.

Packages Used in This Book

The code in this book uses a variety of Julia packages. Some of the key packages used in the context of probability, statistics and machine learning are `DataFrames`, `Distributions`, `Flux`, `GLM`, `Plots`, `Random`, `Statistics`, `StatsBase`, and `StatsPlots` as well as many other important packages. Some of these are built-in with the base installation, for example `Statistics` and `Random`, while others require user installation via the package manager as described above. A short description of each of the packages that we use in the book is contained below.

`Calculus.jl` provides tools for working with basic calculus operations including differentiation and integration both numerically and symbolically.

`CategoricalArrays.jl` provides tools for working with categorical variables.

`Clustering.jl` provides support for various clustering algorithms.

`Combinatorics.jl` is a combinatorics library focusing mostly on enumerative combinatorics and permutations.

`CSV.jl` is a utility library for working with CSV and other delimited files in Julia.

`DataFrames.jl` is a package for working with tabular data.

`DataStructures.jl` provides support for various types of data structures.

`Dates.jl` is one of Julia's standard libraries and provides support for working with dates and times.

`DecisionTree.jl` is a package for decision trees and random forest algorithms.

`DifferentialEquations.jl` is a suite which provides efficient Julia implementations of numerical solvers for various types of differential equations.

`Distributions.jl` provides support for working with probability distributions and associated functions.

`Flux.jl` is a machine learning library written in pure Julia.

`GLM.jl` is a package on linear models and generalized linear models.

HCubature.jl is an implementation of multidimensional "h-adaptive" (numerical) integration in Julia.

HypothesisTests.jl implements a wide range of hypothesis tests and confidence intervals.

HTTP.jl provides HTTP client and server functionality.

IJulia.jl is required to interface Julia with Jupyter notebooks.

Images.jl is an image processing library.

JSON.jl is a package for parsing and printing JSON.

KernelDensity.jl is a kernel density estimation package.

LaTeXStrings.jl makes it easier to type LaTeX equations in string literals.

LIBSVM.jl is a package for Support Vector Machines (SVM) using LIBSVM, a general library for SVM.

LightGraphs.jl provides support for the implementation of graphs in Julia.

LinearAlgebra.jl is one of Julia's standard libraries, and provides linear algebra support.

Measures.jl allows building up and representing expressions involving differing types of units that are then evaluated, resolving them into absolute units.

MultivariateStats.jl is a package for multivariate statistics and data analysis, including ridge regression, PCA, dimensionality reduction and more.

NLsolve.jl provides methods to solve non-linear systems of equations in Julia.

Plots.jl is one of the main plotting packages in the Julia ecosystem. It is the main plotting package used throughout our book.

PyCall.jl provides the ability to directly call and fully interoperate with Python from the Julia language.

PyPlot.jl provides a Julia interface to the Matplotlib plotting library from Python, and specifically to the matplotlib.pyplot module.

QuadGK.jl provides support for one-dimensional numerical integration using adaptive Gauss-Kronrod quadrature.

Random.jl is one of Julia's standard libraries. It provides support for pseudo random number generation.

RCall.jl provides several different ways of interfacing with R from Julia.

RDatasets.jl provides an easy way to interface with the standard datasets that are available in the core of the R language, as well as several datasets included in many of R's more popular packages.

Roots.jl contains simple routines for finding roots of continuous scalar functions of a single real variable.

`SpecialFunctions.jl` contains various special mathematical functions, such as Bessel, zeta, digamma, along with sine and cosine integrals, as well as others.

`Statistics.jl` is one of Julia's standard libraries. It contains functionality for common statistics functions including mean, standard deviation and quantile.

`StatsBase.jl` provides basic support for statistics by implementing a variety of statistics-related functions, such as scalar statistics, high-order moment computation, counting, ranking, covariances, sampling and cumulative distribution function estimation.

`StatsPlots.jl` provides extensive statistical plotting recipes.

`TimeSeries.jl` provides support for working with time series data.

We are grateful to the dozens of developers that have contributed to, and are continuously improving these great Julia open-source packages. You may visit the GitHub pages for these packages and show your support. Many additional useful packages, not employed in our code examples are in Appendix C.

1.3 Crash Course by Example

Almost every procedural programming language needs functions, conditional statements, loops and arrays. Similarly, every scientific programming language needs to support plotting, matrix manipulations, and floating point calculations. Julia is no different. In this section we present several examples, and through them begin to explore various basic programming elements. Each example aims to introduce another aspect of Julia. These examples are not necessarily minimal examples needed for learning the basics of Julia, nor do they build statistical foundations from the ground up. Rather, they are designed to show what can be done with Julia. Hence if you find these examples too complex from either a programming or a mathematical perspective, feel free to skip directly to Chapter 2, where basic probability is demonstrated via simple examples from the ground up.

Alternatively, if you prefer to engage with the language through more simple examples, you may wish to use other resources alongside this book. If you are a beginner to programming, we recommend the introductory book to programming with the Julia language, "Think Julia – How to Think Like a Computer Scientist" by A. Downey, B. Lauwens [DL19]. If you are a seasoned programmer and are looking for a more general purpose text about Julia, see "Julia 1.0 Programming Cookbook" by B. Kamiński, P. Szufel [KS18]. Another option is to visit `https://julialang.org/learning/` for a variety of other resources.

In addition to the general Julia programming resources mentioned above, there are also several other texts that are worth considering for specific aspects of scientific computing, data science and artificial intelligence. The book [KW19] provides an exhaustive introduction to *optimization algorithms* together with Julia code. The book, [Kwon18] focuses on *operations research* using Julia. Finally, the book [MP2018] is an applied *data science* resource, as is [Vou16].

We now present some select examples which are designed to illustrate basic programming (bubble sort), show simple numerical computation (roots of a polynomial), provide examples of how to work

with matrices and randomness (Markov chain), and show how one can interface with the web and do basic text processing.

Bubble Sort

In our first example, we construct a basic sorting algorithm using first principles. The algorithm we consider here is called *Bubble Sort*. This algorithm takes an input *array*, indexed $1, \ldots, n$, then sorts the elements smallest to largest by allowing the larger elements, or "bubbles", to "percolate up". The algorithm is implemented in Listing 1.5. As can be seen from the code, the locations j and $j+1$ are swapped inside the two *nested loops*. This maintains an increasing (non-decreasing) order in the array. The *conditional statement* `if` is used to check if the numbers at indexes j and $j+1$ are in increasing order, and if needed, swap them.

Listing 1.5: Bubble sort

```
function bubbleSort!(a)
    n = length(a)
    for i in 1:n-1
        for j in 1:n-i
            if a[j] > a[j+1]
                a[j], a[j+1] = a[j+1], a[j]
            end
        end
    end
    return a
end

data = [65, 51, 32, 12, 23, 84, 68, 1]
bubbleSort!(data)
```

```
8-element Array{Int64,1}:
1
12
23
32
51
65
68
84
```

In lines 1-11, we define a *function*, named `bubbleSort!()`. The input argument `a` is implicitly expected to be an array. The function sorts `a` in place, and returns a reference to the array. Note that in this case, the function name ends with '!' by convention. This exclamation mark decorates the name of the function, letting us know that the function argument, `a`, will be modified (`a` is sorted in place without memory copying). In Julia, arrays are *passed by reference*. Arrays are indexed from `1` to the length of the array, obtained by `length()`. In line 6 the elements `a[j]` and `a[j+1]` are swapped by using assignment of the form `m,n = x,y` which is syntactic shorthand for `m=x` followed by `n=y`. In line 14, the function is called on `data`. As it is the last line of the code block and is not followed by a '`;`', the expression evaluated in that line is presented as output, in our case the sorted array. Note that it has a type `Array{Int64,1}`, meaning an array of integers. Julia inferred this type automatically. Try changing some of the values in line 13 to floating points, eg. `[65.0, 51.0 ...` (etc)`]` and see how the output changes.

Keep in mind that Julia already contains standard sorting functions such as `sort()` and `sort!()`, so you don't need to implement your own sorting function as we did. For more information on these functions use `?sort`. Also, the bubble sort algorithm is not the most efficient sorting algorithm, but is introduced here as a means of understanding Julia better. For an input array of length n, it will execute line 5 about $n^2/2$ times. For non-small n, this is much slower performance than optimal sorting algorithms where the number of comparisons can be reduced to an order of $n\log(n)$ times.

Roots of a Polynomial

We now consider a different type of programming example that comes from elementary numerical analysis. Consider the polynomial,

$$f(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0,$$

with real-valued coefficients $a_0, \ldots, a_n$. Say we wish to find all x values that solve the equation $f(x) = 0$. We can do this numerically with Julia using the `find_zeros()` function from the `Roots` package. This general purpose solver takes a function as input and numerically tries to find all its roots within some domain. As an example, consider the quadratic polynomial,

$$f(x) = -10x^2 + 3x + 1.$$

Ideally, we would like to supply the `find_zeros()` function with the coefficient values, -10, 3 and 1. However, `find_zeros()` is not designed for a specific polynomial, but rather for any Julia function that represents a real mathematical function. Hence one way to handle this is to define a Julia function specifically for this quadratic $f(x)$ and give it as an argument to `find_zeros()`. However, here we will take this one step further, and create a slightly more general solution. We first create a function called `polynomialGenerator` which takes a list of arguments representing the coefficients, $a_n, a_{n-1}, \ldots, a_0$ and returns the corresponding polynomial function. We then use this function as an argument to the `find_zeros()` function, which then returns the roots of the original polynomial.

Listing 1.6 shows our approach. Note that for our example it is straightforward to solve the roots analytically and verify the code. This is done using the quadratic formula as follows,

$$x = \frac{-3 \pm \sqrt{3^2 - 4(-10)}}{2(-10)} = \frac{3 \pm 7}{20} \qquad \Rightarrow \qquad x_1 = 0.5, \quad x_2 = -0.2.$$

Listing 1.6: Roots of a polynomial

```
1  using Roots
2
3  function polynomialGenerator(a...)
4      n = length(a)-1
5      poly =  function(x)
6                  return sum([a[i+1]*x^i for i in 0:n])
7              end
8      return poly
9  end
10
11 polynomial = polynomialGenerator(1,3,-10)
12 zeroVals = find_zeros(polynomial,-10,10)
13 println("Zeros of the function f(x): ", zeroVals)
```

```
Zeros of the function f(x): [-0.2, 0.5]
```

In line 1 we employ the `using` keyword, indicating to include elements from the package `Roots`. Note that this assumes that the package has already been added as part of the Julia configuration. Lines 3-9 define the function `polynomialGenerator()`. An argument, `a`, along with the *splat operator* `...` indicates that the function will accept a comma separated list of parameters of unspecified length. For our example we have three coefficients, specified in line 11. Line 4 makes use of the `length()` function to determine how many arguments were given to the function `polynomialGenerator()`. Notice that the degree of the polynomial, represented in the local variable `n` is one less than the number of arguments. Lines 5-7 define an internal function with an input argument `x`, and then stores this function as the variable `poly`, returned from `polynomialGenerator()`. One can pass functions as arguments, and assign them to variables. The main workhorse of this function is line 6, where the `sum()` function is used to sum over an array of values. This array is implicitly defined using a *comprehension*. In this case, the comprehension is `[a[i+1]*x^i for i in 0:n]`. This creates an array of length $n+1$ where the i'th element of the array is `a[i+1]*x^i`. In line 12 the `find_zeros()` function from the `Roots` package is used to find the roots of the polynomial. The latter arguments are guesses for the roots which are used for initialization. The roots calculated are then assigned to `zeroVals` and the output printed.

Steady State of a Markov Chain

We now introduce some basic linear algebra computations and simulation through a simple *Markov chain* example. Consider a theoretical city, where the weather is described by three possible states: (1) 'Fine', (2) 'Cloudy' and (3) 'Rain'. On each day, given a certain state, there is a probability distribution for the weather state of the next day. This simplistic weather model constitutes a *discrete time* (homogeneous) Markov chain. This Markov chain can be described by the 3×3 *transition probability matrix*, P, where the entry $P_{i,j}$ indicates the probability of transitioning to state j given that the current state is i. The transition probabilities are illustrated in Figure 1.7.

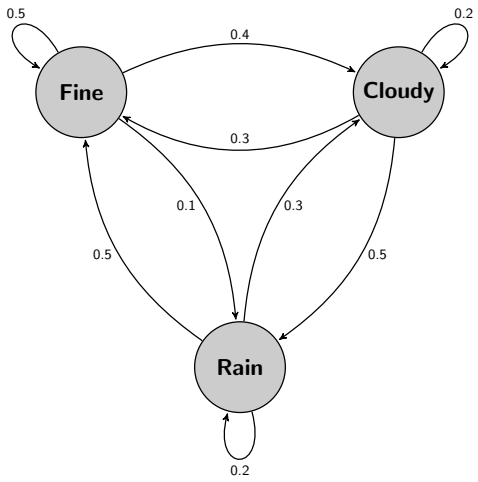

Figure 1.7: Three-state Markov chain of the weather.
Notice the sum of the arrows leaving each state is 1.

One important computable quantity for such a model is the long term proportion of occupancy in each state. That is, in steady state, what proportion of the time is the weather in state 1, 2 or 3. Obtaining this *stationary distribution*, denoted by the vector $\pi = [\pi_1, \pi_2, \pi_3]$ (or an approximation for it) can be achieved in several ways, as shown in Listing 1.7. For pedagogical and exploratory reasons we use four methods to find the stationary distribution. Note that some of these methods involve linear algebra and/or results from the theory of Markov chains. These are not covered here, but rather discussed in Section 10.2 of Chapter 10. If you haven't been exposed to linear algebra, we suggest you only skim through this example. The four methods that we use are:

1. By raising the matrix P to a high power, (repeated matrix multiplication of P with itself), the limiting distribution is obtained in any row. Mathematically,

$$\pi_i = \lim_{n\to\infty} [P^n]_{j,i} \qquad \text{for any index, } j. \tag{1.1}$$

2. We solve the (overdetermined) linear system of equations,

$$\pi P = \pi \qquad \text{and} \qquad \sum_{i=1}^{3} \pi_i = 1. \tag{1.2}$$

   This linear system of equations can be reorganized into a system with 3 equations and 3 unknowns by realizing that one of the equations inside $\pi P = \pi$ is redundant. Written out explicitly we have:

$$\begin{bmatrix} P_{11}-1 & P_{21} & P_{31} \\ P_{12} & P_{22}-1 & P_{32} \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \pi_1 \\ \pi_2 \\ \pi_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}. \tag{1.3}$$

3. By making use of the *Perron Frobenius Theorem* which implies the eigenvector corresponding to the eigenvalue of maximal magnitude is proportional to π, we find this eigenvector and normalize it by the sum of probabilities (L_1 norm).

4. We run a simple Monte Carlo simulation (see also Section 1.5) by generating random values of the weather according to P, and take the long term proportions of each state. In contrast to the previous three approaches, this approach does not require any linear algebra.

The output shows that the four estimates of the vector π are very similar. Each column represents the stationary distribution obtained from methods 1 to 4, while the rows represent the stationary probability of being in each state.

Listing 1.7: Steady state of a Markov chain in several ways

```
using LinearAlgebra, StatsBase

# Transition probability matrix
P = [0.5 0.4 0.1;
     0.3 0.2 0.5;
     0.5 0.3 0.2]

# First way
piProb1 = (P^100)[1,:]

# Second way
A = vcat((P' - I)[1:2,:],ones(3)')
b = [0 0 1]'
piProb2 = A\b

# Third way
eigVecs = eigvecs(copy(P'))
highestVec = eigVecs[:,findmax(abs.(eigvals(P)))[2]]
piProb3 = Array{Float64}(highestVec)/norm(highestVec,1)

# Fourth way
numInState = zeros(Int,3)
state = 1
N = 10^6
for t in 1:N
    numInState[state] += 1
    global state = sample(1:3,weights(P[state,:]))
end
piProb4 = numInState/N

[piProb1 piProb2 piProb3 piProb4]
```

```
3x4 Array{Float64,2}:
0.4375  0.4375  0.4375  0.437521
0.3125  0.3125  0.3125  0.312079
0.25    0.25    0.25    0.2504
```

In lines 4-6 the transition probability matrix P is defined. The notation for explicitly defining a matrix in Julia is the same as that of Matlab. In line 9, (1.1) is implemented and n is taken as 100 (approximating ∞). The first row of the resulting matrix is obtained via `[1,:]`. Note that using `[2,:]` or `[3,:]` instead will approximately yield the same result, since the limit in equation (1.1) is independent of j. Lines 12-14 use quite a lot of matrix operations to set up the system of equations (1.3). The use of `vcat()` (*vertical concatenation*) creates the matrix on the left hand side by concatenating the 2×3 matrix, `(P' - I)[1:2,:]` with a row vector of 1's, `ones(3)'`. Note the use of `I` which is the identity matrix. Finally, the solution is found by using `A\b` in the same fashion as Matlab for solving linear equations of the form $Ax = b$. In lines 17-19 the built-in `eigvecs()` and `eigvals()` functions from `LinearAlgebra` are used to find the eigenvalues and a set of eigenvectors of P respectively. The `findmax()` function is then used to find the index matching the eigenvalue with the largest magnitude. Note that the absolute value function `abs()` works on complex values as well. Also note that when normalizing in line 19, we use the L_1 norm which is essentially the sum of absolute values of the vector. In lines 22-29 a direct Monte Carlo simulation of the Markov chain is carried out through a million iterations and modifications of the `state` variable. We accumulate the occurrences of each state in line 26. Line 27 is the actual transition, which uses the `sample()` function from the `StatsBase` package. At each iteration the next state is randomly chosen based on the probability distribution given the current state. This is done via the use of weight vector, covered later in Section 3.4. Note that the normalization from counts to frequency in line 29, uses the fact that Julia casts integer counts to floating point numbers upon division. That is, both the variables `numInState` and `N` are an array of integers and an integer respectively, but the division (vector by scalar) makes `piProb4` a floating point array.

Web Interfacing, JSON and String Processing

We now look at a different type of example which deals with text. Imagine that we wish to analyze the writings of Shakespeare. In particular, we wish to look at the occurrences of some common words in all of his known texts and present a count of a few of the most common words. One simple and crude way to do this is to pre-specify a list of words to count, and then specify how many of these words we wish to present.

To add another dimension to this problem we will use a JSON (*Java Script Object Notation*) file. This file format is widely used for storing hierarchical datasets both in data science and web development, hence the name. We use the below JSON file in the example that follows.

```
{
  "words": [ "heaven","hell","man","woman","boy","girl","king","queen",
        "prince","sir","love","hate","knife","english","england","god"],
  "numToShow": 5
}
```

The JSON format uses '`{ }`' characters to enclose a hierarchical nested structure of key value pairs. In the example above there isn't any nesting, but rather only one top level set of '`{ }`'. Within this there are two keys: `words` and `numToShow`. Treating this as a JSON object means that the key `numToShow` has an associated value `5`. Similarly, `words` is an array of strings, with each element a potentially interesting word to consider in Shakespeare's texts. In general, JSON files are used for much more complex descriptions of data, but here we use this simple structure for illustration.

Now with some basic understanding of JSON, we can proceed with our example. The code in Listing 1.8 retrieves Shakespeare's texts from the web and then counts the occurrences of each of the `words`, ignoring case. We then show a count for each of the `numToShow` most common words.

Listing 1.8: Web interface, JSON and string parsing

```
1  using HTTP, JSON
2
3  data = HTTP.request("GET",
4  "https://ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.shakespeare.txt")
5  shakespeare = String(data.body)
6  shakespeareWords = split(shakespeare)
7
8  jsonWords = HTTP.request("GET",
9  "https://raw.githubusercontent.com/"*
10 "h-Klok/StatsWithJuliaBook/master/1_chapter/jsonCode.json")
11 parsedJsonDict = JSON.parse( String(jsonWords.body))
12
13 keywords = Array{String}(parsedJsonDict["words"])
14 numberToShow = parsedJsonDict["numToShow"]
15 wordCount = Dict([(x,count(w -> lowercase(w) == lowercase(x), shakespeareWords))
16                   for x in keywords])
17
18 sortedWordCount = sort(collect(wordCount),by=last,rev=true)
19 sortedWordCount[1:numberToShow]
```

```
5-element Array{Pair{String,Int64},1}:
"king"=>1698
"love"=>1279
"man"=>1033
"sir"=>721
"god"=>555
```

In lines 3-4 `HTTP.request` from the `HTTP` package is used to make a HTTP request. In line 5 the `body` of `data` is then parsed to a text string via the `String()` constructor function. In line 6 this string is then split into an array of individual words via the `split()` function.In lines 8-11 the JSON file is first retrieved. Then this string is parsed into a JSON object. The URL string for the JSON file doesn't fit on one line, so we use `*` to concatenate strings. In line 11 the `parse()` function from the `JSON` package is used to parse the body of the file and creates a dictionary. Line 13 shows the strength of using JSON as the value associated with the JSON key `words` is accessed. This value (i.e. array of words) is then cast to an `Array{String}` type. Similarly, the value associated with the key `numToShow` is accessed in line 14. In line 15 a Julia dictionary is created via `Dict()`. It is created from a comprehension of tuples, each with `x` (being a word) in the first element, and the count of these words in `shakespeareWords` as the second element. In using `count` we define the anonymous function as the first argument that compares an input test argument `w` to the given word `x`, only in `lowercase`. Finally line 18 sorts the dictionary by its values, and line 19 displays as output the first most popular `numberToShow` values.

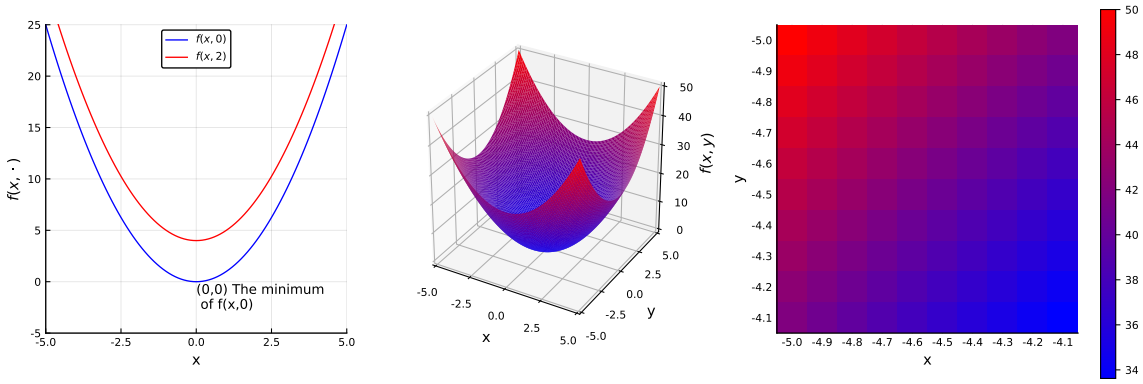

Figure 1.8: An introductory `Plots` example.

1.4 Plots, Images and Graphics

There are many different plotting packages available in Julia, including `PyPlot`, `Gadfly`, `Makie` as well as several others. Arguably, as a starting point, two of the most useful plotting packages are the `Plots` package and the `StatsPlots` package. `Plots` simplifies the process of creating plots, as it brings together many different plotting packages under a single API. With `Plots`, you can learn a single syntax, and then use the backend of your choice to create plots. Almost all of the examples throughout this book use the `Plots` package, and in almost all of the examples the code presented directly generates the figures. That is, if you want examples of how to create certain plots, one way of doing this is to browse through the figures of the book until you find one of interest, and then look at the associated code block and use this as inspiration for your plotting needs.

In `Plots`, input data is passed positionally, while aspects of the plot can be customized by specifying keywords for specific plot attributes, such as line color or width. In general, each attribute can take on a range of values, and in additon, many attributes have aliases which empower one to write short, conscise code. For example `color=:blue` can be shortened to `c=:blue`, and we make use of this alias mechanic throughout the books examples.

Since the code listings from this book can be used as direct examples, we don't present an extensive tutorial on the finer aspects of creating plots. Rather, if you are seeking detailed instructions or further references on finer points, we recommend that you visit:

`http://docs.juliaplots.org/`

As a minimal overview, the following is a brief list of some of the more commonly used `Plots` package functions for generating plots:

`plot()` - Can be used to plot data in various ways, including series data, single functions, multiple functions, as well as for presenting and merging other plots. This is the most common plotting function.

`scatter()` - Used for plotting scattered data points not connected by a line.

`bar()` - Used for plotting bar graphs.

`heatmap()` - Used to plot a matrix, or an image.

`surface()` - Used to plot surfaces (3D plots). This is the typical way in which one would plot a real valued function of two variables.

`contour()` - Used to create a contour plot. This is an alternative way to plot a real valued function of two variables.

`contourf()` - Similar to `contour()`, but with shading between contour lines.

`histogram()` - Used to plot histograms of data.

`stephist()` - A stepped histogram. This is a histogram with no filling.

In addition, each of these functions also has a companion function with a '`!`' suffix, for e.g. `plot!()`. These functions modify the previous plot, adding additional plotting aspects to them. This is shown in many examples throughout the book. Furthermore, the `Plots` package supplies additional important functions such as `savefig()` for saving a plot, `annotate!()` for adding annotations to plots, `default()` for setting plotting default arguments, and many more. Note that in the examples throughout this book `pyplot()` is called. This activates the `PyPlot` backend for plotting.

As a basic introductory example focused solely on plotting, we present Listing 1.9. In this listing, the main object is the real valued function of two variables, $f(x,y) = x^2 + y^2$. We use this *quadratic form* as a basic example, and also consider the cases of $y = 0$ and $y = 2$. The code generates Figure 1.8. Note the use of the `LaTeXStrings` package enabling LaTeX formatted formulas. See for example, `http://tug.ctan.org/info/undergradmath/undergradmath.pdf`.

Listing 1.9: Basic plotting

```
using Plots, LaTeXStrings, Measures; pyplot()

f(x,y) = x^2 + y^2
f0(x) = f(x,0)
f2(x) = f(x,2)

xVals, yVals = -5:0.1:5 , -5:0.1:5
plot(xVals, [f0.(xVals), f2.(xVals)],
        c=[:blue :red], xlims=(-5,5), legend=:top,
        ylims=(-5,25), ylabel=L"f(x,\cdot)", label=[L"f(x,0)" L"f(x,2)"])
p1 = annotate!(0, -0.2, text("(0,0) The minimum\n of f(x,0)", :left, :top, 10))

z = [ f(x,y) for y in yVals, x in xVals ]
p2 = surface(xVals, yVals, z, c=cgrad([:blue, :red]),legend=:none,
        ylabel="y", zlabel=L"f(x,y)")

M = z[1:10,1:10]
p3 = heatmap(M, c=cgrad([:blue, :red]), yflip=true, ylabel="y",
        xticks=([1:10;], xVals), yticks=([1:10;], yVals))

plot(p1, p2, p3, layout=(1,3), size=(1200,400), xlabel="x", margin=5mm)
```

Line 1 includes the following packages: `Plots` for plotting; `LaTeXStrings` for displaying labels using LaTeX formatting as in line 10; and `Measures` for specifying margins such as in line 21. In line 1, as part of a second statement following '`;`', `pyplot()` is called to indicate that the PyPlot plotting backend is activated. In line 3 we define the two variable real valued function `f()` which is the main object of this example. We then define two related single variable functions, `f0()` and `f2()` i.e. $f(x, 0)$ and $f(x, 2)$. In line 7 we define the ranges `xVals` and `yVals`. Line 8 is the first call to `plot()` where `xVals` is the first argument indicating the horizontal coordinates, and the array `[f0.(xVals), f2.(xVals)]` represents two data series to be plotted. Then in the same function call on lines 9 and 10, we specify colors, x-limits, y-limits, location of the legend, and the labels, where `L` denotes LaTeX. In line 11 `annotate!()` modifies the current plot with an annotation. The return value is the plot object stored in `p1`. Then in lines 13-15 we create a surface plot. The 'height' values are calculated via a two way comprehension and stored in the matrix `z` on line 13. Then `surface()` is used in lines 14-15 to crate the plot, which is then stored in the variable `p2`. Note the use of the `cgrad()` function to create a color gradient. In lines 17-19 a matrix of values is plotted via `heatmap()`. The argument `yflip=true` is important for orienting the matrix in the standard manner. Finally, in line 21 the three previous subplots are plotted together as a single figure via the `plot()` function.

Histogram of Hailstone Sequence Lengths

In this example we use `Plots` to create a *histogram* in the context of a well-known mathematical problem. Consider that we generate a sequence of numbers as follows: given a positive integer x, if it is even, then the next number in the sequence is $x/2$, otherwise it is $3x + 1$. That is, we start with some x_0 and then iterate $x_{n+1} = f(x_n)$ with

$$f(x) = \begin{cases} x/2 & \text{if } x \bmod 2 = 0, \\ 3x + 1 & \text{if } x \bmod 2 = 1. \end{cases}$$

The sequence of numbers arising from this function is called the *hailstone sequence*. As an example, if $x_0 = 3$, the resulting sequence is,

$$3, 10, 5, 16, 8, 4, 2, 1, \ldots,$$

where the cycle $4, 2, 1$ continues forever. We call the number of steps (possibly infinite) needed to hit 1 the length of the sequence, in this case 8. Note that different values of x_0 will result in different hailstone sequences of different lengths.

It is conjectured that, regardless of the x_0 chosen, the sequence will always converge to 1. That is, the length is always finite. However, this has not been proven to date and remains an open question, known as the *Collatz conjecture*. In addition, a counter-example has not yet been computationally found. That is, there is no known x_0 for which the sequence doesn't eventually go down to 1.

Now that the context of the problem is set, we create a histogram of lengths of hailstone sequences based on different values of x_0. Our approach is shown in Listing 1.10, where we first create a function which calculates the length of a hailstone sequence based on a chosen value of x_0. We then use a comprehension to evaluate this function for each value, $x_0 = 2, 3, \ldots, 10^7$, and finally plot a histogram of these lengths, shown in Figure 1.9.

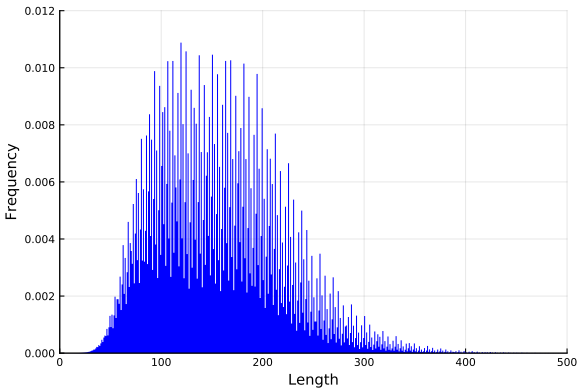

Figure 1.9: Histogram of hailstone sequence lengths.

Listing 1.10: Histogram of hailstone sequence lengths

```
1  using Plots; pyplot()
2
3  function hailLength(x::Int)
4      n = 0
5      while x != 1
6          if x % 2 == 0
7              x = Int(x/2)
8          else
9              x = 3x +1
10         end
11         n += 1
12     end
13     return n
14 end
15
16 lengths = [hailLength(x0) for x0 in 2:10^7]
17
18 histogram(lengths, bins=1000, normed=:true,
19     fill=(:blue, true), la=0, legend=:none,
20     xlims=(0, 500), ylims=(0, 0.012),
21     xlabel="Length", ylabel="Frequency")
```

In lines 3-14 the function `hailLength()` is created, which evaluates the length of a hailstone sequence, `n`, given the first number in the sequence, `x`. Note the use of `::Int`, which indicates the method implemented operates only on integer types. A `while` loop is used to sequentially and repeatedly evaluate all code contained within it, until the specified condition is `false`. In this case until we obtain a hailstone number of 1. Note the use of the *not-equals comparison operator*, `!=`. In line 6 the *modulo* operator, `%`, and *equality operator*, ==, are used in conjunction to check if the current number is even. If `true`, then we proceed to line 7, else we proceed to line 9. In line 11 our hailstone sequence length is increased by one each time we generate a new number in our sequence. In line 13 length of the sequence is returned. In line 16 a comprehension is used to evaluate our function for integer values of x_0 between 2 and 10^7. In lines 18-21 the `histogram()` function is used to plot a histogram using an arbitrary bin count of 1000.

Creating Animations

We now present an example of a live *animation* which sequentially draws the edges of a fully-connected mathematical *graph*. A graph is an object that consists of *vertices*, represented by dots, and *edges*, represented by lines connecting the vertices.

In this example we construct a series of equally spaced vertices around the *unit circle*, given an integer number of vertices, n. To add another aspect to this example, we obtain the points around the unit circle by considering the complex numbers,

$$z_n = e^{2\pi i \frac{k}{n}}, \qquad \text{for} \qquad k = 1, \ldots, n. \tag{1.4}$$

We then use the real and imaginary parts of z_n to obtain the horizontal and vertical coordinates for each vertex respectively, which distributes n points evenly on the unit circle. The example in Listing 1.11 sequentially draws all possible edges connecting each vertex to all remaining vertices, and animates the process. Each time an edge is created, a frame snapshot of the figure is saved, and by quickly cycling through the frames generated, we can generate an animated *GIF*. A single frame approximately half way through the GIF animation is shown in Figure 1.10.

Listing 1.11: Animated edges of a graph

```
using Plots; pyplot()

function graphCreator(n::Int)
    vertices = 1:n
    complexPts = [exp(2*pi*im*k/n) for k in vertices]
    coords = [(real(p),imag(p)) for p in complexPts]
    xPts = first.(coords)
    yPts = last.(coords)
    edges = []
    for v in vertices, u in (v+1):n
        push!(edges,(v,u))
    end

    anim = Animation()
    scatter(xPts, yPts, c=:blue, msw=0, ratio=1,
        xlims=(-1.5,1.5), ylims=(-1.5,1.5), legend=:none)

    for i in 1:length(edges)
        u, v = edges[i][1], edges[i][2]
        xpoints = [xPts[u], xPts[v]]
        ypoints = [yPts[u], yPts[v]]
        plot!(xpoints, ypoints, line=(:red))
        frame(anim)
    end

    gif(anim, "graph.gif", fps = 60)
end

graphCreator(16)
```

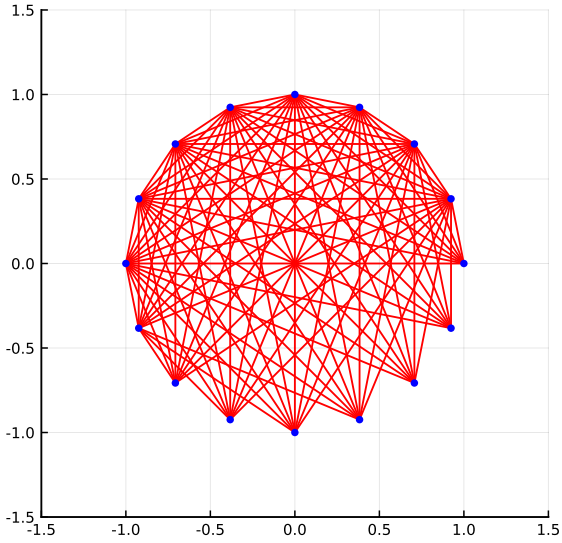

Figure 1.10: Sample frame from a graph animation.

The code defines the function `graphCreator()`, which constructs the animated GIF based on n number of vertices. In line 5 the complex points calculated via (1.4) are stored in the array `complexPoints`. In line 6 `real()` and `imag()` extract the real and imaginary parts of each complex number respectivly, and store them as paired tuples. In lines 7-8, the x and y coordinates are retrieved via `first()` and `last()` respectively. Note lines 5-8 could be shortened and implemented in various other ways, however the current implementation is useful for demonstrating several aspects of the language. Then lines 10-12 loop over `u` and `v`, and in line 11 the tuple `(u,v)` is added to `edges`. In line 14 an `Animation()` object is created. The vertices are plotted in lines 15-16 via `scatter()`. The loop in lines 18-24 plots a line for each of the edges via `plot!()`. Then `frame(anim)` adds the current figure as another frame to the animation object. The `gif()` function in line 26 saves the animation as the file `graph.gif` where `fps` defines how many frames per second are rendered.

Raster Images

We now present an example of working with *raster images*, namely images composed of individual pixels. In Listing 1.12 we load a sample image of stars in space and locate the brightest star. Note that the image contains some amount of noise, in particular as seen from the output, the single brightest pixel is located at $[192, 168]$ in *row major*. Therefore if we wanted to locate the brightest star by a single pixel's intensity, we would not identify the correct coordinates.

Since looking at single pixels can be deceiving, to find the highest intensity star, we use a simple method of parsing a kernel over the image. This technique smoothes the image and eliminates some of the noise. The results are in Figure 1.11 where the two subplots show the original image vs. the smoothed image, and the location of the brightest star for each.

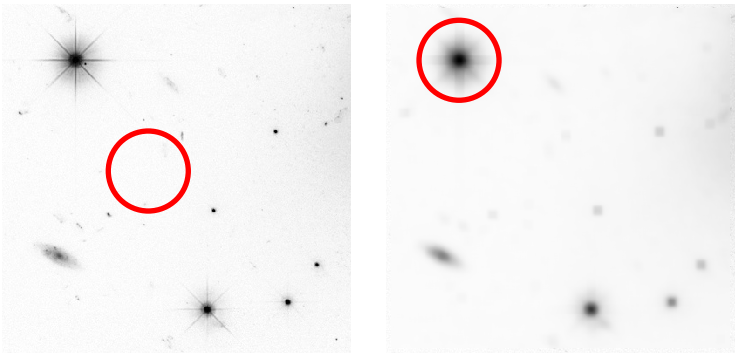

Figure 1.11: Left: Original image.
Right: Smoothed image after noise removal.

Listing 1.12: Working with images

```
using Plots, Images; pyplot()

img = load("../data/stars.png")
gImg = red.(img)*0.299 + green.(img)*0.587 + blue.(img)*0.114
rows, cols = size(img)

println("Highest intensity pixel: ", findmax(gImg))

function boxBlur(image,x,y,d)
    if x<=d || y<=d || x>=cols-d || y>=rows-d
        return image[x,y]
    else
        total = 0.0
        for xi = x-d:x+d
            for yi = y-d:y+d
                total += image[xi,yi]
            end
        end
        return total/((2d+1)^2)
    end
end

blurImg = [boxBlur(gImg,x,y,5) for x in 1:cols, y in 1:rows]

yOriginal, xOriginal = argmax(gImg).I
yBoxBlur, xBoxBlur   = argmax(blurImg).I

p1 = heatmap(gImg, c=:Greys, yflip=true)
p1 = scatter!((xOriginal, yOriginal), ms=60, ma=0, msw=4, msc=:red)
p2 = heatmap(blurImg, c=:Greys, yflip=true)
p2 = scatter!((xBoxBlur, yBoxBlur), ms=60, ma=0, msw=4, msc=:red)

plot(p1, p2, size=(800, 400), ratio=:equal, xlims=(0,cols), ylims=(0,rows),
        colorbar_entry=false, border=:none, legend=:none)
```

```
Highest intensity pixel: (0.9999999999999999, CartesianIndex(192, 168))
```

In line 3 the image is read into memory via the `load()` function and stored as `img`. Since the image is 400×400 pixels, it is stored as a 400×400 array of RGBA tuples of length 4. Each element of these tuples represents one of the color layers in the following order: red, green, blue, and luminosity. In line 4 we create a grayscale image from the original image data via a linear combination of its RBG layers. This choice of coefficients is a common "Grayscale algorithm". The gray image is stored as the matrix `gImg`. In line 5 the `size()` function is used to determine then number of rows and columns of `gImg`, which are then stored as `rows` and `cols` respectively. In line 7 `findmax()` is used to find the highest intensity element (pixel) in `gImg`. It returns a tuple of value and index, where in this case the index is of type `CartesianIndex` because `gImg` is a two dimensional array (matrix). In lines 9-21 the function `boxBlur` is created. This function takes an array of values as input, representing an image, and then passes a kernel over the image data, taking a linear average in the process. This is known as "box blur". In other words, at each pixel, the function returns a single pixel with a brightness weighting based on the average of the surrounding pixels (or array values) in a given neighborhood within a box of dimensions $2d + 1$. Note that the edges of the image are not smoothed, as a border of un-smoothed pixels of 'depth' d exists around the images edges. Visually, this kernel smoothing method has the effect of blurring the image. In line 23, the function `boxBlur()` is parsed over the image for a value of $d = 5$, i.e. a 10×10 kernel. The smoothed data is then stored as `blurImg`. In lines 25-26 we use the `argmax()` function which is similar to `findmax()`, but only returns the index. We use it to find the index of the pixel with the largest value, for both the non-smoothed and smoothed image data. Note the use of the trailing `.I` at the end of each `argmax()`, which extracts the `Tuple` of values of the co-ordinates from the `CartesianIndex` type. As the Cartesian index of matrices is row major, we reverse the row and column order for the plotting that follows. The remaining lines create Figure 1.11.

1.5 Random Numbers and Monte Carlo Simulation

Many of the code examples in this book make use of *pseudorandom number generation*, often coupled with the so-called *Monte Carlo simulation method* for obtaining numerical estimates. The phrase "Monte Carlo" associated with random number generation comes from the European province in Monaco famous for its many casinos. We now overview the core ideas and principles of random number generation and Monte Carlo simulation.

The main player in this discussion is the `rand()` function. When used without input arguments, `rand()` generates a "random" number in the interval $[0, 1]$. Several questions can be asked. How is it random? What does random within the interval $[0, 1]$ really mean? How can it be used as an aid for statistical and scientific computation? For this we discuss pseudorandom numbers in a bit more generality.

The "random" numbers we generate using Julia, as well as most "random" numbers used in any other scientific computing platform, are actually pseudorandom. That is, they aren't really random but rather appear random. For their generation, there is some deterministic (non-random and well defined) sequence, $\{x_n\}$, specified by

$$x_{n+1} = f(x_n, x_{n-1}, \ldots), \tag{1.5}$$

originating from some specified *seed*, x_0. The mathematical function, $f(\cdot)$ is often (but not always) quite a complicated function, designed to yield desirable properties for the sequence $\{x_n\}$ that make it appear random. Among other properties we wish for the following to hold:

(i) Elements x_i and x_j for $i \neq j$ should appear statistically independent. That is, knowing the value of x_i should not yield information about the value of x_j.

(ii) The distribution of $\{x_n\}$ should appear uniform. That is, there shouldn't be values (or ranges of values) where elements of $\{x_n\}$ occur more frequently than others.

(iii) The range covered by $\{x_n\}$ should be well defined.

(iv) The sequence should repeat itself as rarely as possible.

Typically, a mathematical function such as $f(\cdot)$ is designed to produce integers in the range $\{0, \ldots, 2^\ell - 1\}$ where ℓ is typically 16, 32, 64 or 128 (depending on the number of bits used to represent an integer). Hence $\{x_n\}$ is a sequence of pseudorandom integers. Then if we wish to have a pseudorandom number in the range $[0, 1]$ (represented via a floating point number), we normalize via,

$$U_n = \frac{x_n}{2^\ell - 1}.$$

When calling `rand()` in Julia (as well as in many other programming languages), what we are doing is effectively requesting the system to present us with U_n. Then, in the next call, U_{n+1}, and in the call after this U_{n+2} etc. As a user, we don't care about the actual value of n, we simply trust the computing system that the next pseudorandom number will differ and adhere to the properties (i) - (iv) mentioned above.

One may ask, where does the sequence start? For this we have a special name that we call x_0. It is known as the *seed* of the pseudorandom sequence. Typically, as a scientific computing system starts up, it sets x_0 to be the current time. This implies that on different system startups, $x_0, x_1, x_2, \ldots$ will be different sequences of pseudorandom numbers. However, we may also set the seed ourselves. There are several uses for this and it is often useful for reproducibility of results. Listing 1.13 illustrates setting the seed using Julia's `Random.seed!()` function.

Listing 1.13: Pseudorandom number generation

```
1  using Random
2
3  Random.seed!(1974)
4  println("Seed 1974: ",rand(),"\t", rand(), "\t", rand())
5  Random.seed!(1975)
6  println("Seed 1975: ",rand(),"\t", rand(), "\t", rand())
7  Random.seed!(1974)
8  println("Seed 1974: ",rand(),"\t", rand(), "\t", rand())
```

```
Seed 1974: 0.21334106865797864  0.12757925830167505    0.5047074487066832
Seed 1975: 0.7672833719737708   0.8664265778687816     0.5807364110163316
Seed 1974: 0.21334106865797864  0.12757925830167505    0.5047074487066832
```

As can be seen from the output, setting the seed to 1974 produces the same sequence. However, setting the seed to 1975 produces a completely different sequence.

One may ask why use random or pseudorandom numbers? Sometimes having arbitrary numbers alleviates programming tasks or helps randomize behavior. For example, when designing computer

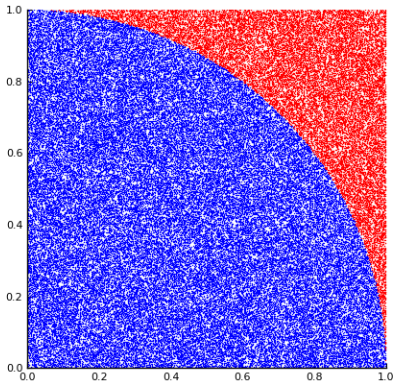

Figure 1.12: Estimating π via Monte Carlo.

video games, having enemies appear at random spots on the screen yields for a simple implementation. In the context of scientific computing and statistics, the answer lies in the Monte Carlo simulation method. Here the idea is that computations can be aided by repeated sampling and averaging out the result. Many of the code examples in our book do this and we illustrate one such simple example below.

Monte Carlo Simulation

As an example of Monte Carlo, say we wish to estimate the value of π. There are hundreds of known numerical methods to do this and here we explore one. Observe that the area of one quarter section of the unit circle is $\pi/4$. Now if we generate random points, (x, y), within a unit box, $[0, 1] \times [0, 1]$, and calculate the proportion of total points that fall within the quarter circle, we can approximate π via,

$$\hat{\pi} = 4 \, \frac{\text{Number of points with } x^2 + y^2 \leq 1}{\text{Total number of points}}.$$

This is performed in Listing 1.14 for 10^5 points. The listing also creates Figure 1.12.

Listing 1.14: Estimating π

```
using Random, LinearAlgebra, Plots; pyplot()
Random.seed!()

N = 10^5
data     = [[rand(),rand()] for _ in 1:N]
indata   = filter((x)-> (norm(x) <= 1), data)
outdata  = filter((x)-> (norm(x) > 1), data)
piApprox = 4*length(indata)/N
println("Pi Estimate: ", piApprox)

scatter(first.(indata),last.(indata), c=:blue, ms=1, msw=0)
scatter!(first.(outdata),last.(outdata), c=:red, ms=1, msw=0,
        xlims=(0,1), ylims=(0,1), legend=:none, ratio=:equal)
```

```
Pi Estimate: 3.14068
```

In Line 2 the seed of the random number generator is set with `Random.seed!()`. This is done to ensure that each time the code is run the estimate obtained is the same. In Line 4, the number of repetitions, `N`, is set. Most code examples in this book use `N` as the number of repetitions in a Monte Carlo simulation. Line 5 generates an array of arrays. That is, the pair, `[rand(),rand()]` is an array of random coordinates in $[0,1] \times [0,1]$. Line 6 filters those points to use for the numerator of $\hat{\pi}$. It uses the `filter()` function, where the first argument is an anonymous function, `(x) -> (norm(x) <= 1)`. Here, `norm()` defaults to the L_2 norm, i.e. $\sqrt{x^2+y^2}$. The resulting `indata` array only contains the points that fall within the unit circle (with each represented as an array of length 2). Line 7 creates the analogous `outdata` array. It is not used for the estimation, but is used in plotting. Line 8 calculates the approximation, with `length()` used for the numerator of $\hat{\pi}$ and `N` for the denominator. Lines 11-13 are used to create Figure 1.12.

Inside a Simple Pseudorandom Number Generator

Number theory and related fields play a central role in the mathematical study of pseudorandom number generation, the internals of which are determined by the specifics of $f(\cdot)$ of (1.5). However, typically this is not of direct interest statisticians. Nevertheless, for exploratory purposes we illustrate how one can make a simple pseudorandom number generator.

A simple to implement class of pseudo-random number generators is the class of *Linear Congruential Generators* (LCG). These types of LCGs are common in older systems. Here the function $f(\cdot)$ is nothing but an affine (linear) transformation modulo m,

$$x_{n+1} = (a\, x_n + c) \bmod m. \tag{1.6}$$

The integer parameters a, c and m are fixed and specify the details of the LCG. Some number theory research has determined "good" values of a and c for specific values of m. For example, for $m = 2^{32}$, setting $a = 69069$ and $c = 1$ yields sensible performance (other possibilities work well, but not all). In Listing 1.15 we generate values based on this LCG, see also Figure 1.13.

Listing 1.15: A linear congruential generator

```
1  using Plots, LaTeXStrings, Measures; pyplot()
2
3  a, c, m = 69069, 1, 2^32
4  next(z) = (a*z + c) % m
5
6  N = 10^6
7  data = Array{Float64,1}(undef, N)
8
9  x = 808
10 for i in 1:N
11     data[i] = x/m
12     global x = next(x)
13 end
14
15 p1 = scatter(1:1000, data[1:1000],
16     c=:blue, m=4, msw=0, xlabel=L"n", ylabel=L"x_n")
17 p2 = histogram(data, bins=50, normed=:true,
18     ylims=(0,1.1), xlabel="Support", ylabel="Density")
19 plot(p1, p2, size=(800, 400), legend=:none, margin = 5mm)
```

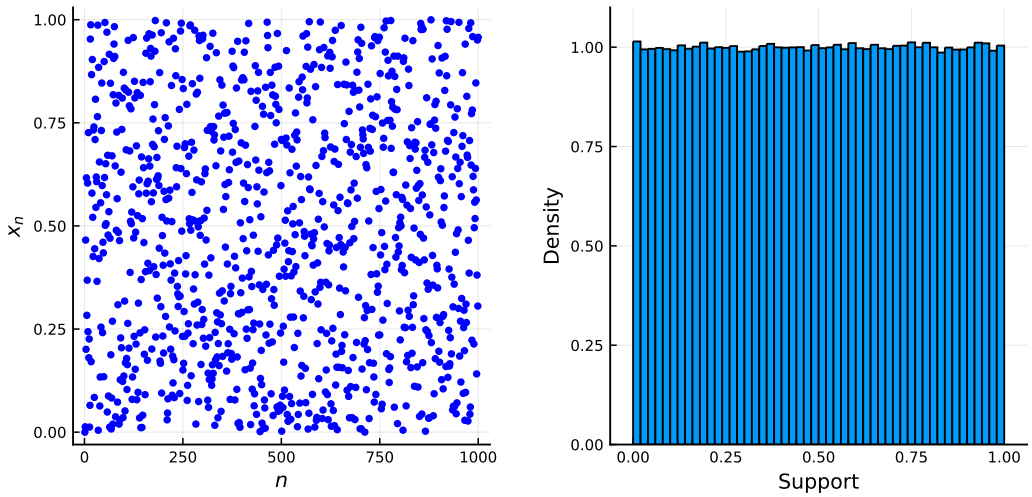

Figure 1.13: Left: The first 1,000 values generated by a linear congruential generator, plotted sequentially. Right: A histogram of 10^6 random values.

In line 4 (1.6) is implemented as the function `next()`. In line 7 an array of `Float64` of length `N` is preallocated. In line 9 the seed is arbitrarily set as the value `808`. In lines 10-13 a loop is used `N` times. In line 11 the current value of `x` is divided by m to obtain a number in the range $[0, 1]$. Note that in Julia division of two integers results in a floating point number. In line 12 (1.6) is applied recursively via `next()` to set a new value for `x`. In lines 15-16 a scatterplot of the first 1000 values of `data` is created, while lines 17-18 create a histogram of all values of `data` with 50 bins. As expected by the theory of LCG, a uniform distribution is obtained.

More About Julia's `rand()`

Having covered the basics, we now describe a few more aspects of Julia's random number generation. The key function at play is `rand()`. However, as you already know, a Julia function may be implemented by different methods. The `rand()` function is no different. To see this, key in `methods(rand)` and you'll see dozens of different methods of `rand()`. Furthermore, if you do this after loading the `Distributions` package into the namespace (by running `using Distributions`) that number will grow substantially. Hence in short, there are many ways to use the `rand()` function in Julia. Throughout the rest of this book we use it in various ways, including in conjunction with probability distributions. However we now focus on functionality from the `Base` package.

There are other functions related to `rand()`, such as `randn()` for generating normally distributed random variables. Also after invoking `using Random`, the following functions are available: `Random.seed!()`, `randsubseq()`, `randstring()`, `randcycle()`, `bitrand()`, as well as `randperm()` and `shuffle()` for permutations. There is also the `MersenneTwister()` constructor among others. These are discussed in the Julia documentation. You may also use the built-in help to enquire about them. We now focus on the `MersenneTwister()` constructor and explain how it can be used in conjunction with `rand()` and variants.

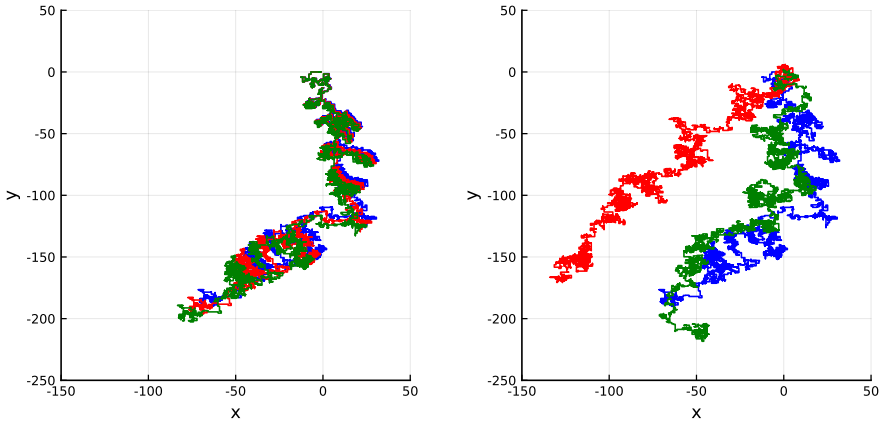

Figure 1.14: Random walks with slightly different parameters.
Left: Trajectories with same seed. Right: Different seed per trajectory.

The term *Mersenne Twister* refers to a type of pseudorandom number generator. It is an algorithm that is considerably more complicated than the LCG described above. Generally, its statistical properties are much better than those of LCG. Due to this it has made its way into most scientific programming environments in the past two decades. Julia has adopted it as the standard as well.

Our interest in mentioning the Mersenne Twister is due to the fact that in Julia we can create an object representing a random number generator implemented via this algorithm. To create such an object we write for example `rng = MersenneTwister(seed)`, where `seed` is some initial seed value. Then the object `rng` acts as a random number generator, and may serve as an additional input to `rand()` and related functions. For example, calling `rand(rng)` uses the specific random number generator object passed to it. In addition to `MersenneTwister()`, there are also other ways to create similar objects, such as for example `RandomDevice()`. However we leave it to the reader to investigate these via the online help.

By creating random number generator objects, you may have more than one random sequence in your application, essentially operating simultaneously. In Chapter 10, we investigate scenarios where this is advantageous from a Monte Carlo simulation perspective. For now we show how a random number generator may be passed into a function as an argument, allowing the function to generate random values using that specific generator.

Listing 1.16 creates random paths in the plane. Each path starts at $(x, y) = (0, 0)$ and moves up, right, down or left at each step. The movements up (`x+=1`) and right (`y+=1`) are with steps of size 1. However the movements down and left are with steps that are uniformly distributed in the range $[0, 2 + \alpha]$. Hence if $\alpha > 0$, on average the path drifts in the down-left direction. The virtue of this initial example is that by using *common random numbers* and simulating paths for varying α, we get very different behavior than if we use a different set of random numbers for each path. See Figure 1.14. We discuss more advanced applications of using multiple random number generators in Chapter 10, however we implicitly use this Monte Carlo technique throughout the book, often by setting the seed to a specific value in the code examples.

Listing 1.16: Random walks and seeds

```
1  using Plots, Random, Measures; pyplot()
2
3  function path(rng, alpha, n=5000)
4      x, y = 0.0, 0.0
5      xDat, yDat = [], []
6      for _ in 1:n
7          flip = rand(rng,1:4)
8          if flip == 1
9              x += 1
10         elseif flip == 2
11             y += 1
12         elseif flip == 3
13             x -= (2+alpha)*rand(rng)
14         elseif flip == 4
15             y -= (2+alpha)*rand(rng)
16         end
17         push!(xDat,x)
18         push!(yDat,y)
19     end
20     return xDat, yDat
21 end
22
23 alphaRange = [0.2, 0.21, 0.22]
24
25 default(xlabel = "x", ylabel = "y", xlims=(-150,50), ylims=(-250,50))
26 p1 = plot(path(MersenneTwister(27), alphaRange[1]), c=:blue)
27 p1 = plot!(path(MersenneTwister(27), alphaRange[2]), c=:red)
28 p1 = plot!(path(MersenneTwister(27), alphaRange[3]), c=:green)
29
30 rng = MersenneTwister(27)
31 p2 = plot(path(rng, alphaRange[1]), c=:blue)
32 p2 = plot!(path(rng, alphaRange[2]), c=:red)
33 p2 = plot!(path(rng, alphaRange[3]), c=:green)
34
35 plot(p1, p2, size=(800, 400), legend=:none, margin=5mm)
```

Lines 3-21 define the function `path()`. As a first argument it takes a random number generator, `rng`. That is, the function is designed to receive an object such as `MersenneTwister` as an argument. The second argument is `alpha` and the third argument is the number of steps in the path with a default value of `5000`. In lines 6-19 we loop `n` times, each time updating the current coordinate (`x` and `y`) and then pushing the values into the arrays, `xDat` and `yDat`. Line 7 generates a random value in the range `1:4`. Observe the use of `rng` as a first argument to `rand()`. In lines 13 and 15 we multiply `rand(rng)` by `(2+alpha)`. This creates uniform random variables in the range $[0, 2+\alpha]$. Line 20 returns a tuple of two arrays `xDat,yDat`. After setting `alphaRange` in line 23 and setting default plotting arguments in line 25, we create and plot paths with common random numbers in lines 26-28. This is because in each call to `path()` we use the same seed to a newly created `MersenneTwister()` object. Here `27` is just an arbitrary starting seed. In contrast, lines 31-33 have repeated calls to `path()` using a single stream, `rng`, created in line 30. Hence here, we don't have common random numbers because each subsequent call to `path()` starts at a fresh point in the stream of `rng`.

1.6 Integration with Other Languages

We now briefly overview how Julia can interface with the R-language, Python, and C. Note that there are several other packages that enable integration with other languages as well.

Using and Calling R Packages

R code, functions, and libraries can be called in Julia via the `RCall` package which provides several different ways of interfacing with R from Julia. When working with the REPL, one may use `$` to switch between a Julia REPL and an R REPL. However in this case variables are not carried over between the two environments. The second way is via the `@rput` and `@rget` macros, which can be used to transfer variables from Julia to the R environment. Finally, the `R"""` (or `@R_str`) macro can also be used to parse R code contained within the string. This macro returns an `RObject` as output, which is a Julia wrapper type around an R object.

We provide a brief example in Listing 1.17. It is related to Chapter 7 and focuses on the statistical method of ANOVA (Analysis of Variance) covered in Section 7.3. The purpose here is to demonstrate R-interoperability, and not so much on ANOVA. This example calculates the ANOVA F-statistic and p-value, complementing Listing 7.10. It makes use of the R `aov()` function and yields the same numerical results.

Listing 1.17: Using R from Julia

```
using CSV, DataFrames, RCall

data1 = CSV.read("../data/machine1.csv", header=false)[:,1]
data2 = CSV.read("../data/machine2.csv", header=false)[:,1]
data3 = CSV.read("../data/machine3.csv", header=false)[:,1]

function R_ANOVA(allData)
    data = vcat([ [x fill(i, length(x))] for (i, x) in
                        enumerate(allData) ]...)
    df = DataFrame(data, [:Diameter, :MachNo])
    @rput df

    R"""
    df$MachNo <- as.factor(df$MachNo)
    anova <- summary(aov( Diameter ~ MachNo, data=df))
    fVal <- anova[[1]]["F value"][[1]][1]
    pVal <- anova[[1]]["Pr(>F)"][[1]][1]
    """
    println("R ANOVA f-value: ", @rget fVal)
    println("R ANOVA p-value: ", @rget pVal)
end

R_ANOVA([data1, data2, data3])
```

```
R ANOVA f-value: 10.516968568709089
R ANOVA p-value: 0.00014236168817139574
```

In line 1 we specify usage of the required packages, including `RCall`. In lines 3-5 the data is loaded. In lines 7-21 we create the Julia function `R_ANOVA`, which takes a Julia array of arrays as input, `allData`. It outputs the summary results of an ANOVA test carried out in R via the `aov()` function. In lines 8-9 the array of arrays `allData` is re-arranged into a 2-dimensional array, where the first column contains the observations from each of the arrays, and the second column contains the array index from which each observation has come. The data is re-arranged like this due to the format that the R `aov()` function requires. This re-arrangement is performed via the `enumerate()` function, along with the `vcat()` function and splat '`...`' operator. In line 10, the 2-dimensional array `data` is converted to a `DataFrame`. Data frames are covered in Section 4.1. In line 11 the `@rput` macro is used to transfer the data frame `df` to the R workspace. In lines 13-18 a multi-line R code block is executed inside the `R"""` macro. In line 14, the `MachNo` column of the R data frame `df` is defined as a factor, i.e. a categorical column via the R code `as.factor()` and `<-`. In line 15 an ANOVA test of the `Diameter` column of the R data frame `df` is conducted via `aov()` and passed to the `summary()` function, with the result stored as `anova`. In lines 16-17, the F-value and p-value is extracted from `anova`. Lines 19 and 20 are back to Julia where the output is printed. Note the use of `@rget` which is used to copy the variables from R back to Julia using the same name.

In addition to various R functions, users of R will most likely also be familiar with *R Datasets*. This is a collection of datasets commonly used in teaching and exploring statistics. You can read more about R Datasets at,

`https://vincentarelbundock.github.io/Rdatasets/datasets.html`.

Access to this collection of datasets from Julia is possible via the `RDatasets` package. Once installed in Julia, datasets can be loaded by using the `datasets()` function and specifying an 'R datasets package name' followed by a 'dataset name'. For example, `datasets("datasets", "mtcars")`, will load `mtcars`. Several code listings in this book use R datasets.

Using and Calling Python Packages

It is possible to import Python modules and call Python functions directly in Julia via the `PyCall` package. It automatically converts types, and allows data structures to be shared between Python and Julia. By default, `add PyCall` uses the `Conda` package to install a minimal Python distribution that is private to Julia. Further python packages can then be installed from within Julia via the Julia `Conda` package.

Alternatively, one can use a pre-existing Python installation on the system. In order to do this, one must first set the Python environment variable to the path of the executable, and then re-build the `PyCall` package. For example, on a system with Anaconda installed, one would issue commands similar to the below from within the Julia REPL:

```
] add PyCall

ENV["PYTHON"] = "C:\\Program Files\\Anaconda3\\python.exe"

] build PyCall
```

We now provide a brief example which makes use of the TextBlob Python library, which provides a simple API for conducting *Natural Language Processing* (NLP) tasks, including part-of-speech tagging, noun phrase extraction, sentiment analysis, classification, translation, and more. For our example we use TextBlob to analyze the sentiment of several sentences. The sentiment analyzer of TextBlob outputs a tuple of values, with the first value being the polarity of the sentence (a rating of positive to negative), and the second value a rating of subjectivity (factual to subjective).

In order for Listing 1.18 to work, the TextBlob Python library must first be installed. The lines below do this when executed in a shell or command prompt. Note that one can swap from the Julia REPL to a shell via '`;`'.

```
pip3 install -U textblob

python -m textblob.download_corpora
```

Once Python and TextBlob are configured, Listing 1.18 can be executed. This example only briefly touches on the `PyCall` package with more information available in the package documentation.

Listing 1.18: NLP via Python's TextBlob

```
1  using PyCall
2  TB = pyimport("textblob")
3
4  str =
5  """Some people think that Star Wars The Last Jedi is an excellent movie,
6  with perfect, flawless storytelling and impeccable acting. Others
7  think that it was an average movie, with a simple storyline and basic
8  acting. However, the reality is almost everyone felt anger and
9  disappointment with its forced acting and bad storytelling."""
10
11 blob = TB.TextBlob(str)
12 [ i.sentiment for i in blob.sentences ]
```

```
(0.625, 0.636)
(-0.0375, 0.221)
(-0.46, 0.293)
```

In line 2 the `pyimport()` function is used to wrap the Python library `textblob`, which is then given the Julia alias `TB`. In lines 4-9 the string `str` is created. For this example, the string is written as a first hand account, and contains many words that give the text a negative tone. Note the use of multi-line strings using `"""`. In line 11 the `TextBlob()` function from `TB` is used to parse each sentence in `str`. The output is stored as `blob`. This is where the call to Python is made. In line 12 a comprehension is used to print the sentiment field for each sentence in `blob`. Note that `sentiment` is a Python based field name accessible via Julia. As detailed in the TextBlob documentation, the sentiment of the blob is as an ordered pair of polarity and subjectivity, with polarity measured over $[-1.0, 1.0]$ (very negative to very positive), and subjectivity over $[0.0, 1.0]$ (very objective to very subjective). The results indicate that the first sentence is the most positive but is also the most subjective, while the last sentence, is the most negative but also more objective.

Other Integrations

Julia also allows C and Fortran calls to be made directly via the `ccall()` function, which is in Julia `Base`. These calls are made without adding any extra overhead than a standard library call from C code. Note that the code to be called must be available as a shared library. For example, in Windows systems, `msvcrt` can be called instead of `libc` (`msvcrt` is a module containing C library functions, and is part of the Microsoft C Runtime Library).

When using the `ccall()` function, shared libraries are referenced in the format `(:function, "library")`. The following is an example where the C function `cos()` is called,

```
ccall( (:cos, "msvcrt"), Float64, (Float64,), pi ).
```

For this example, the `cos()` function is called from the `msvcrt` library. Here, `ccall()` takes four arguments, the first is the function and library as a tuple, the second is the return type, the third is a tuple of input types (here there is just one), and the last is the input argument, π in this case. Running this in Julia on a Windows machine returns -1.

There are also several other packages that support various other languages as well, such as `Cxx.jl` and `CxxWrap.jl` for C++, `MATLAB.jl` for Matlab, and `JavaCall.jl` for Java. Note that many of these packages are available from `https://github.com/JuliaInterop`.

Chapter 2

Basic Probability - DRAFT

In this chapter we introduce elementary probability concepts. We describe key notions of a probability space along with independence and conditional probability. It is important to note that most of the probabilistic analysis carried out in statistics is based on distributions of random variables. These are introduced in the next chapter. In this chapter we focus solely on probability, events, and the simple mathematical set-up of a random experiment embodied in a probability space.

The notion of *probability* is the chance of something happening, quantified as a number between 0 and 1 with higher values indicating a higher likelihood of occurrence. However, how do we formally describe probabilities? The standard way to do this is to consider a *probability space*; which mathematically consists of three elements: (1) A *sample space* - the set of all possible outcomes of a certain *experiment*. (2) A collection of *events* - each event is a subset of the sample space. (3) A *probability measure* also denoted here as *probability function* - which indicates the chance of each possible event occurring. Note: do not confuse this with a probability mass function, which we define in the next chapter.

As a simple example, consider the case of flipping a coin twice. Recall that the sample space is the set of all possible outcomes. We can represent the sample space mathematically as follows,

$$\Omega = \{hh, ht, th, tt\}.$$

Now that the sample space, Ω, is defined, we can consider individual events. For example, let A be the event of getting at least one heads. Hence,

$$A = \{hh, ht, th\}.$$

Or alternately, let B be the event of getting one heads and one tails in any order,

$$B = \{ht, th\}.$$

There can also be events that consist of a single possible outcome, for example $C = \{th\}$ is the event of getting tails first, followed by heads. Mathematically, the important point is that events are subsets of Ω and often contain more than one outcome. Possible events also include the empty set, $\emptyset$ (nothing happening) and Ω itself (something happening). In the setup of probability, we assume there is a *random experiment* where something is bound to happen.

The final component of a probability space is the probability function, also sometimes called *probability measure*. This function, $\mathbb{P}(\cdot)$, takes an event as an input argument and returns real numbers in the range $[0, 1]$. It always satisfies $\mathbb{P}(\emptyset) = 0$ and $\mathbb{P}(\Omega) = 1$. It also satisfies the fact that the probability of the union of two disjoint events is the sum of their probabilities, and furthermore the probability of the complement of an event is one minus the original probability.

This chapter is structured as follows: In Section 2.1 we explore the basic setup of random experiments with a few examples. In Section 2.2 we explore working with sets in Julia as well as probability examples dealing with unions of events. In Section 2.3 we introduce and explore the concept of independence. In Section 2.4 we move on to conditional probability. Finally, in Section 2.5 we explore Bayes' rule for conditional probability.

2.1 Random Experiments

We now explore a few examples where we set-up a *probability space*. In most examples we present a Monte Carlo simulation of the random experiment, and then compare results to theoretical ones where possible.

Rolling Two Dice

Consider the *random experiment* where two independent, fair, six sided dice are rolled, and we wish to find the probability that the sum of the outcomes of the dice is even. Here the sample space can be represented as $\Omega = \{1, \ldots, 6\}^2$, i.e. the *Cartesian product* of the set of single roll outcomes with itself. That is, elements of the sample space are *tuples* of the form (i, j) with $i, j \in \{1, \ldots, 6\}$. Say we are interested in the probability of the event,

$$A = \{(i, j) \mid i + j \text{ is even}\}.$$

In this random experiment, since the dice have no inherent bias, it is sensible to assume a *symmetric probability function*. That is, for any $B \subset \Omega$,

$$\mathbb{P}(B) = \frac{|B|}{|\Omega|},$$

	1	**2**	**3**	**4**	**5**	**6**
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Table 2.1: All possible outcomes for the sum of two dice. Even sums are shaded.

where $|\cdot|$ counts the number of elements in the set. It is called symmetric because every outcome in Ω has the same probability. Hence for our event, A, we can see from Table 2.1 that,

$$\mathbb{P}(A) = \frac{18}{36} = 0.5.$$

We now obtain this in Julia via both direct calculation and Monte Carlo simulation. A direct calculation counts the number of even faces. A Mote Carlo simulation repeats the experiment many times and estimates $\mathbb{P}(A)$ based on the number of times that event A occurred.

Listing 2.1: Even sum of two dice

```
1  N, faces = 10^6, 1:6
2
3  numSol = sum([iseven(i+j) for i in faces, j in faces]) / length(faces)^2
4  mcEst  = sum([iseven(rand(faces) + rand(faces)) for i in 1:N]) / N
5
6  println("Numerical solution = $numSol \nMonte Carlo estimate = $mcEst")
```

```
Numerical solution = 0.5
Monte Carlo estimate = 0.499644
```

In line 1 we set the number of simulation runs, `N`, and the range of faces on the dice, `1:6`. In line 3, we use a comprehension to cycle through the sum of all possible combinations of the addition of the outcomes of the two dice. The outcome of the two dice are represented by `i` and `j` respectively, both of which take on the values of `faces`. We start with `i`=1, `j`=1 and add them, and we use the `iseven()` function to return `true` if even, and `false` if not. We then repeat the process for `i`=1, `j`=2 and so on, all the way to `i`=6, `j`=6. Finally, we count the number of `true` values by summing all the elements of the comprehension via `sum()`. The result, normalized by the total number of possible outputs is stored in `numSol`. Line 4 also uses a comprehension, but in this case we uniformly and randomly select the values which the dice take, akin to rolling them. Again `iseven()` is used to return `true` if even and `false` if not, and we repeat this process `N` times. Using similar logic to line 3, we store the proportion of outcomes which were true in `mcEst`. Line 6 prints the results using the `println()` function. Notice the use of `\n` for creating a newline.

Partially Matching Passwords

We now consider an alphanumeric example. Assume that a password to a secured system is exactly 8 characters in length. Each character is one of 62 possible characters: the letters 'a'–'z', the letters 'A'–'Z' or the digits '0'–'9'.

In this example let Ω be the set of all possible passwords, i.e. $|\Omega| = 62^8$. Now, again assuming a symmetric probability function, the probability of an attacker guessing the correct (arbitrary) password is $62^{-8} \approx 4.6 \times 10^{-15}$. Hence at a first glance, the system seems very secure.

Elaborating on this example, let us also assume that as part of the system's security infrastructure, when a login is attempted with a password that matches 1 or more of the characters, an event is logged in the system's security portal (taking up hard drive space). For example, say the original password is `3xyZu4vN`, and a login is attempted using the password **3**`5xyZ`**4vN**. In this case 4 of the characters match (displayed in bold) and therefore an event is logged.

While the chance of guessing a password and logging in seems astronomically low, in this simple (fictional and overly simplistic) system, there exists a secondary security flaw. That is, hackers may attempt to overload the event logging system via random attacks. If hackers continuously try to log into the system with random passwords, every password that matches one or more characters will log an event, thus taking up more hard-drive space.

We now ask what is the probability of logging an event with a random password? Denote the event of logging a password A. In this case, it turns out to be much more convenient to consider the *complement*, $A^c := \Omega \setminus A$, which is the event of having 0 character matches. We have that $|A^c| = 61^8$ because given any (arbitrary) correct password, there are $61 = 62 - 1$ character options for each character, in order ensure A^c holds. Hence,

$$\mathbb{P}(A^c) = \frac{61^8}{62^8} \approx 0.87802.$$

We then have that the probability of logging an event is $\mathbb{P}(A) = 1 - \mathbb{P}(A^c) \approx 0.12198$. So if, for example, 10^7 login attempts are made, we can expect that about 1.2 million login attempts would be written to the security log. We now simulate such a scenario in Listing 2.2.

Listing 2.2: Password matching

```
using Random
Random.seed!()

passLength, numMatchesForLog = 8, 1
possibleChars = ['a':'z' ; 'A':'Z' ; '0':'9']

correctPassword = "3xyZu4vN"

numMatch(loginPassword) =
    sum([loginPassword[i] == correctPassword[i] for i in 1:passLength])

N = 10^7

passwords = [String(rand(possibleChars,passLength)) for _ in 1:N]
numLogs   = sum([numMatch(p) >= numMatchesForLog for p in passwords])
println("Number of login attempts logged: ", numLogs)
println("Proportion of login attempts logged: ", numLogs/N)
```

```
Number of login attempts logged: 1221801
Proportion of login attempts logged: 0.1221801
```

In line 2 the seed of the random number generator is set so that the same passwords are generated each time the code is run. This is done for reproducibility. In line 4 the password length is defined along with the minimum number of character matches before a security log entry is created. In line 5 an array is created, which contains all valid characters which can be used in the password. Note the use of '`;`', which performs *array concatenation* of the three ranges of characters. In line 7 we set an arbitrary correct login password. Note that the type of `correctPassword` is a `String` containing only characters from `possibleChars`. In lines 9 and 10 the function `numMatch()` is created, which takes the password of a login attempt and checks each index against that of the actual password. If the index character is correct, it evaluates `true`, else `false`. The function then returns how many characters were correct by using `sum()`. Line 14 uses the function `rand()` and the constructor `String()` along with a comprehension to randomly generate `N` passwords. Note that `String()` is used to convert from an array of single characters to a string. Line 15 checks how many times `numMatchesForLog` or more characters were guessed correctly, for each password in our array of randomly generated passwords. It then stores how many times this occurs as the variable `numLogs`.

The Birthday Problem

For our next example, consider a room full of people. We then ask what is the probability of finding a pair of people that share the same birthday. Obviously, ignoring leap years, if there are 366 people present, then it happens with certainty via the *pigeonhole principle*. However, what if there are fewer people? Interestingly, with about 50 people, a birthday match is almost certain, and with 23 people in a room, there is about a 50% chance of two people sharing a birthday. At first glance this non-intuitive result is surprising, and hence this famous probability example earned the name *the birthday paradox*. However, we just refer to it as the *birthday problem*.

To carry out the analysis, we assume birthdays are uniformly distributed in the set $\{1, \ldots, 365\}$. For n people in a room, we wish to evaluate the probability that at least two people share the same birthday. Set the sample space, Ω, to be composed of ordered tuples $(x_1, \ldots, x_n)$ with $x_i \in \{1, \ldots, 365\}$. Hence, $\left|\Omega\right| = 365^n$. Now set the event A to be the set of all tuples $(x_1, \ldots, x_j)$ where $x_i = x_j$ for some distinct i and j.

As in the previous example, we consider A^c instead. It consists of tuples where $x_i \neq x_j$ for all distinct i and j (the event of no birthday pair in the group). In this case,

$$\left|A^c\right| = 365 \cdot 364 \cdot \ldots \cdot (365 - n + 1) = \frac{365!}{(365 - n)!}.$$

Hence we have,

$$\mathbb{P}(A) = 1 - \mathbb{P}(A^c) = 1 - \frac{\left|A^c\right|}{\left|\Omega\right|} = 1 - \frac{365 \cdot 364 \cdot \ldots \cdot (365 - n + 1)}{365^n}. \tag{2.1}$$

From this we can compute that for $n = 23$, $\mathbb{P}(A) \approx 0.5073$, and for $n = 50$, $\mathbb{P}(A) \approx 0.9704$.

The code in Listing 2.3 calculates both the analytic probabilities, as well as estimates them via Monte Carlo (MC) simulation. The results are presented in Figure 2.1. For the numerical solutions, it employs two alternative implementations, `matchExists1()` and `matchExists2()`. The maximum error between the two numerical implementations is presented.

Listing 2.3: The birthday problem

```
using StatsBase, Combinatorics, Plots ; pyplot()

matchExists1(n) = 1 - prod([k/365 for k in 365:-1:365-n+1])
matchExists2(n) = 1- factorial(365,365-big(n))/365^big(n)

function bdEvent(n)
    birthdays = rand(1:365,n)
    dayCounts = counts(birthdays, 1:365)
    return maximum(dayCounts) > 1
end

probEst(n) = sum([bdEvent(n) for _ in 1:N])/N

xGrid = 1:50
analyticSolution1 = [matchExists1(n) for n in xGrid]
analyticSolution2 = [matchExists2(n) for n in xGrid]
println("Maximum error: $(maximum(abs.(analyticSolution1 - analyticSolution2)))")

N = 10^3
mcEstimates = [probEst(n) for n in xGrid]

plot(xGrid, analyticSolution1, c=:blue, label="Analytic solution")
scatter!(xGrid, mcEstimates, c=:red, ms=6, msw=0, shape=:xcross,
        label="MC estimate", xlims=(0,50), ylims=(0, 1),
        xlabel="Number of people in room",
        ylabel="Probability of birthday match",
        legend=:topleft)
```

```
Maximum error: 2.461172365062727820892938 5e-16
```

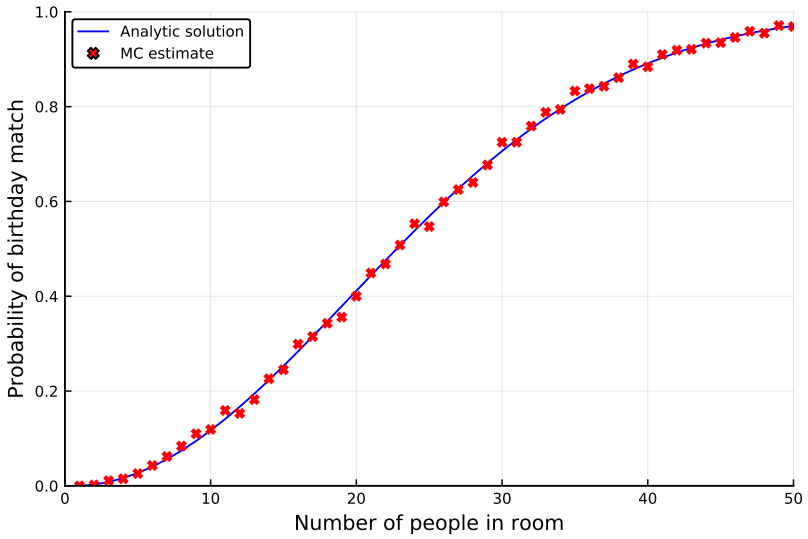

Figure 2.1: Probability that in a room of n people, at least two people share a birthday.

In lines 3 and 4, two alternative functions for calculating the probability in (2.1) are defined, `matchExists1()` and `matchExists2()` respectively. The first uses the `prod()` function to apply a product over a comprehension. This is in fact a numerically stable way of evaluating the probability. The second implementation evaluates (2.1) in a much more explicit manner. It uses the `factorial()` function from the `Combinatorics` package. Note that the basic `factorial()` function is included in Julia `Base`, however the method with two arguments comes from the `Combinatorics` package. Also, the use of `big()` ensures the input argument is a `BigInt` type. This is needed to avoid overflow for non-small values of `n`. Lines 6-10 define the function `bdEvent()`, which simulates a room full of n people, and if at least two people share a birthday, returns `true`, otherwise returns `false`. We now explain how it works. Line 7 creates the array `birthdays` of length n, and uniformly and randomly assigns an integer in the range $[1, 365]$ to each index. The values of this array can be thought of as the birth dates of individual people. Line 8 uses the function `counts()` from the `StatsBase` package to count how many times each birth date occurs in `birthdays`, and assigns these counts to the new array `dayCounts`. The logic can be thought of as follows: if two indices have the same value, then this represents two people having the same birthday. Line 9 checks the array `dayCounts`, and if the maximum value of the array is greater than one (i.e. if at least two people share the same birth date) then returns `true`, else `false`. Line 12 defines the function `probEst()`, which, when given n number of people, uses a comprehension to simulate `N` rooms, each containing `n` people. For each element of the comprehension, i.e. room, the `bdEvent()` function is used to check if at least one birthday pair exists. Then, for each room, the total number of at least one birthday pair is summed up and divided by the total number of rooms `N`. For large `N`, the function `probEst()` will be a good estimate for the analytic solution of finding at least one birthday pair in a room of `n` people. Lines 14-17 evaluate the analytic solutions over the grid, `xGrid`, and prints the maximal absolute error between the solutions. The output shows that the numerical error is negligible. Line 20 evaluates the Monte Carlo estimates. Lines 22-27 plot the analytic and numerical estimates of these probabilities on the same graph.

Sampling With and Without Replacement

Consider a small pond with a small population of 7 fish, 3 of which are gold and 4 of which are silver. Now say we fish from the pond until we catch 3 fish, either gold or silver. Let G_n denote the event of catching n gold fish. It is clear that unless $n = 0, 1, 2$ or 3, $\mathbb{P}(G_n) = 0$. However, what is $\mathbb{P}(G_n)$ for $n = 0, 1, 2, 3$? Before continuing, let us make a distinction between two sampling policies:

Catch and keep - We sample from the population *without replacement*. That is, whenever we catch a fish, we remove it from the population.

Catch and release - We sample from the population *with replacement*. That is, whenever we catch a fish, we return it to the population (pond) before continuing to fish.

The computation of the probabilities $\mathbb{P}(G_n)$ for these two cases of catch and keep, and catch and release, may be obtained via the *Hypergeometric distribution* and *Binomial distribution* respectively. These are both covered in more detail in Section 3.5. We now estimate these probabilities using Monte Carlo simulation. Listing 2.4 below simulates each policy N times, counts how many times zero, one, two and three gold fish are sampled in total, and finally presents these as proportions of the total number of simulations. Note that the total probability in both cases sum to one. The probabilities are plotted in Figure 2.2.

Listing 2.4: Fishing with and without replacement

```
using StatsBase, Plots ; pyplot()

function proportionFished(gF,sF,n,N,withReplacement = false)
    function fishing()
        fishInPond = [ones(Int64,gF); zeros(Int64,sF)]
        fishCaught = Int64[]

        for fish in 1:n
            fished = rand(fishInPond)
            push!(fishCaught,fished)
            if withReplacement == false
                deleteat!(fishInPond, findfirst(x->x==fished, fishInPond))
            end
        end
        sum(fishCaught)
    end

    simulations = [fishing() for _ in 1:N]
    proportions = counts(simulations,0:n)/N

    if withReplacement
        plot!(0:n, proportions,
             line=:stem, marker=:circle, c=:blue, ms=6, msw=0,
             label="With replacement",
             xlabel="n",
             ylims=(0, 0.6), ylabel="Probability")
    else
        plot!(0:n, proportions,
            line=:stem, marker=:xcross, c=:red, ms=6, msw=0,
            label="Without replacement")
    end
end

N = 10^6
goldFish, silverFish, n = 3, 4, 3
plot()
proportionFished(goldFish, silverFish, n, N)
proportionFished(goldFish, silverFish, n, N, true)
```

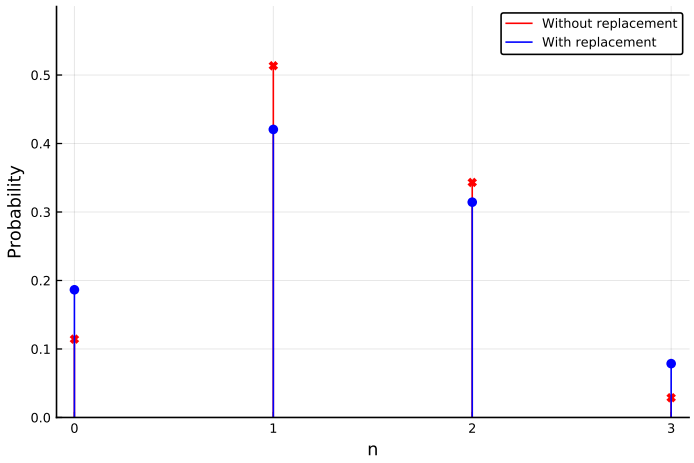

Figure 2.2: Estimated probabilities of catching n of gold fish, with and without replacement.

Lines 3-32 define the function `proportionFished()`, which takes five arguments: the number of gold fish in the pond `gF`, the number of silver fish in the pond `sF`, the number of times we catch a fish `n`, the total number of simulation runs `N`, and a policy of whether we throw back (i.e. replace) each caught fish, `withReplacement`, which is set to `false` by default. In lines 4-16 we create an inner function `fishing()` that generates one random instance of a fishing day, returning the number of gold fish caught. Line 5 generates an array, where the values in the array represent fish in the pond, with 0's and 1's representing silver and gold fish respectively. Notice the use of the `zeros()` and `ones()` functions, each with a first argument, `Int64` indicating the Julia type. Line 6 initializes an empty array, which represents the fish to be caught. Lines 8-14 perform the act of fishing `n` times via the use of a `for` loop. Lines 9-10 randomly sample a "fish" from our "pond", and then stores this in value in our `fishCaught` array. Line 12 is only run if `false` is used, in which case we "remove" the caught "fish" from the pond via the function `deleteat!()`. Note that technically we don't remove the exact caught fish, but rather a fish with the same value (0 or 1) via `findfirst()`. Our use of this function returns the first index in `fishInPond` with a value equalling `fished`. Line 15 is the (implicit) return statement for the function `fishing()` and is the sum of how many gold fish were caught (since gold fish are stored as 1's and silver fish as 0's). Line 18 implements our chosen policy N times total, with the total number of gold fish each time stored in the array `simulations`. Line 19 uses the `counts()` function to return the proportion of times $0, \ldots, n$ gold fish were caught. Lines 21-31 then use `plot!()` to overlay the existing plot with the probabilities. The `proportionFished()` function is then called twice in lines 37 and 38 to generate the resulting plot.

Lattice Paths

We now consider a square grid on which an ant walks from the south west corner to the north east corner, taking either a step north or a step east at each grid intersection. This is illustrated in Figure 2.3 where it is clear that there are many possible paths the ant could take. Let us set the sample space to be,

$$\Omega = \text{All possible lattice paths,}$$

where the term *lattice path* describes a trajectory of the ant going from the south west point, $(0, 0)$ to the north east point, (n, n). Since Ω is finite, we can consider the number of elements in it, denoted $|\Omega|$. For a general $n \times n$ grid,

$$|\Omega| = \binom{2n}{n} = \frac{(2n)!}{(n!)^2}.$$

For example if $n = 5$ then $|\Omega| = 252$. The use of the *binomial coefficient* here is because out of the $2n$ steps that the ant needs to take, n steps need to be 'north' and n need to be 'east'.

Within this context of lattice paths, there are a variety of questions. One common question has to do with the event (or set):

$$A = \text{Lattice paths that stay above the diagonal the whole way from } (0, 0) \text{ to } (n, n).$$

The set A then describes all lattice paths where at any point, the ant has not taken more easterly steps than northerly steps. The question of the size of A, namely $|A|$, has interested many people in combinatorics, and it turns out that,

$$|A| = \frac{\binom{2n}{n}}{n+1}.$$

For each counting value of n, the above is called the n'th *Catalan Number*. For example, if $n = 1$ then $|A| = 1$, if $n = 2$, $|A| = 2$ and if $n = 3$ then $|A| = 5$. You can try to sketch all possible paths in A for $n = 3$ (there are 5 in total).

So far we have discussed the sample space Ω, and a potential event A. One interesting question to ask deals with the probability of A. That is: *What is the chance that the ant stays on or above the diagonal as it journey's from* $(0, 0)$ *to* (n, n)*?*

The answer to this question depends on the probability function/measure that we specify for this experiment (sometimes called a *probability model*). There are infinity many choices for the model and the choice of the right model depends on the context. Here we consider two examples:

Model I - As in the previous examples, assume a symmetric probability space, i.e. each lattice path is equally likely. For this model, obtaining probabilities is a question of counting and the result just follows the combinatorial expressions above:

$$\mathbb{P}_{\mathrm{I}}(A) = \frac{|A|}{|\Omega|} = \frac{1}{n+1}. \tag{2.2}$$

Model II - We assume that at each grid intersection where the ant has an option of where to go ('east' or 'north'), it chooses either east or north, both with equal probability 1/2. In the

case where there is no option for the ant (i.e. it hits the east or north border) then it simply continues along the border to the final destination (n, n). For this model, it isn't as simple to obtain an expression for $\mathbb{P}(A)$. One way to do it is by considering a *recurrence relation* for the probabilities (sometimes known as *first step analysis*). We omit the details and present the result:

$$\mathbb{P}_{\mathrm{II}}(A) = \frac{|A|}{|\Omega|} = \frac{\binom{2n-1}{n}}{2^{2n-1}}.$$

Hence we see that the probability of the event, depends on the probability model used - and this choice is not always a straightforward nor obvious one. For example, for $n = 5$ we have,

$$\mathbb{P}_{\mathrm{I}}(A) = \frac{1}{6} \approx 0.166, \qquad \mathbb{P}_{\mathrm{II}}(A) = \frac{126}{512} \approx 0.246.$$

We now verify these values for $\mathbb{P}_{\mathrm{I}}(A)$ and $\mathbb{P}_{\mathrm{II}}(A)$ by simulating both Model I and Model II in Listing 2.5, which also creates Figure 2.3.

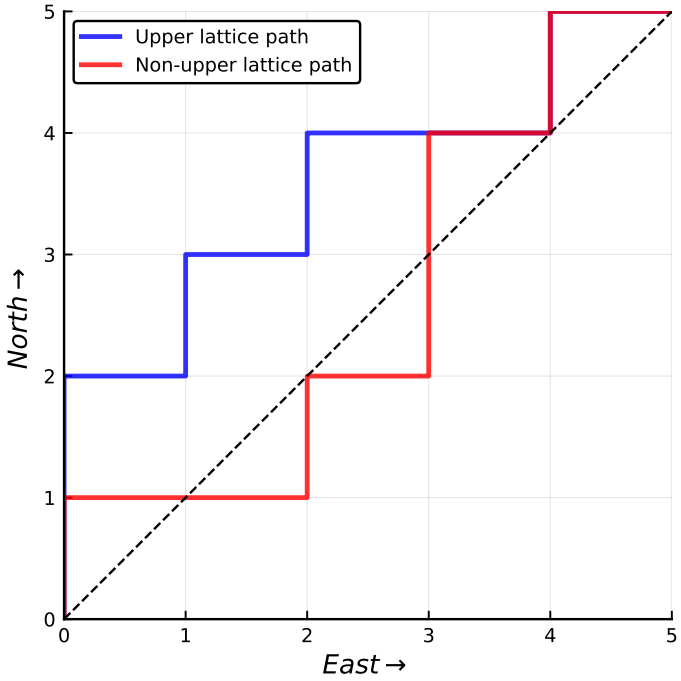

Figure 2.3: Example of two different lattice paths.

Listing 2.5: Lattice paths

```
using Random, Combinatorics, Plots, LaTeXStrings ; pyplot()
Random.seed!(12)

n, N = 5, 10^5

function isUpperLattice(v)
    for i in 1:Int(length(v)/2)
        sum(v[1:2*i-1]) >= i ? continue : return false
    end
    return true
end

omega = unique(permutations([zeros(Int,n);ones(Int,n)]))
A = omega[isUpperLattice.(omega)]
pA_modelI = length(A)/length(omega)

function randomWalkPath(n)
    x, y = 0, 0
    path = []
    while x<n && y<n
       if rand()<0.5
            x += 1
            push!(path,0)
        else
            y += 1
            push!(path,1)
        end
    end
    append!(path, x<n ? zeros(Int64,n-x) : ones(Int64,n-y))
    return path
end

pA_modelIIest = sum([isUpperLattice(randomWalkPath(n)) for _ in 1:N])/N
println("Model I: ",pA_modelI, "\t Model II: ", pA_modelIIest)

function plotPath(v,l,c)
    x,y = 0,0
    graphX, graphY = [x], [y]
    for i in v
        if i == 0
            x += 1
        else
            y += 1
        end
        push!(graphX,x), push!(graphY,y)
    end
    plot!(graphX, graphY,
            la=0.8, lw=2, label=l, c=c, ratio=:equal, legend=:topleft,
            xlims=(0,n), ylims=(0,n),
            xlabel=L"East\rightarrow", ylabel=L"North\rightarrow")
end
plot()
plotPath(rand(A), "Upper lattice path", :blue)
plotPath(rand(setdiff(omega,A)), "Non-upper lattice path", :red)
plot!([0, n], [0,n], ls=:dash, c=:black, label="")
```

```
Model I: 0.16666666666666666     Model II: 0.24696
```

In the code, a path is encoded by a sequence of 0 and 1 values, indicating "move east" or "move north" respectively. The function `isUpperLattice()` defined in lines 5-10 checks if a path is an upper lattice path by summing all the odd partial sums, and returning false if any sum ends up at a coordinate below the diagonal. Note the use of the `? :` operator in line 7 . Also note that in line 6, `Int()` is used to convert the division `length(v)/2` to an integer type. In line 12, a collection of all possible lattice paths is created by applying the `permutations()` function from the `Combinatorics` package to an initial array of `n` zeros and `n` ones. The `unique()` function is then used to remove all duplicates. In line 13 the `isUpperLattice()` function is applied to each element of `omega` via the '`.`' operator just after the function name. The result is a boolean array. Then `omega[]` selects the indices of `omega` where the value is `true` and in the next line `pA_modelI` is calculated. In lines 17-31 the function `randomWalkPath()` is implemented, which creates a random path according to Model II. Note that the code in line 29 appends either zeros or ones to the path, depending on if it hit the north boundary or east boundary first. Then in line 33, the Monte Carlo estimate, `pA_modelIIest` is determined. The function `plotPath()` defined in lines 36-50 plots a path with a specified label and color. It is then invoked in line 52 for an upper lattice path selected via `rand(A)` and again in the next line for a non-upper path by using `setdiff(omega,A)` to determine the collection of non upper lattice paths. Functions dealing with sets are covered in more detail in the next section.

2.2 Working With Sets

As evident from the examples in Section 2.1 above, mathematical *sets* play an integral part in the evaluation of probability models. Subsets of the sample space Ω are also called *events*. By carrying out *intersections*, *unions* and *differences* of sets, we may often express more complicated events based on smaller ones.

A set is an unordered collection of unique *elements*. A set A is a *subset* of the set B if every element that is in A is also an element of B. The *union* of two sets, A and B, denoted $A \cup B$ is the set of all elements that are either in A or B, or both. The *intersection* of the two sets, denoted $A \cap B$, is the set of all elements that are in both A and B. The *difference*, denoted $A \setminus B$ is the set of all elements that are in A but not in B.

In the context of probability, the sample space Ω is often considered as the *universal set*. This allows us to then consider the *complement* of a set A, denoted A^c, which can be constructed via all elements of Ω that are not in A. Note that $A^c = \Omega \setminus A$. Also observe that in the presence of a universal set: $A \setminus B = A \cap B^c$.

Representing Sets in Julia

Julia includes built-in capability for working with sets. Unlike an `Array`, a `Set` is an unordered collection of unique objects. Listing 2.6 illustrates how to construct a `Set` in Juila, and illustrates the use of the `union()`, `intersect()`, `setdiff()`, `issubset()` and `in()` functions. There are also other functions related to sets that you may explore independently. These include `issetequal()` `symdiff()`, `union!()`, `setdiff!()`, `symdiff!()` and `intersect!()`. See the online Julia documentation under "Collections and Data Structures".

Listing 2.6: Basic set operations

```
1  A = Set([2,7,2,3])
2  B = Set(1:6)
3  omega = Set(1:10)
4
5  AunionB = union(A, B)
6  AintersectionB = intersect(A, B)
7  BdifferenceA = setdiff(B,A)
8  Bcomplement = setdiff(omega,B)
9  AsymDifferenceB = union(setdiff(A,B),setdiff(B,A))
10 println("A = $A, B = $B")
11 println("A union B = $AunionB")
12 println("A intersection B = $AintersectionB")
13 println("B diff A = $BdifferenceA")
14 println("B complement = $Bcomplement")
15 println("A symDifference B = $AsymDifferenceB")
16 println("The element '6' is an element of A: $(in(6,A))")
17 println("Symmetric difference and intersection are subsets of the union: ",
18         issubset(AsymDifferenceB,AunionB),", ", issubset(AintersectionB,AunionB))
```

```
A = Set([7, 2, 3]), B = Set([4, 2, 3, 5, 6, 1])
A union B = Set([7, 4, 2, 3, 5, 6, 1])
A intersection B = Set([2, 3])
B diff A = Set([4, 5, 6, 1])
B complement = Set([7, 9, 10, 8])
A symDifference B = Set([7, 4, 5, 6, 1])
The element '6' is an element of A: false
Symmetric difference and intersection are subsets of the union: true, true
```

In lines 1-3 three different sets are created via the `Set()` function (a constructor). Note that `A` contains only three elements, since sets are meant to be a collection of unique elements. Also note that unlike arrays order is not preserved. Lines 5-9 perform various operations using the sets created. Lines 10-18 create the listing output. Note the use of the functions `in()` and `issubset()` in lines 16-18.

The Probability of a Union

Consider now two events (sets) A and B. If $A \cap B = \emptyset$, then $\mathbb{P}(A \cup B) = \mathbb{P}(A) + \mathbb{P}(B)$. However more generally, when A and B are not *disjoint*, the probability of the *intersection*, $A \cap B$ plays a role. For such cases the *inclusion exclusion formula* is useful:

$$\mathbb{P}(A \cup B) = \mathbb{P}(A) + \mathbb{P}(B) - \mathbb{P}(A \cap B). \tag{2.3}$$

To help illustrate this, consider the simple example of choosing a random lower case letter, 'a'-'z'. Let A be the event that the letter is a vowel (one of 'a', 'e', 'i', 'o', 'u'). Let B be the event that the letter is one of the first three letters (one of 'a', 'b', 'c'). Now since $A \cap B = \{$'a'$\}$, a set with one element, we have,

$$\mathbb{P}(A \cup B) = \frac{5}{26} + \frac{3}{26} - \frac{1}{26} = \frac{7}{26}.$$

For another similar example, consider the case where A is the set of vowels as before, but $B = \{$'x', 'y', 'z'$\}$. In this case, since the intersection of A and B is empty, we immediately know that

$\mathbb{P}(A \cup B) = (5+3)/26 \approx 0.3077$. While this example is elementary, we now use it to illustrate a type of conceptual error that one may make when using Monte Carlo simulation.

Consider code Listing 2.7, and compare `mcEst1` and `mcEst2` from lines 12 and 13 respectively. Both variables are designed to be estimators of $\mathbb{P}(A \cup B)$. However, one of them is a correct estimator and the other is faulty. In the following we look at the output given from of both, and explore the fault in the underlying logic.

Listing 2.7: An innocent mistake with Monte Carlo

```
1  using Random, StatsBase
2  Random.seed!(1)
3
4  A = Set(['a','e','i','o','u'])
5  B = Set(['x','y','z'])
6  omega = 'a':'z'
7
8  N = 10^6
9
10 println("mcEst1 \t \tmcEst2")
11 for _ in 1:5
12     mcEst1 = sum([in(sample(omega),A) || in(sample(omega),B) for _ in 1:N])/N
13     mcEst2 = sum([in(sample(omega),union(A,B)) for _ in 1:N])/N
14     println(mcEst1,"\t",mcEst2)
15 end
```

First observe line 12. In Julia, `||` means "or", so at first glance the estimator `mcEst1` looks sensible, since:

$$A \cup B = \text{the set of all elements that are in } A \text{ or } B.$$

Hence we are generating a random element via `sample(omega)` and checking if it is an element of A or an element of B. However there is a subtle error. Each of the N random experiments involves two separate calls to `sample(omega)`. Hence the code in line 12 simulates a situation where conceptually, the sample space, Ω is composed of pairs of letters (2-tuples), not single letters!

Hence the code computes probabilities of the event, $A_1 \cup B_2$ where,

$$A_1 = \text{First element of the tuple is a vowel,}$$
$$B_2 = \text{Second element of the tuple is an 'x', 'y', or 'z' letter.}$$

Now observe that A_1 and B_2 are not disjoint events, hence,

$$\mathbb{P}(A_1 \cup B_2) = \mathbb{P}(A_1) + \mathbb{P}(B_2) - \mathbb{P}(A_1 \cap B_2).$$

Further it holds that $\mathbb{P}(A_1 \cap B_2) = \mathbb{P}(A_1)\mathbb{P}(B_2)$. This follows from independence (further explored in Section 2.3). Now that we have identified the error, we can predict the resulting output.

$$\mathbb{P}(A_1 \cup B_2) = \mathbb{P}(A_1) + \mathbb{P}(B_2) - \mathbb{P}(A_1)\mathbb{P}(B_2) = \frac{5}{26} + \frac{3}{26} - \frac{5}{26}\frac{3}{26} \approx 0.2855.$$

It can be seen from the code output, which repeats the comparison 5 times, that `mcEst1` consistently underestimates the desired probability, yielding estimates near 0.2855 instead.

```
mcEst1          mcEst2
0.285158        0.307668
0.285686        0.307815
0.285022        0.308132
0.285357        0.307261
0.285175        0.306606
```

In lines 11-15 a `for` loop is implemented, which generates 5 Monte Carlo predictions. Note that lines 12 and 13 contain the main logic of this example. Line 12 is our incorrect simulation, and yields incorrect estimates. See the text above for a detailed explanation as to why the use of two separate calls to `sample()` are incorrect in this case. Line 13 is our correct simulation, and for large `N` yields results close to the expected result. Note that the `union()` function is used on `A` and `B`, instead of the "or" operator, `||`, used in line 12. The important point is that only a single sample is generated for each iteration of the composition.

Secretary with Envelopes

Now consider a more general form of the *inclusion exclusion principle* applied to a collection of sets, $C_1, \ldots, C_n$. It is presented below, written in two slightly different forms:

$$\mathbb{P}\Big(\bigcup_{i=1}^{n} C_i\Big) = \sum_{i=1}^{n} \mathbb{P}(C_i) - \sum_{\text{pairs}} \mathbb{P}(C_i \cap C_j) + \sum_{\text{triplets}} \mathbb{P}(C_i \cap C_j \cap C_k) - \ \ldots \ + (-1)^{n-1}\mathbb{P}(C_1 \cap \ldots \cap C_n)$$

$$= \sum_{i=1}^{n} \mathbb{P}(C_i) \ - \sum_{i<j} \mathbb{P}(C_i \cap C_j) \ + \sum_{i<j<k} \mathbb{P}(C_i \cap C_j \cap C_k) \ - \ \ldots \ + \ (-1)^{n-1} \ \mathbb{P}\Big(\bigcap_{i=1}^{n} C_i\Big).$$

Notice that there are n major terms. The first term deals with probabilities of individual events; the second term deals with pairs; the third with triplets; and the sequence continues until a single final term involving a single intersection is reached. The ℓ'th term has $\binom{n}{\ell}$ summands. For example, there are $\binom{n}{2}$ pairs, $\binom{n}{3}$ triplets, etc. Notice also the alternating signs via $(-1)^{\ell-1}$. It is possible to conceptually see the validity of this formula for the case of $n = 3$ by drawing a *Venn diagram* and seeing the role of all summands. In this case,

$$\mathbb{P}\big(C_1 \cup C_2 \cup C_3\big) = \mathbb{P}\big(C_1\big) + \mathbb{P}\big(C_2\big) + \mathbb{P}\big(C_3\big) - \mathbb{P}\big(C_1 \cap C_2\big) - \mathbb{P}\big(C_1 \cap C_3\big) - \mathbb{P}\big(C_2 \cap C_3\big) + \mathbb{P}\big(C_1 \cap C_2 \cap C_3\big).$$

Let us now consider a classic example that uses this inclusion exclusion principle. Assume that a secretary has an equal number of pre-labelled envelopes and business cards, n. Suppose that at the end of the day, he is in such a rush to go home that he puts each business card in an envelope at random without any thought of matching the business card to its intended recipient on the envelope. The probability that each of the business cards will go to the correct envelope is easy to obtain. It is $1/n!$, which goes to zero very quickly as n grows. However, what is the probability that each of the business cards will go to a wrong envelope?

As an aid, let A_i be the event that the i'th business card is put in the correct envelope. We have a handle on events involving intersections of distinct A_i values. For example, if $n = 10$, then $\mathbb{P}(A_1 \cap A_4 \cap A_6) = 7!/10!$, or more generally, the probability of an intersection of k such events is $p_k := (n-k)!/n!$.

The event we are seeking to evaluate is, $B = A_1^c \cap A_2^c \cap \ldots \cap A_n^c$. Hence by *De Morgan's laws*, $B^c = A_1 \cup \ldots \cup A_n$. Hence using the inclusion exclusion formula together with p_k, we can simplify factorials and binomial coefficients to obtain:

$$\mathbb{P}(B) = 1 - \mathbb{P}(A_1 \cup \ldots \cup A_n) = 1 - \sum_{k=1}^{n} (-1)^{k+1} \binom{n}{k} p_k = 1 - \sum_{k=1}^{n} \frac{(-1)^{k+1}}{k!} = \sum_{k=0}^{n} \frac{(-1)^k}{k!}. \quad (2.4)$$

Observe that as $n \to \infty$ this probability converges to $1/e \approx 0.3679$, yielding a simple *asymptotic approximation*. Listing 2.8 evaluates $\mathbb{P}(B)$ in several alternative ways for $n = 1, 2, \ldots, 8$. The function `bruteSetsProbabilityAllMiss()` works by creating all possibilities and counting. Although a highly inefficient way of evaluating $\mathbb{P}(B)$, it is presented here as it is instructive. The function `formulaCalcAllMiss()` evaluates the analytic solution from (2.4). Finally, the function `mcAllMiss()` estimates the probability via Monte Carlo simulation.

Listing 2.8: Secretary with envelopes

```
using Random, StatsBase, Combinatorics
Random.seed!(1)

function bruteSetsProbabilityAllMiss(n)
    omega = collect(permutations(1:n))
    matchEvents = []
    for i in 1:n
        event = []
        for p in omega
            if p[i] == i
                push!(event,p)
            end
        end
        push!(matchEvents,event)
    end
    noMatch = setdiff(omega,union(matchEvents...))
    return length(noMatch)/length(omega)
end

formulaCalcAllMiss(n) = sum([(-1)^k/factorial(k) for k in 0:n])

function mcAllMiss(n,N)
    function envelopeStuffer()
        envelopes = Random.shuffle!(collect(1:n))
        return sum([envelopes[i] == i for i in 1:n]) == 0
    end
    data = [envelopeStuffer() for _ in 1:N]
    return sum(data)/N
end

N = 10^6

println("n\tBrute Force\tFormula\t\tMonte Carlo\tAsymptotic",)
for n in 1:6
    bruteForce = bruteSetsProbabilityAllMiss(n)
    fromFormula = formulaCalcAllMiss(n)
    fromMC = mcAllMiss(n,N)
    println(n,"\t",round(bruteForce,digits=4),"\t\t",round(fromFormula,digits=4),
        "\t\t",round(fromMC,digits=4),"\t\t",round(1/MathConstants.e,digits=4))
end
```

n	Brute Force	Formula	Monte Carlo	Asymptotic
1	0.0	0.0	0.0	0.3679
2	0.5	0.5	0.4994	0.3679
3	0.3333	0.3333	0.3337	0.3679
4	0.375	0.375	0.3747	0.3679
5	0.3667	0.3667	0.3665	0.3679
6	0.3681	0.3681	0.3678	0.3679

Lines 4-18 define the function `bruteSetsProbabilityAllMiss()`, which uses a brute force approach to calculate $\mathbb{P}(B)$. The nested loops in lines 7-15 populate the array `matchEvents` with elements of `omega` that have a match. The inner loop in lines 9-13, puts elements from `omega` in `event` if they satisfy an i'th match. In line 16, notice the use of the 3 dots *splat operator*, `...`. Here `union()` is applied to all the elements of `matchEvents`. The return value in line 17 is a direct implementation via counting the elements of `noMatch`. The function on line 20 implements (2.4) in straightforward manner. Lines 22-29 implement the function `mcAllMiss()` that estimates the probability via Monte Carlo. The inner function, `envelopeStuffer()` returns a result from a single experiment. Note that `shuffle!()` is used to create a random permutation in line 24. The remainder of the code prints the output, and compares the results to the asymptotic formula obtained via `1/MathConstants.e`.

An Occupancy Problem

We now consider a problem related to the previous example. Imagine now the secretary placing r identical business cards randomly into n envelopes, with $r \geq n$ and no limit on the number of business cards that can fit in an envelope. We now ask what is the probability that all envelopes are non-empty (i.e. occupied)?

To begin, denote A_i as the event that the i'th envelope is empty, and hence A_i^c is the event that the i'th envelope is occupied. Hence as before, we are seeking the probability of the event $B = A_1^c \cap A_2^c \cap \ldots \cap A_n^c$. Using the same logic as in the previous example,

$$\begin{aligned}\mathbb{P}(B) &= 1 - \mathbb{P}(A_1 \cup \ldots \cup A_n) \\ &= 1 - \sum_{k=1}^{n} (-1)^{k+1} \binom{n}{k} \tilde{p}_k,\end{aligned}$$

where $\tilde{p}_k$ is the probability of at least k envelopes being empty. Now from basic counting considerations,

$$\tilde{p}_k = \frac{(n-k)^r}{n^r} = \left(1 - \frac{k}{n}\right)^r.$$

Thus we arrive at,

$$\mathbb{P}(B) = 1 - \sum_{k=1}^{n} (-1)^{k+1} \binom{n}{k} \left(1 - \frac{k}{n}\right)^r = \sum_{k=0}^{n} (-1)^k \binom{n}{k} \left(1 - \frac{k}{n}\right)^r. \tag{2.5}$$

We now calculate $\mathbb{P}(B)$ in Listing 2.9 and compare the results to Monte Carlo simulation estimates. In the code we consider several situations by varying the number of envelopes in the range $n = 1, \ldots, 100$, and for every n, consider the number of business cards $r = Kn$ for $K = 2, 3, 4$. The results are displayed in Figure 2.4.

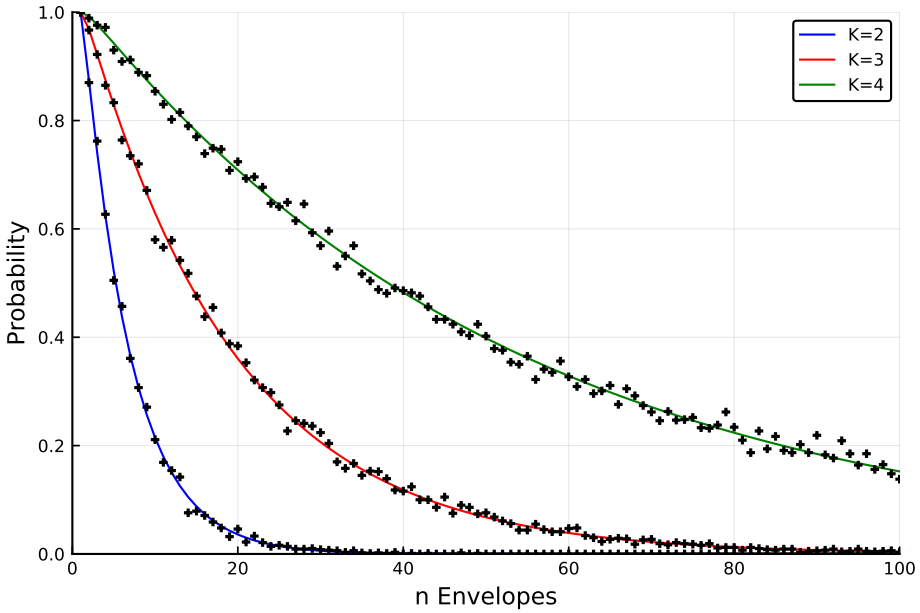

Figure 2.4: Analytic and estimated probabilities that no envelopes are empty, for various cases of n envelopes, and Kn business cards.

Listing 2.9: An occupancy problem

```
using Plots ; pyplot()

occupancyAnalytic(n,r) =  sum([(-1)^k*binomial(n,k)*(1 - k/n)^r for k in 0:n])

function occupancyMC(n,r,N)
    fullCount = 0
    for _ in 1:N
        envelopes = zeros(Int,n)
        for k in 1:r
            target = rand(1:n)
            envelopes[target] += 1
        end
        numFilled = sum(envelopes .> 0)
        if numFilled == n
            fullCount += 1
        end
    end
    return fullCount/N
end

max_n, N, Kvals = 100, 10^3, [2,3,4]

analytic = [[occupancyAnalytic(big(n),big(k*n)) for n in 1:max_n] for k in Kvals]
monteCarlo = [[occupancyMC(n,k*n,N) for n in 1:max_n] for k in Kvals]

plot(1:max_n, analytic, c=[:blue :red :green],
        label=["K=2" "K=3" "K=4"])
scatter!(1:max_n, monteCarlo, mc=:black, shape=:+,
        label="", xlims=(0,max_n),ylims=(0,1),
        xlabel="n Envelopes", ylabel="Probability", legend=:topright)
```

In line 3 we create the function `occupancyAnalytic()`, which evaluates (2.5). Note the use of the `binomial()` function. Lines 5-19 define the function `occupancyMC()`, which approximates $\mathbb{P}(B)$ for specific inputs via Monte Carlo simulation. Note the additional argument `N`, which is the total number of simulation runs. Line 5 defines the variable `fullcount`, which represents the total number of times all envelopes are full. Lines 7-17 contain the core logic of this function, and represent the act of the secretary assigning all business cards randomly to the envelopes, and repeating this process `N` times total. Observe that in this `for` loop, there is no need to keep a count of the loop iteration number, hence for clarity we use underscore in line 7. Line 13 checks each element of `envelopes` to see if they are empty (i.e 0), and evaluates the total number of envelopes which are not empty. Note the use of element-wise comparison `.>`, resulting in an array of boolean values that can be summed. Lines 14-16 checks if all envelopes have been filled, and if so increments `fullCount` by 1. In lines 23 and 24 we create `analytic` and `monteCarlo` respectively. Each of these is an array of arrays, with an internal array for `k=2`, `k=3` and `k=4`. The results are then plotted.

2.3 Independence

We now consider *independence* and *independent events*. Two events, A and B, are said to be independent if the probability of their *intersection* is the product of their probabilities:

$$\mathbb{P}(A \cap B) = \mathbb{P}(A)\mathbb{P}(B).$$

A classic example is a situation where a random experiment involves physical components that are assumed to not interact, for example flipping two coins. Independence is often a modeling assumption and plays a key role in many models presented in the remainder of the book.

Note that "independent events" should not be confused with "disjoint events". However, these concepts are completely different. Take disjoint events A, and B, with $\mathbb{P}(A) > 0$ and $\mathbb{P}(B) > 0$. This means that $\mathbb{P}(A)\mathbb{P}(B) > 0$. It is easy to see that the events are not independent. Since they are disjoint, $A \cap B = \emptyset$ and $\mathbb{P}(\emptyset) = 0$, however,

$$0 = \mathbb{P}(\emptyset) = \mathbb{P}(A \cap B) \neq \mathbb{P}(A)\mathbb{P}(B).$$

To explore independence, it is easiest to consider a situation where it does not hold. Consider drawing a number uniformly from the range $10, 11, \ldots, 25$. What is the probability of getting the number 13? Clearly there are $25 - 10 + 1 = 16$ options, and hence the probability is $1/16 = 0.0625$. However, the event of obtaining 13 could be described as the intersection of the events $A :=$ {first digit is 1} and $B :=$ {second digit is 3}. The probabilities of which are $10/16 = 0.625$ and $2/16 = 0.125$ respectively. Notice that the product of these probabilities is not 0.0625, but rather $20/256 = 0.078125$. Hence we see that, $\mathbb{P}(AB) \neq \mathbb{P}(A)\mathbb{P}(B)$ and the events are not independent.

One way of viewing this lack of independence is as follows. Witnessing the event A gives us some information about the likelihood of B. Since if A occurs, we know that the number is in the range $10, \ldots, 19$ and hence there is a $1/10$ chance for B to occur. However, if A does not occur then we lie in the range $20, \ldots, 25$ and there is a $1/6$ chance for B to occur.

If however we change the range of random digits to be $10, \ldots, 29$ then the two events are independent. This can be demonstrated by running Listing 2.10, and then modifying line 4.

Listing 2.10: Independent events

```
1  using Random
2  Random.seed!(1)
3
4  numbers = 10:25
5  N = 10^7
6
7  firstDigit(x) = Int(floor(x/10))
8  secondDigit(x) = x%10
9
10 numThirteen, numFirstIsOne, numSecondIsThree = 0, 0, 0
11
12 for _ in 1:N
13     X = rand(numbers)
14     global numThirteen += X == 13
15     global numFirstIsOne += firstDigit(X) == 1
16     global numSecondIsThree += secondDigit(X) == 3
17 end
18
19 probThirteen, probFirstIsOne, probSecondIsThree =
20     (numThirteen,numFirstIsOne,numSecondIsThree)./N
21
22 println("P(13) = ", round(probThirteen, digits=4),
23         "\nP(1_) = ",round(probFirstIsOne, digits=4),
24         "\nP(_3) = ", round(probSecondIsThree, digits=4),
25         "\nP(1_)*P(_3) = ",round(probFirstIsOne*probSecondIsThree, digits=4))
```

```
P(13) = 0.0626
P(1_) = 0.6249
P(_3) = 0.1252
P(1_)*P(_3) = 0.0783
```

Lines 4 and 5 set the range of numbers considered and the number of simulation runs respectively. Line 7 defines a function that returns the first digit of our number through the use of the `floor()` function, and converts the resulting value to an integer type. Line 8 defines a function that uses the *modulus* operator `%` to return the second digit of our number. In line 10 we initialize three placeholder variables, which represent the number chosen, and its first and second digits respectively. Lines 12-17 contain the core logic of this example, where `N` random digits are generated. For each random digit, `X` that is generated, lines 14, 15 and 16 increment the count by 1 if the specified condition is met. Line 19-20 evaluate the total proportions.

2.4 Conditional Probability

It is often the case that knowing an event has occurred, say B, modifies our belief about the chances of another event occurring, say A. This concept is captured via the *conditional probability* of A given B, denoted by $\mathbb{P}(A \mid B)$ and defined for B where $\mathbb{P}(B) > 0$. In practice, given a probability model, $\mathbb{P}(\cdot)$, we construct the conditional probability, $\mathbb{P}(\cdot \mid B)$ via,

$$\mathbb{P}(A \mid B) := \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}. \tag{2.6}$$

This immediately shows that if events A and B are independent then $P(A \mid B) = P(A)$.

As an elementary example, refer back to Table 2.1 depicting the outcome of rolling two dice. Set B as the event of the sum being greater than or equal to 10. In other words,

$$B = \{(i, j) \mid i + j \geq 10\}.$$

To help illustrate this further, consider a game player who rolls the dice without showing us the result, and then poses to us the following: "The sum is greater or equal to 10. Is it even or odd?". Let A be the event of the sum being even. We then evaluate,

$$\mathbb{P}(A \mid B) = \mathbb{P}(\text{Sum is even} \mid B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)} = \frac{\mathbb{P}(\text{Sum is 10 or 12})}{\mathbb{P}(\text{Sum is } \geq 10)} = \frac{4/36}{6/36} = \frac{2}{3},$$
$$\mathbb{P}(A^c \mid B) = \mathbb{P}(\text{Sum is odd} \mid B) = \frac{\mathbb{P}(A^c \cap B)}{\mathbb{P}(B)} = \frac{\mathbb{P}(\text{Sum is 11})}{\mathbb{P}(\text{Sum is } \geq 10)} = \frac{2/36}{6/36} = \frac{1}{3}.$$

It can be seen that given B, it is more likely that A occurs (even) as opposed to A^c (odd), hence we are better off answering "even".

The Law of Total Probability

Often our probability model is comprised of conditional probabilities as elementary building blocks. In such cases, (2.6) is better viewed as,

$$\mathbb{P}(A \cap B) = \mathbb{P}(B)\ \mathbb{P}(A \mid B).$$

This is particularly useful when there exists some *partition* of Ω, namely, $\{B_1, B_2, \ldots\}$. A partition of a set U is a collection of non-empty sets that are mutually disjoint and whose union is U. Such a partition allows us to represent A as a disjoint union of the sets $A \cap B_k$, and treat $\mathbb{P}(A \mid B_k)$ as model data. In such a case, we have the *law of total probability*,

$$\mathbb{P}(A) = \sum_{k=0}^{\infty} \mathbb{P}(A \cap B_k) = \sum_{k=0}^{\infty} \mathbb{P}(A \mid B_k)\ \mathbb{P}(B_k).$$

As an exotic fictional example, consider the world of semi-conductor manufacturing. Room cleanliness in the manufacturing process is critical, and dust particles are kept to a minimum. Let A be the event of a manufacturing failure, and assume that it depends on the number of dust particles via,

$$\mathbb{P}(A \mid B_k) = 1 - \frac{1}{k+1},$$

where B_k is the event of having k dust particles in the room ($k = 0, 1, 2, \ldots$). Clearly the larger k, the higher the chance of manufacturing failure. Furthermore assume that,

$$\mathbb{P}(B_k) = \frac{6}{\pi^2 (k+1)^2} \qquad \text{for } k = 0, 1, \ldots.$$

From the well known *Basel Problem*, we have $\sum_{k=1}^{\infty} k^{-2} = \pi^2/6$. This implies that $\sum_k \mathbb{P}(B_k) = 1$.

Now we ask, what is the probability of manufacturing failure? The analytic solution is given by,

$$\mathbb{P}(A) = \sum_{k=0}^{\infty} \mathbb{P}(A \mid B_k)\, \mathbb{P}(B_k) = \sum_{k=0}^{\infty} \left(1 - \frac{1}{k+1}\right) \frac{6}{\pi^2 (k+1)^2}.$$

With some calculus, the infinite series can be explicitly evaluated to,

$$\mathbb{P}(A) = 1 - \frac{6\,\zeta(3)}{\pi^2} \approx 0.2692,$$

where $\zeta(\cdot)$ is the *Riemann Zeta Function*,

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s},$$

and $\zeta(3) \approx 1.2021$. Note that the appearance of $\zeta(\cdot)$ in this example is by design due to the fact that we chose $\mathbb{P}(A \mid B_k)$ and $\mathbb{P}(B_k)$ to have the specific structure. Listing 2.11 approximates the infinite series numerically (truncating at `n = 2000`) and compares the result to the analytic solution.

Listing 2.11: Defects in manufacturing

```
1  using SpecialFunctions
2
3  n = 2000
4
5  probAgivenB(k) = 1- 1/(k+1)
6  probB(k) = 6/(pi*(k+1))^2
7
8  numerical= sum([probAgivenB(k)*probB(k) for k in 0:n])
9  analytic = 1 - 6*zeta(3)/pi^2
10
11 println("Analytic: ", analytic, "\tNumerical: ", numerical)
```

```
Analytic: 0.26923703059856086   Numerical: 0.26893337073278945
```

This listing is self-explanatory, however note the use of the Julia function `zeta()` from the `SpecialFunctions` package in line 9. Note also that `pi` is a defined constant.

2.5 Bayes' Rule

Bayes' rule, also known as *Bayes' theorem*, is nothing but a simple manipulation of (2.6) yielding,

$$\mathbb{P}(A \mid B) = \frac{\mathbb{P}(B \mid A)\mathbb{P}(A)}{\mathbb{P}(B)}. \tag{2.7}$$

However, the consequences are far reaching. Often we observe a *posterior outcome* or measurement, say the event B, and wish to evaluate the probability of a *prior condition*, say the event A. That is, given some measurement or knowledge we wish to evaluate how likely is it that a prior condition occurred. Equation (2.7) allows us to do just that.

Was it a *0* or a *1*?

As an example, consider a communication channel involving a stream of transmitted bits (0's and 1's), where 70% of the bits are 1, and the rest 0. A typical snippet from the channel ...0101101011101111101....

The channel is imperfect due to physical disturbances such as interfering radio signals, and furthermore the bits received are sometimes distorted. Hence there is a chance (ε_0) of interpreting a bit as 1 when it is actually 0, and similarly, there is a chance (ε_1) of interpreting a bit as 0 when it is actually 1.

Now say that we received (Rx) a bit, and interpreted it as 1. This is the posterior outcome. What is the chance that it was in-fact transmitted (Tx) as a 1? Applying Bayes' rule:

$$\mathbb{P}(\text{Tx } 1 \mid \text{Rx } 1) = \frac{\mathbb{P}(\text{Rx } 1 \mid \text{Tx } 1)\mathbb{P}(\text{Tx } 1)}{\mathbb{P}(\text{Rx } 1)} = \frac{(1-\varepsilon_1)0.7}{0.7(1-\varepsilon_1)+0.3\varepsilon_0}. \tag{2.8}$$

For example, if $\varepsilon_0 = 0.1$ and $\varepsilon_1 = 0.05$ we have that $\mathbb{P}(\text{Tx } 1 \mid \text{Rx } 1) = 0.9568$. Listing 2.12 illustrates this via simulation.

Listing 2.12: Tx Rx Bayes

```
using Random
Random.seed!(1)

N = 10^5
prob1 = 0.7
eps0, eps1 = 0.1, 0.05

flipWithProb(bit,prob) = rand() < prob ? xor(bit,1) : bit

TxData = rand(N) .< prob1
RxData = [x == 0 ? flipWithProb(x,eps0) : flipWithProb(x,eps1) for x in TxData]

numTx1 = 0
totalRx1 = 0
for i in 1:N
   if RxData[i] == 1
        global totalRx1 += 1
        global numTx1 += TxData[i]
    end
end

monteCarlo = numTx1/totalRx1
analytic = ((1-eps1)*0.7)/((1-eps1)*0.7+0.3*eps0)

println("Monte Carlo: ", monteCarlo, "\t\tAnalytic: ", analytic)
```

```
Monte Carlo: 0.9576048007598325         Analytic: 0.9568345323741007
```

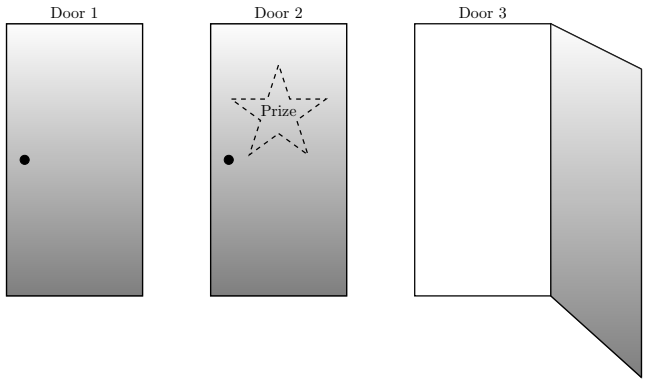

Figure 2.5: Monty Hall: If the prize is behind Door 2 and Door 1 is chosen, the game show host must reveal Door 3.

In lines 8 the function `flipWithProb()` is defined. It uses the `xor()` function to randomly flip the input argument `bit`, according to the rate given by the argument `prob`. Line 10 generates the array `TxData`, which contains true and false values representing our transmitted bits of 1's and 0's respectively. It does this by uniformly and randomly generating numbers on the range $[0, 1]$, and then evaluating element-wise if they are less than the specified probability of receiving a 1, `prob1`. Line 11 generates the array `RxData`, which represents our simulated received data. First the type of received bit is checked, and the `flipWithProb()` function is used to flip received bits at the rates specified in line 6 if the received bit is a 0 or 1. Lines 13-20 are used to check the nature of all bits. If the bit received is 1, then it increments the counter `totalRx1` by 1. It also increments the counter `numTx1` by the value of the transmitted bit (which may be 1, but could also be 0). The remaining lines then calculate the Monte Carlo based estimate and compare to the analytic solution from (2.8).

The Monty Hall Problem

The *Monty Hall problem* is a famous problem which was first posed and solved in 1975 by the mathematician Steve Selvin [SBK75]. It is a famous example illustrating how probabilistic reasoning may sometimes yield to surprising results.

Consider a contestant on a television game show, with three doors in front of her. One of the doors contains a prize, while the other two are empty. The contestant is then asked to guess which door contains the prize, and she makes a random guess. Following this, the game show host (GSH) reveals an empty (losing) door from one of the two remaining doors not chosen. The contestant is then asked if she wishes to stay with their original choice, or if she wishes to switch to the remaining closed door. Following the choice of the contestant to stay or switch, the door with the prize is revealed. The question is: should the contestant stay with their original choice, or switch? Alternatively, perhaps it doesn't matter.

For example, in Figure 2.5 we see the situation where the hidden prize is behind door 2. Say the contestant has chosen door 1. In this case, the GSH has no choice but to reveal door 3. Alternatively, if the contestant has chosen door 2, then the GSH will reveal either door 1 or door 3.

The two possible policies (or strategies of play) for the contestant are:

Policy I - Stay with their original choice after the door is revealed.

Policy II- Switch after the door is revealed.

Let us consider the probability of winning for the two different policies. If the player adopts Policy I then she always stays with her initial guess regardless of the GSH action. In this case, her chance of success is 1/3; that is, she wins if her initial choice is correct.

However if she adopts Policy II then she always switches after the GSH reveals an empty door. In this case we can show that her chance of success is 2/3; that is, she actually wins if her initial guess is incorrect. This is because the GSH must always reveal a losing door. If she originally chose a losing door, then the GSH must reveal the second losing door every time (otherwise he would reveal the prize). That is, if the player chooses an incorrect door at the start, the non-revealed door will always be the winning door. The chance of such an event is 2/3.

As a further aid for understanding imagine a case of 100 doors and a single prize behind one of them. In this case assume that the player chooses a door, for example door 1, and following this the GSH reveals 98 losing doors. There are now only two doors remaining, her choice door 1, and (say for example), door 38. The intuition of the problem suddenly becomes obvious. The player's original guess was random and hence door 1 had a 1/100 chance of containing the prize, however the GSH's actions were constrained. He had to reveal only losing doors, and hence there is a 99/100 chance that door 38 contains the prize. Hence, Policy II is clearly superior.

We now analyze the case of 3 doors by applying Bayes' theorem. Let A_i be the event that the prize is behind door i. Let B_i be the event that door i is revealed by the GSH. Then, for example, if the player initially chooses door 1 and then the GSH reveals door 2, we have the following:

$$\mathbb{P}(A_1 \mid B_2) = \frac{\mathbb{P}(B_2 \mid A_1)\mathbb{P}(A_1)}{\mathbb{P}(B_2)} = \frac{\frac{1}{2} \times \frac{1}{3}}{\frac{1}{2}} = \frac{1}{3}, \qquad \textbf{(Policy I)}$$

$$\mathbb{P}(A_3 \mid B_2) = \frac{\mathbb{P}(B_2 \mid A_3)\mathbb{P}(A_3)}{\mathbb{P}(B_2)} = \frac{1 \times \frac{1}{3}}{\frac{1}{2}} = \frac{2}{3}. \qquad \textbf{(Policy II)}$$

In the second case note that $\mathbb{P}(B_2 \mid A_3) = 1$ because the GSH must reveal door 2 if the prize is behind door 3 since door 1 was already picked. Hence, we see that while neither policy guarantees a win, Policy II clearly dominates Policy I.

Now that we have shown this analytically, we perform a Monte Carlo simulation of the Monty Hall problem in Listing 2.13.

Listing 2.13: The Monty Hall problem

```
1  using Random
2  Random.seed!(1)
3
4  function montyHall(switchPolicy)
5      prize, choice = rand(1:3), rand(1:3)
6      if prize == choice
7          revealed = rand(setdiff(1:3,choice))
8      else
9          revealed = rand(setdiff(1:3,[prize,choice]))
10     end
11
12     if switchPolicy
13         choice = setdiff(1:3,[revealed,choice])[1]
14     end
15     return choice == prize
16 end
17
18 N = 10^6
19 println("Success probability with policy I (stay): ",
20         sum([montyHall(false) for _ in 1:N])/N)
21 println("Success probability with policy II (switch): ",
22         sum([montyHall(true) for _ in 1:N])/N)
```

```
Success probability with policy I (stay): 0.332913
Success probability with policy II (switch): 0.667027
```

In lines 4-16 the function `montyHall()` is defined, which performs one simulation run of the problem given a policy, with `false` indicating policy I and `true` indicating policy II (switching). At the start of the game, the location of the `prize` and the player's door `choice` are uniformly and randomly initialized. Lines 6-10 contain the logic and action of the GSH. Since he knows the location of both the prize and the chosen door, he first mentally checks if they are the same. If they are, he reveals a door according to line 7. If not, then he proceeds to reveal a door according to the logic in line 9. In either case, the revealed door is stored in the variable `revealed`. Line 7 represents his action if the initial `choice` door is the same as the `prize` door. In this case, he is free to reveal either of the remaining two doors, i.e. the set difference between all doors and the player's `choice` door. In this case the set difference has 2 elements. Line 9 represents the GSH action if the `choice` door is different to the `prize` door. In this case, his hand is forced. As he cannot reveal the player's chosen door or the prize door, he is forced to reveal the one remaining door, which can be thought of as the set difference between `1:3` (all doors) and `[prize, choice]`. In this case the set difference has a single element. Line 13 represents the contestant's action, after the GSH revelation, based on either a switch (`true`) or stay (`false`) policy. If the contestant chooses to stay with her initial guess (`false`), then we skip to Line 15. However, if she chooses to swap (`true`), then we reassign our initial `choice` to the one remaining door in line 13. Note the use of `[1]`, which is used to assign the value from the array to `choice`, rather than the array itself. Line 15 checks if the player's choice is the same as the prize, and returns `true` if she wins, or `false` if she loses. Lines 19-22 repeat this experiment `N` times for each of the policies and print the Monte Carlo estimates.

Chapter 3

Probability Distributions - DRAFT

In this chapter, we introduce random variables, different types of distributions and related concepts. In the previous chapter we explored probability spaces without much emphasis on numerical random values. However, when carrying out random experiments, there are almost always numerical values involved. In the context of probability, these values are often called *random variables*. Mathematically, a random variable X is a function of the sample space, Ω, and takes on integer, real, complex, or even a vector of values. That is, for every possible outcome $\omega \in \Omega$, there is some possible outcome, $X(\omega)$.

The chapter is organized as follows: In Section 3.1 we introduce the concept of a random variable and its probability distribution. In Section 3.2 we introduce the mean, variance and other numerical descriptors of probability distributions. In Section 3.3 we explore several alternative functions for describing probability distributions. In Section 3.4 we focus on Julia's `Distributions` package which is useful when working with probability distributions. Then Section 3.5 explores a variety of discrete distributions. This is followed by Section 3.6 where we explore some continuous distributions together with additional concepts such as hazard rates and more. We close with Section 3.7, where we explore multi-dimensional probability distributions.

3.1 Random Variables

As an example, consider a sample space Ω which consists of 6 names. Assume that the probability function (or probability measure), $\mathbb{P}(\cdot)$, assigns uniform probabilities to each of the names. Let now, $X : \Omega \to \mathbb{Z}$, be the function (i.e. random variable) that counts the number of letters in each name. The question is then finding:

$$p(x) := \mathbb{P}\big(X = x\big), \qquad \text{for} \qquad x \in \mathbb{Z}.$$

The function $p(x)$ represents the *probability distribution* of the random variable X. In this case, since X measures name lengths, X is a discrete random variable, and its probability distribution may be represented by a *Probability Mass Function* (*PMF*), such as $p(x)$.

To illustrate this, we carry out a simulation of many such random experiments, yielding many

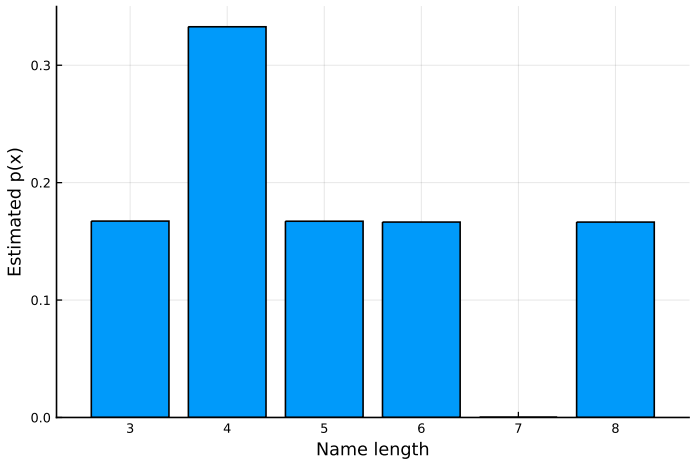

Figure 3.1: A discrete probability distribution taking values on $\{3, 4, 5, 6, 8\}$.

replications of the random variable X, which we then use to estimate $p(x)$. This is performed in Listing 3.1 below.

Listing 3.1: A simple random variable

```
1  using StatsBase, Plots; pyplot()
2
3  names = ["Mary","Mel","David","John","Kayley","Anderson"]
4  randomName() = rand(names)
5  X = 3:8
6  N = 10^6
7  sampleLengths = [length(randomName()) for _ in 1:N]
8
9  bar(X,counts(sampleLengths)/N, ylims=(0,0.35),
10     xlabel="Name length", ylabel="Estimated p(x)", legend=:none)
```

In line 3 we create the array `names`, which contains names with different character lengths. Note that two names have four characters, namely "Mary" and "John", while there is no name with 7 characters. In line 4, we define the function `randomName()` which randomly selects, with equal probability, an element from the array `names`. In line 5, we specify that we will count names of 3 to 8 characters in length. Line 6 specifies how many random experiments of choosing a name we will perform. Line 7 uses a comprehension and the function `length()` to count the length of each random name, and stores the results in the array `sampleLengths`. Here the Julia function `length()` is the analog of the random variable. That is, it is a function of the sample space, Ω, yielding a numerical value. Line 9 uses the function `counts()` to count how many words are of length 3, 4, up to 8. The `bar()` function is then used to plot a bar-chart of the proportion of counts for each word length. Two key observations can be made. It can be seen that words of length 4 occurred twice as much as words of lengths 3, 5, 6 and 8. In addition, no words of length 7 were selected, as no name in our original array had a length of 7.

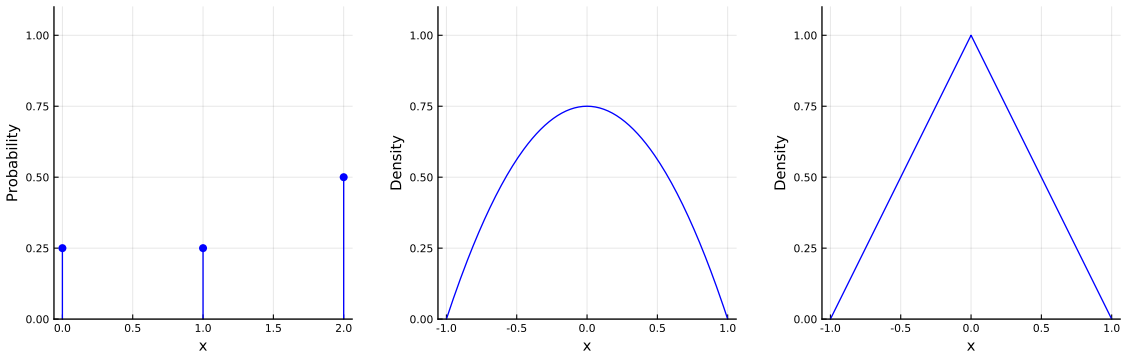

Figure 3.2: Three different examples of probability distributions.

Types of Random Variables

In the previous example, the random variable X took on discrete values and is thus called a *discrete random variable*. However, quantities measured in nature are often continuous, in which case a *continuous random variable* better describes the situation. For example, the weights of people randomly selected from a big population.

In describing the probability distribution of a continuous random variable, the probability mass function, $p(x)$, is no longer applicable. This is because for a continuous random variable X, $\mathbb{P}(X = x)$ for any particular value of x is 0. Hence in this case, the *Probability Density Function* (*PDF*), $f(x)$ is used, where,

$$f(x)\Delta \;\approx\; \mathbb{P}\big(x \leq X \leq x + \Delta\big).$$

Here the approximation becomes exact as $\Delta \to 0$. Figure 3.2, illustrates three examples of probability distributions. The one on the left is discrete and the other two are continuous.

The discrete probability distribution appearing on the left in Figure 3.2 can be represented mathematically by the probability mass function

$$p(x) = \begin{cases} 0.25 & \text{for } x = 0, \\ 0.25 & \text{for } x = 1, \\ 0.5 & \text{for } x = 2. \end{cases} \tag{3.1}$$

The smooth continuous probability distribution is defined by the probability density function,

$$f_1(x) = \frac{3}{4}(1 - x^2) \qquad \text{for } -1 \leqslant x \leqslant 1.$$

Finally, the triangular probability distribution is defined by the probability density function,

$$f_2(x) = \begin{cases} x + 1 & \text{for } x \in [-1, 0], \\ 1 - x & \text{for } x \in (0, 1]. \end{cases}$$

Note that for both the probability mass function and the probability density function, it is implicitly assumed that $p(x)$ and $f(x)$ are zero for x values not specified in the equation.

It can be verified that for the discrete distribution,

$$\sum_x p(x) = 1,$$

and for the continuous distributions,

$$\int_{-\infty}^{\infty} f_i(x)\,dx = 1 \qquad \text{for} \qquad i = 1, 2.$$

There are additional descriptors of probability distributions other than the PMF and PDF, and these are further discussed in Section 3.3. Note that Figure 3.2 was generated by Listing 3.2 below.

Listing 3.2: Plotting discrete and continuous distributions

```
1  using Plots, Measures; pyplot()
2
3  pDiscrete = [0.25, 0.25, 0.5]
4  xGridD = 0:2
5
6  pContinuous(x) = 3/4*(1 - x^2)
7  xGridC = -1:0.01:1
8
9  pContinuous2(x) = x < 0 ? x+1 : 1-x
10
11 p1 = plot(xGridD, line=:stem, pDiscrete, marker=:circle, c=:blue, ms=6, msw=0)
12 p2 = plot(xGridC, pContinuous.(xGridC), c=:blue)
13 p3 = plot(xGridC, pContinuous2.(xGridC), c=:blue)
14
15 plot(p1, p2, p3, layout=(1,3), legend=false, ylims=(0,1.1), xlabel="x",
16         ylabel=["Probability" "Density" "Density"], size=(1200, 400), margin=5mm)
```

In line 3 we define an array specifying the PMF of our discrete distribution, and in lines 6 and 9 we define functions specifying the PDFs of our continuous distributions. In lines 11-16 we create plots of each of our distributions. Note that in the discrete case we use the `line=:stem` argument together with `marker=:circle`.

3.2 Moment Based Descriptors

The probability distribution of a random variable fully describes the probabilities of the events, $\{\omega \in \Omega \ : \ X(\omega) \in A\}$, for all sensible $A \subset \mathbb{R}$. However, it is often useful to describe the nature of a random variable via a single number or a few numbers. The most common example of this is the *mean* which describes the center of mass of the probability distribution. Other examples include the *variance* and *moments* of the probability distribution. We expand on these now.

Mean

The mean, also known as the *expected value* of a random variable X, is a measure of the central tendency of the distribution of X. It is represented by $\mathbb{E}[X]$, and is the value we expect to obtain

"on average" if we continue to take observations of X and average out the results. The mean of a discrete distribution with PMF $p(x)$ is

$$\mathbb{E}[X] = \sum_x x\, p(x).$$

In the example of the discrete distribution given by (3.1) it is,

$$\mathbb{E}[X] = 0 \times 0.25 \ + \ 1 \times 0.25 \ + \ 2 \times 0.5 = 1.25.$$

The mean of a continuous random variable, with PDF $f(x)$ is

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x \ f(x) \ dx,$$

which in the examples of $f_1(\cdot)$ and $f_2(\cdot)$ from Section 3.1 yield,

$$\int_{-1}^{1} x \ \frac{3}{4}(1-x^2) = 0,$$

and,

$$\int_{-1}^{0} x+1 \ \ dx \ + \ \int_{0}^{1} 1-x \ \ dx = 0,$$

respectively. As can be seen, both continuous distributions have the same mean even though their shapes are different. For illustration purposes, we now carry out this integration numerically in Listing 3.3.

Listing 3.3: Expectation via numerical integration

```
1  using QuadGK
2
3  sup = (-1,1)
4  f1(x) = 3/4*(1-x^2)
5  f2(x) = x < 0 ? x+1 : 1-x
6
7  expect(f,support) = quadgk((x) -> x*f(x),support...)[1]
8
9  println("Mean 1: ", expect(f1,sup))
10 println("Mean 2: ", expect(f2,sup))
```

```
Mean 1: 0.0
Mean 2: -2.0816681711721685e-17
```

In line 1 we specify usage of the `QuadGK` package, which contains functions that support one-dimensional numerical integration via a method called *adaptive Gauss-Kronrod quadrature*. In lines 4 and 5 we define the PDF's of the distributions via `f1()` and `f2()`. In line 7 we define the function `expect()` which takes two arguments, a function to integrate `f`, and a domain over which to integrate the function `support`. It uses the `quadgk()` function to evaluate the 1-dimensional integral given above. For this an anonymous function `(x) -> x*f(x)` is created. Note that the start and end points of the integral are `support[1]` and `support[2]` respectively. These are "splatted" into the second and third argument of `quadgk()` via the '`...`' operator. Note also that the function `quadgk()` returns two arguments, the evaluated integral and an estimated upper bound on the absolute error. Hence `[1]` is included at the end of the function, so that only the integral is returned. Lines 9-10 then evaluate the numerical integrals of the functions `f1` and `f2` over the interval `sup` and display the output. As can be seen, both integrals are effectively evaluated to zero.

General Expectation and Moments

In general, for a function $h : \mathbb{R} \to \mathbb{R}$ and a random variable X, we can consider the random variable $Y := h(X)$. The distribution of Y will typically be different from the distribution of X. As for the mean of Y, we have,

$$\mathbb{E}[Y] = \mathbb{E}[h(X)] = \begin{cases} \sum_x h(x)\, p(x) & \text{for discrete,} \\ \int_{-\infty}^{\infty} h(x)\, f(x)\, dx & \text{for continuous.} \end{cases} \tag{3.2}$$

Note that the above expression does not require explicit knowledge of the distribution of Y but rather uses the distribution (PMF or PDF) of X.

A common case is $h(x) = x^\ell$, in which case we call $\mathbb{E}[X^\ell]$, the ℓ'th moment of X. Then, for a random variable X with PDF $f(x)$, the ℓ^{th} *moment* of X is,

$$\mathbb{E}[X^\ell] = \int_{-\infty}^{\infty} x^\ell \; f(x) \; dx.$$

Note that the first moment is the mean and the zero'th moment is always 1. The second moment, is related to the variance as we explain below.

Variance

The *variance* of a random variable X, often denoted $\mathrm{Var}(X)$ or σ^2, is a measure of the spread, or *dispersion*, of the distribution of X. It is defined by,

$$\mathrm{Var}(X) := \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - \Big(\mathbb{E}[X]\Big)^2. \tag{3.3}$$

Here we apply (3.2) by considering $h(x) = (x - \mathbb{E}[X])^2$. The second expression of (3.3) illustrates the role of the first and second moments in the variance. It follows from the first expression by expansion.

For the discrete distribution, (3.1), we have:

$$\mathrm{Var}(X) = (0 - 1.25)^2 \times 0.25 \; + \; (1 - 1.25)^2 \times 0.25 \; + \; (2 - 1.25)^2 \times 0.5 = 0.6875.$$

For the continuous distributions from Section 3.1, $f_1(\cdot)$ and $f_2(\cdot)$, with respective random variables X_1 and X_2, we have

$$\mathrm{Var}(X_1) = \int_{-1}^{1} x^2 \; \frac{3}{4}(1 - x^2) \; dx - \big(\mathbb{E}[X_1]\big)^2 = \frac{3}{4}\left[\frac{x^3}{3} - \frac{x^5}{5}\right]_{-1}^{1} \; - \; 0 \; = \; 0.2,$$
$$\mathrm{Var}(X_2) = \int_{-1}^{0} x^2(x+1) \; dx + \int_{0}^{1} x^2(1-x) \; dx - \big(\mathbb{E}[X_2]\big)^2 \; = \; \frac{1}{6}.$$

The variance of X can also be considered as the expectation of a new random variable, $Y := (X - \mathbb{E}[X])^2$. However, when considering variance, the distribution of Y is seldom mentioned.

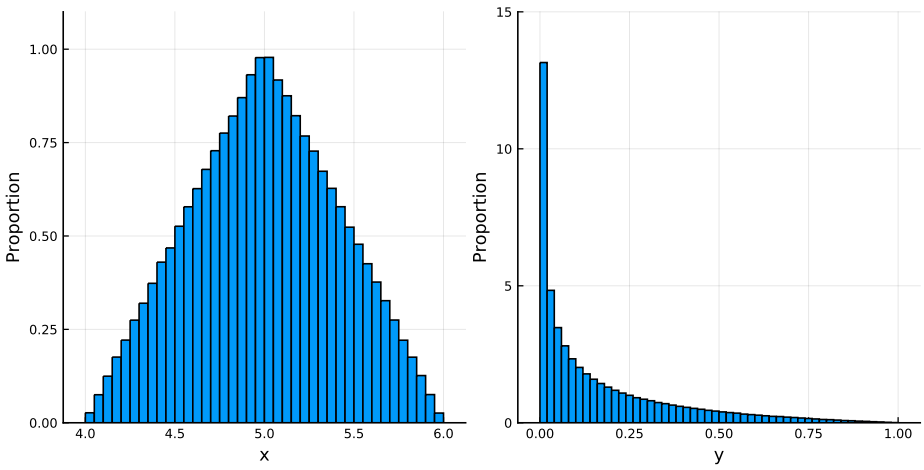

Figure 3.3: Histograms for samples of the random variables X and Y.

Nevertheless, as an exercise we explore this now. Consider a random variable X, with density,

$$f(x) = \begin{cases} x - 4 & \text{for } x \in [4, 5], \\ 6 - x & \text{for } x \in (5, 6]. \end{cases}$$

This density is similar to $f_2(\cdot)$ previously covered, but with support $[4, 6]$. In Listing 3.4, we generate random observations from X, and calculate data-points for Y based on these observations. We then plot both the distribution of X and Y, and show that the sample mean of Y is the sample variance of X. Note that our code uses some elements from the `Distributions` package, which is covered in more detail in Section 3.4.

Listing 3.4: Variance of X as the mean of Y

```
using Distributions, Plots; pyplot()

dist = TriangularDist(4,6,5)
N = 10^6
data = rand(dist,N)
yData=(data .- 5).^2

println("Mean: ", mean(yData), " Variance: ", var(data))

p1 = histogram(data, xlabel="x", bins=80, normed=true, ylims=(0,1.1))
p2 = histogram(yData, xlabel="y", bins=80, normed=true, ylims=(0,15))
plot(p1, p2, ylabel="Proportion", size=(800, 400), legend=:none)
```

```
Mean(Y) = 0.16671191478072614   Variance(X) = 0.1667120530661165
```

Line 1 calls the `Distributions` package. This package supports a variety of distribution types through the many functions it contains. We expand further on the use of the `Distributions` package in Section 3.4. Line 2 uses the `Triangular()` function from the `Distributions` package to create a triangular distribution type object with a mean of 5 and a symmetric shape over the bound $[4, 6]$. We assign this as the variable `dist`. In line 5 we generate an array of `N` observations from the distribution by applying the `rand()` function on the distribution `dist`. Line 6 takes the observations in `data` and from them generates observations for the new random variable Y. The values are stored in the array `yData`. Line 8 uses the functions `mean()` and `var()` on the arrays `yData` and `data` respectively. It can be seen from the output that the mean of the distribution Y is the same as the variance of X. Lines 10-12 are used to plot histograms of the data in the arrays `data` and `yData`. It can be observed that the histogram on the left approximates the PDF of our triangular distribution, while the histogram on the right approximates the distribution of the new variable Y. The distribution of Y is seldom considered when evaluating the variance of X.

Higher Order Descriptors: Skewness and Kurtosis

As described previously, the second moment plays a role defining the dispersion of a distribution via the variance. What about higher order moments? We now briefly define the skewness and kurtosis of a distribution utilizing the first three moments and first four moments respectively.

Take a random variable X with $\mathbb{E}[X] = \mu$ and $\text{Var}(X) = \sigma^2$, then the *skewness*, is defined as,

$$\gamma_3 = \mathbb{E}\Big[\Big(\frac{X-\mu}{\sigma}\Big)^3\Big] = \frac{\mathbb{E}[X^3] - 3\mu\sigma^2 - \mu^3}{\sigma^3},$$

and the *kurtosis* is defined as,

$$\gamma_4 = \mathbb{E}\Big[\Big(\frac{X-\mu}{\sigma}\Big)^4\Big] = \frac{\mathbb{E}[(X-\mu)^4]}{\sigma^4}.$$

Note that, γ_3 and γ_4 are invariant to changes in location and scale of the distribution.

The skewness is a measure of the asymmetry of the distribution. For a distribution having a symmetric density function about the mean, we have $\gamma_3 = 0$. Otherwise, it is either positive or negative depending on the distribution being *skewed to the right* or *skewed to the left* respectively.

The kurtosis is a measure of the tails of the distribution. As a benchmark, any normal probability distribution (covered in detail in Section 3.6) has $\gamma_4 = 3$. Then, a probability distribution with a higher value of γ_4 can be interpreted as having 'heavier tails' (than a normal distribution), while a probability distribution with a lower value is said to have 'lighter tails' (than a normal distribution). This benchmark even yields a term called *excess kurtosis* defined as $\gamma_4 - 3$. Hence, a positive excess kurtosis implies 'heavy tails' and a negative value implies 'light tails'.

Laws of Large Numbers

Throughout this book, our Monte-Carlo experiments rely on *laws of large numbers*. This suite of mathematical statements claim that empirical averages converge to expected values. Stated as mathematical theorems, these laws come in different forms including the *weak law of large numbers*

and the *strong law of large numbers*. In both cases, a sequence of independent and identically distributed random variables, $X_1, X_2, \ldots$, is considered. Then for each n, we compute the sample mean,

$$\overline{X}_n = \frac{1}{n}\sum_{k=1}^{n} X_k,$$

and consider the sequence of sample means.

$$\overline{X}_1, \overline{X}_2, \ldots.$$

If the mean of each of the random variables X_i is μ, then a law of large numbers is a claim that the sequence $\{\overline{X}_n\}_{n=1}^{\infty}$ converge to μ. The distinction between "weak" and "strong" lies with the *mode of convergence*. For example, the weak law of large numbers claims that the sequence of probabilities,

$$w_n = \mathbb{P}(|\overline{X}_n - \mu| > \epsilon),$$

converges to 0 for any positive ϵ. That is, as n grows, the likelihood of the sample mean $\overline{X}_n$ to be farther away than ϵ from the mean μ vanishes. This is a statement about the sequence of probabilities, $w_1, w_2, \ldots$. In contrast, the strong law of large numbers states that,

$$\mathbb{P}\Big(\lim_{n\to\infty} \overline{X}_n = \mu\Big) = 1. \tag{3.4}$$

This means that with certainty, every sequence of sample means converges to the expectation. From a practical perspective the implication is similar to the weak law of large numbers, however, mathematically the statement is different. In fact, the strong law of large numbers condition (3.4) implies the weak law of large numbers.

It turns out that proving the weak law of large numbers is much easier than proving the strong law of large numbers. Also, for the strong law of large numbers, if we are willing to assume that $\mathbb{E}[X_i^4] < \infty$ then a proof isn't too difficult, however the minimal conditions are that $\mathbb{E}[X_i]$ is finite, and under these conditions a proof is more involved. See [Ros06] for an introduction to such aspects of *rigorous probability theory*, including proofs. Also related is the example presented later in Listing 3.30. It deals with the Cauchy distribution and illustrates a scenario where the law of large numbers breaks because $\mathbb{E}[X_i]$ does not exist.

Keep in mind that in many cases, we convert the sequence $X_1, X_2, \ldots$ into the sequence $I_1, I_2, \ldots$ via,

$$I_i = \begin{cases} 1 & \text{if } X_i \text{ satisfies some condition,} \\ 0 & \text{if } X_i \text{ does not satisfy the condition.} \end{cases}$$

In such a case,

$$\mathbb{E}[I_i] = \mathbb{P}(X_i \text{ satisfies the condition}),$$

and the average,

$$\overline{I}_n = \frac{1}{n}\sum_{i=1}^{n} I_i,$$

is the proportion of samples over $1, \ldots, n$ that satisfy the condition. Here strong laws of large numbers (weak or strong) imply that empirical proportions converge to probabilities.

3.3 Functions Describing Distributions

As alluded to in Section 3.2, a probability distribution can be described by a probability mass function (PMF) in the discrete case, or a probability density function (PDF) in the continuous case. However, there are other popular descriptors of probability distributions, such as the *cumulative distribution function* (CDF), the *complementary cumulative distribution function* (CCDF), and the *inverse cumulative distribution function*. There are also transform-based descriptors including the *moment generating function* (MGF), *probability generating function* (PGF), as well as related functions such as the *characteristic function* (CF), or alternative names, including the *Laplace transform*, *Fourier transform* or *z transform*. Then, for non-negative random variables there is also the *hazard function* which we explore along with the Weibull distribution in Section 3.6. The main point to take away here is that a probability distribution can be described in many alternative ways. We now explore a few of these descriptors.

Cumulative Probabilities

Consider first the CDF of a random variable X, defined as,

$$F(x) := \mathbb{P}(X \leq x),$$

where X can be discrete, continuous or a more general random variable. The CDF is a very popular descriptor because unlike the PMF or PDF, it is not restricted to just the discrete or just the continuous case. A closely related function is the CCDF, $\bar{F}(x) := 1 - F(x) = \mathbb{P}(X > x)$.

From the definition of the CDF, $F(\cdot)$,

$$\lim_{x \to -\infty} F(x) = 0 \qquad \text{and} \qquad \lim_{x \to \infty} F(x) = 1.$$

Furthermore, $F(\cdot)$ is a non-decreasing function. In fact, any function with these properties constitutes a valid CDF and hence a probability distribution of a random variable.

In the case of a continuous random variable, the PDF $f(\cdot)$, and the CDF $F(\cdot)$, are related via,

$$f(x) = \frac{d}{dx} F(x) \qquad \text{and} \qquad F(x) = \int_{-\infty}^{x} f(u)\, du.$$

Also, as a consequence of the CDF properties,

$$f(x) \geq 0, \qquad \text{and} \qquad \int_{-\infty}^{\infty} f(x)\, dx = 1. \tag{3.5}$$

Analogously, while less appealing than the continuous counter-part, in the case of discrete random variable, the PMF $p(\cdot)$ is related to the CDF via,

$$p(x) = F(x) - \lim_{t \to x^-} F(t) \qquad \text{and} \qquad F(x) = \sum_{k \leq x} p(k). \tag{3.6}$$

Note that here we consider $p(x)$ to be 0 for x not in the support of the random variable. The important point in presenting (3.5) and (3.6) is to show that $F(\cdot)$ is a valid description of the probability distribution.

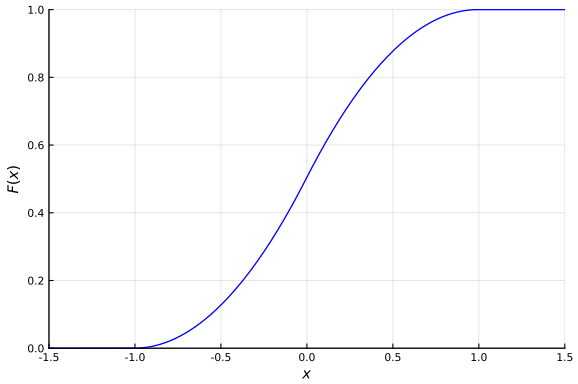

Figure 3.4: The CDF associated with the PDF $f_2(x)$.

In Listing 3.5, we look at an elementary example, where we consider the PDF $f_2(\cdot)$ of Section 3.1 and integrate it via a crude *Riemann sum* to obtain the CDF:

$$F(x) = \mathbb{P}(X \leq x) = \int_{-\infty}^{x} f_2(u)\ du \approx \sum_{u=-\infty}^{x} f_2(u)\,\Delta u. \tag{3.7}$$

Listing 3.5: CDF from the Riemann sum of a PDF

```
1   using Plots, LaTeXStrings; pyplot()
2
3   f2(x) = (x<0 ? x+1 : 1-x)*(abs(x)<1 ? 1 : 0)
4   a, b = -1.5, 1.5
5   delta = 0.01
6
7   F(x) = sum([f2(u)*delta for u in a:delta:x])
8
9   xGrid = a:delta:b
10  y = [F(u) for u in xGrid]
11  plot(xGrid, y, c=:blue, xlims=(a,b), ylims=(0,1),
12          xlabel=L"x", ylabel=L"F(x)", legend=:none)
```

In line 3 we define the function `f2()`. The second set of brackets in the equation are used to ensure that the PDF is zero outside of the region $[-1, 1]$, as it acts like an*indicator function*, and evaluates to 0 everywhere else. In line 4 and 5 we set the limits of our integral, and the stepwise `delta` used. In line 7 we create a function that approximates the value of the CDF through a crude Riemann sum by evaluating the PDF at each point `u`, multiplying this by `delta`, and repeating this process for each progressively larger interval up to the specified value `x`. The total area is then approximated via the `sum()` function. See (3.7). In line 9 we specify the grid of values over which we will plot our approximated CDF. Line 10 uses the function `F()` to create the array `y`, which contains the actual approximation of the CDF over the grid of value specified. Lines 11-12 plot Figure 3.4.

Inverse and Quantiles

Where the CDF answers the question "what is the probability of being less than or equal to x", a dual question often asked is "what value of x corresponds to a probability of the random variable being less than or equal to u". Mathematically, we are looking for the *inverse function* of $F(x)$. In cases where the CDF is continuous and strictly increasing over all values, the inverse, $F^{-1}(\cdot)$ is well defined, and can be found via the equation,

$$F\big(F^{-1}(u)\big) = u, \qquad \text{for } u \in [0,1]. \tag{3.8}$$

For example, take the *sigmoid function* as the CDF, which is as a type of *logistic function*,

$$F(x) = \frac{1}{1+e^{-x}}.$$

Solving for $F^{-1}(u)$ in (3.8) yields,

$$F^{-1}(u) = \log \frac{u}{1-u}.$$

Observe that as $u \to 0^+$ we get $F^{-1}(u) \to -\infty$ and as $u \to 1^-$ we get $F^{-1}(u) \to \infty$. This is the *inverse CDF* for the distribution. Schematically, given a specified probability u, it allows us to find x values such that,

$$\mathbb{P}(X \leq x) = u. \tag{3.9}$$

The value x satisfying (3.9) is also called the u'th *quantile* of the distribution. If u is given as a percent, then it is called a *percentile*. The *median* is another related term, and is also known as the 0.5'th quantile. Other related terms are the *quartiles*, with the *first quartile* at $u = 0.25$, the *third quartile* at $u = 0.75$ and the *inter-quartile range*, which is defined as $F^{-1}(0.75) - F^{-1}(0.25)$. These same terms used again in respect to summarizing datasets in Section 4.2.

In more general cases, where the CDF is not necessarily strictly increasing and continuous, we may still define the inverse CDF via,

$$F^{-1}(u) := \inf\{x \ : \ F(x) \geq u\}.$$

As an example of such a case, consider an arbitrary customer arriving to a queue where the server is utilized 80% of the time, and an average service takes 1 minute. How long does such a customer wait in the queue until service starts? Some customers won't wait at all (20% of the customers), whereas others will need to wait until those that arrived before them are serviced. Results from the field of *queueing theory* (some of which are partially touched in Chapter 10) give rise to the following distribution function for the waiting time:

$$F(x) = 1 - 0.8e^{-(1-0.8)x} \qquad \text{for} \quad x \geq 0. \tag{3.10}$$

Notice that at $x = 0$, $F(0) = 0.2$, indicating the fact that there is a 0.2 chance for zero wait. Such a distribution is an example of a *mixed discrete and continuous distribution*. Notice that this distribution function only holds for a specific case of assumptions known as the stationary stable M/M/1 queue, explored further in Section 10.3. We now plot both $F(x)$ and $F^{-1}(u)$ in Listing 3.6 where we construct $F^{-1}(\cdot)$ programmatically. Observe Figure 3.5 where the CDF $F(x)$ exhibits a jump at 0 indicating the "probability mass". The inverse CDF then evaluates to 0 for all values of $u \in [0, 0.2]$.

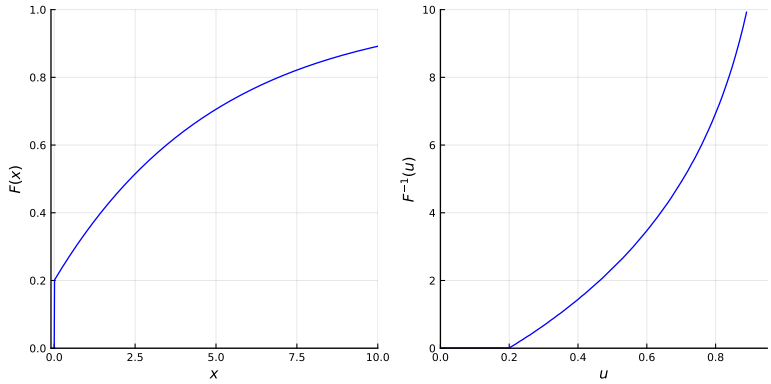

Figure 3.5: The CDF $F(x)$, and its inverse $F^{-1}(u)$.

Listing 3.6: The inverse CDF

```
1  using Plots, LaTeXStrings; pyplot()
2
3  xGrid = 0:0.01:10
4  uGrid = 0:0.01:1
5  busy = 0.8
6
7  F(t)= t<=0 ? 0 : 1 - busy*exp(-(1-busy)t)
8
9  infimum(B) = isempty(B) ? Inf : minimum(B)
10 invF(u) = infimum(filter((x) -> (F(x) >= u),xGrid))
11
12 p1 = plot(xGrid,F.(xGrid), c=:blue, xlims=(-0.1,10), ylims=(0,1),
13         xlabel=L"x", ylabel=L"F(x)")
14
15 p2 = plot(uGrid,invF.(uGrid), c=:blue, xlims=(0,0.95), ylims=(0,maximum(xGrid)),
16         xlabel=L"u", ylabel=L"F^{-1}(u)")
17
18 plot(p1, p2, legend=:none, size=(800, 400))
```

Line 3 defines the grid over which we will evaluate the CDF. Line 4 defines the grid over which we will evaluate the inverse CDF. In line 5 we define the time proportion during which the server is busy. In line 7 we define the function `F()` as in (3.10). Note that for values less than zero, the CDF evaluates to 0. In line 9 we define the function `infimum()`, which implements similar logic to the mathematical operation $\inf\{\}$. It takes an input and checks if it is empty via the `isempty()` function, and if it is, returns `Inf`, else returns the minimum value of the input. This agrees with the typical mathematical notation where the infimum of the empty set is ∞. In line 10 we define the function `invF()`. It first creates an array (representing a set) $\{x : F(x) \geq u\}$ directly via the Julia `filter()` function. Note that as a first argument, we use an anonymous Julia function, `(x) -> (F(x) >= u)`. We then use this function as a filter over `xGrid`. Finally we apply the infimum over this mathematical set (represented by a vector of coordinates on the x axis). Lines 12-18 are used to plot both the original CDF via the `F()` function, and the inverse CDF via the `invF()` functions respectively.

Integral Transforms

In general terms, an *integral transform* of a probability distribution is a representation of the distribution on a different domain. Here we focus on the moment generating function (MGF). Other examples include the characteristic function (CF), probability generating function (PGF) and similar transforms.

For a random variable X and a real or complex fixed value s, consider the expectation, $\mathbb{E}[e^{sX}]$. When viewed as a function of s, this is the moment generating function. We present this here for such a continuous random variable with PDF $f(\cdot)$:

$$M(s) = \mathbb{E}[e^{sX}] = \int_{-\infty}^{\infty} f(x)\, e^{sx}\, dx. \tag{3.11}$$

This is also known as the bi-lateral *Laplace transform* of the PDF (with argument $-s$). Many useful Laplace transform properties carry over from the theory of Laplace transforms to the MGF. A full exposition of such properties are beyond the scope of this book, however we illustrate a few via an example.

Consider two distributions with densities,

$$\begin{aligned} f_1(x) &= 2x \qquad \text{for} \qquad x \in [0,1], \\ f_2(x) &= 2-2x \qquad \text{for} \qquad x \in [0,1], \end{aligned}$$

where the respective random variables are denoted X_1 and X_2. Computing the MGF of these distributions we obtain,

$$M_1(s) = \int_0^1 2x\, e^{sx}\, dx = 2\frac{1+e^s(s-1)}{s^2},$$

$$M_2(s) = \int_0^1 (2-2x)\, e^{sx}\, dx = 2\frac{e^s-1-s}{s^2}.$$

Define now a random variable, $Z = X_1 + X_2$ where X_1 and X_2 are assumed independent. In this case, it is known that the MGF of Z is the product of the MGFs of X_1 and X_2. That is,

$$M_Z(s) = M_1(s)M_2(s) = 4\frac{\big(1+e^s(s-1)\big)\big(e^s-1-s\big)}{s^4}. \tag{3.12}$$

The new MGF $M_Z(\cdot)$ fully specifies the distribution of Z. It also yields a rather straightforward computation of moments, hence the name MGF. A key property of any MGF $M(s)$ of a random variable X is that,

$$\frac{d^n}{ds^n} M(s)\Big|_{s=0} = \mathbb{E}[X^n]. \tag{3.13}$$

This can be easily verified from (3.11). Hence to calculate the n'th moment, one can simply evaluate the derivative of the MGF at $s = 0$. Note that in certain cases, evaluating the limit of $s \to 0$ is required.

In Listing 3.7, we estimate both the PDF and MGF of Z and compare the estimated MGF to $M_Z(s)$ above. The listing also creates Figure 3.6 where on the right hand side plot it can be seen that the slope of the tangent line to the MGF at $s = 0$ is 1.0, in agreement with the mean.

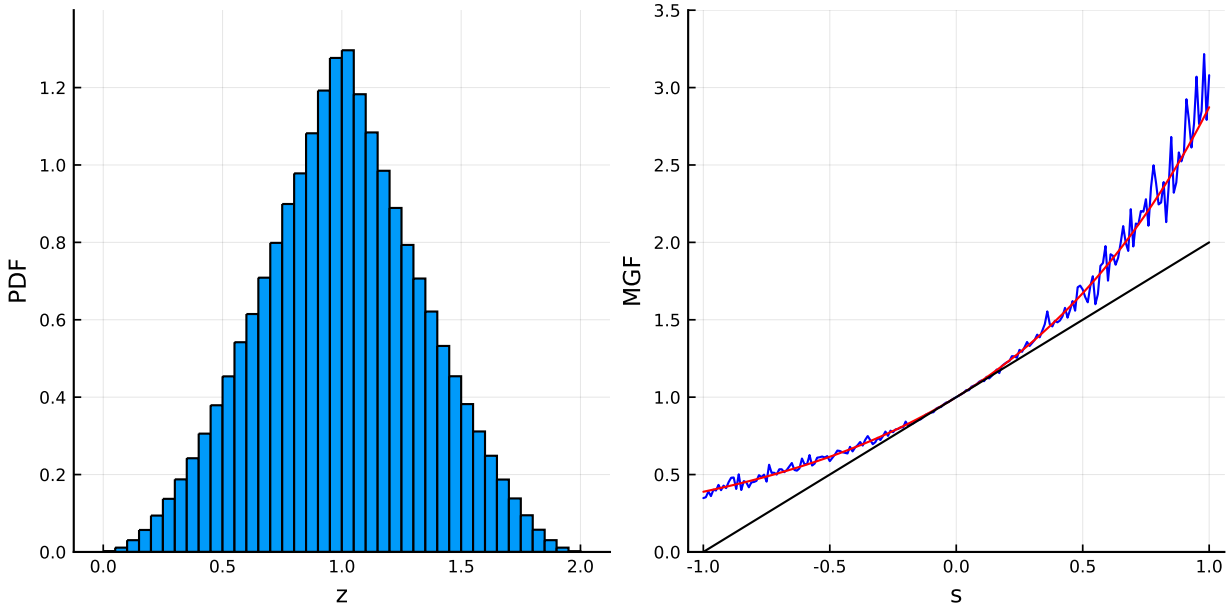

Figure 3.6: Left: The estimate of the PDF of Z via a histogram.
Right: The theoretical MGF in red vs. a Monte Carlo estimate in blue.
The slope of the black line is the mean.

Listing 3.7: A sum of two triangular random variables

```
using Distributions, Statistics, Plots; pyplot()

dist1 = TriangularDist(0,1,1)
dist2 = TriangularDist(0,1,0)
N=10^6

data1, data2 = rand(dist1,N), rand(dist2,N)
dataSum = data1 + data2

mgf(s) = 4(1+(s-1)*MathConstants.e^s)*(MathConstants.e^s-1-s)/s^4

mgfPointEst(s) = mean([MathConstants.e^(s*z) for z in
                                        rand(dist1,20) + rand(dist2,20)])

p1 = histogram(dataSum, bins=80, normed=:true,
        ylims=(0,1.4), xlabel="z", ylabel="PDF")

sGrid = -1:0.01:1
p2 = plot(sGrid, mgfPointEst.(sGrid), c=:blue, ylims=(0,3.5))
p2 = plot!(sGrid, mgf.(sGrid), c=:red)
p2 = plot!( [minimum(sGrid),maximum(sGrid)],
        [minimum(sGrid),maximum(sGrid)].+1,
        c=:black, xlabel="s", ylabel="MGF")

plot(p1, p2, legend=:none, size=(800, 400))
```

In lines 3 and 4 we create two separate triangular distribution type objects `dist1` and `dist2`, matching the densities $f_1(x)$ and $f_2(x)$ respectively. Note that the third argument of the `TriangularDist()` function is the location of the "peak" of the triangle (or the mode of the distribution). Distribution objects are covered further in Section 3.4 below. In line 7 we generate random observations from `dist1` and `dist2`, and store these observations separately in the two arrays `data1` and `data2` respectively. In line 8 we generate observations for Z by performing element-wise summation of the values in our arrays `data1` and `data2`. In line 10 we implement the MGF function as in (3.12). In lines 12-13 we define the function `mgfPointEst()`, which crudely estimates the MGF at the point `s`. We purposefully only use 20 observations, each time estimating the sample mean of e^{sZ} for a specified s. The remainder of the code uses the data and the defined functions to generate Figure 3.6. Lines 21-23 plot the black line.

3.4 `Distributions` and Related Packages

As touched on previously in Listing 3.4 and Listing 3.7, Julia has a well developed package for distributions. The `Distributions` package allows us to create distribution type objects based on what family they belong to (more on families of distributions in Sections 3.5 and 3.6). These distribution objects can then be used as arguments for other functions, for example `mean()` and `var()`. Of key importance is the ability to randomly sample from a distribution using `rand()`. We can also use distributions with other functions including `pdf()`, `cdf()`, and `quantile()` to name a few. In addition, the built-in `Statistics` package as well as the `StatsBase` package contain many functions which have methods for distribution type objects. A useful paper describing the distributions package is [BAABLPP19].

Weighted Vectors

In the case of discrete distributions of finite support, the `StatsBase` package provides the "weight vector" object via `Weights()`, which allows for an array of values, or outcomes, to be given probabilistic weights. This is also known as a *probability vector*. In order to generate observations we use the `sample()` function (from `StatsBase`) on a vector given its weights, instead of the `rand()` function. Note that an alternative is to use the `Categorical` distribution supplied via the `Distributions` package. Listing 3.8 below provides a brief example of the use of weight vectors.

Listing 3.8: Sampling from a weight vector

```
1  using StatsBase, Random
2  Random.seed!(1)
3
4  grade = ["A","B","C","D","E"]
5  weightVect = Weights([0.1,0.2,0.1,0.2,0.4])
6
7  N = 10^6
8  data = sample(grade,weightVect,N)
9  [count(i->(i==g),data) for g in grade]/N
```

```
5-element Array{Float64,1}:
 0.099901
```

```
0.200248
0.099704
0.20068
0.399467
```

In line 4 we define an array of strings "A" to "E", which represent possible outcomes. In line 5 we define their weights. Note the fact that `Weights()` is capitalized, signifying the fact that the function creates a new object. This type of function is known as a *Constructor*. Line 8 uses the function `sample()` to sample `N` observations from our array `grade`, according to the weights given by the weight vector `weightVect`. Line 9 uses the `count()` function to count how many times each entry `g` in `grade` occurs in `data`, and then evaluates the proportion of times total each grade occurs. It can be observed that the grades have been sampled according to the probabilities specified in the array `weightVect`. Note that you can also use the `Categorical()` object in the `Distributions` package as alternative.

Using Distribution Type Objects

We now introduce some important functionality of the `Distributions` package and distribution type objects through an example. Consider a distribution from the "Triangular" family, with the following density,

$$f(x) = \begin{cases} x & \text{for } x \in [0,1], \\ 2-x & \text{for } x \in (1,2]. \end{cases}$$

In Listing 3.9, rather than creating the density manually as in the previous sections, we use the `TriangularDist()` constructor to create a distribution type object, and then use this to create plots of the PDF, CDF and inverse CDF as in Figure 3.7.

Listing 3.9: Using the `pdf()`, `cdf()`, and `quantile()` functions with `Distributions`

```
1  using Distributions, Plots, LaTeXStrings; pyplot()
2
3  dist = TriangularDist(0,2,1)
4  xGrid = 0:0.01:2
5  uGrid = 0:0.01:1
6
7  p1 = plot( xGrid, pdf.(dist,xGrid), c=:blue,
8                  xlims=(0,2), ylims=(0,1.1),
9                  xlabel="x", ylabel="f(x)")
10 p2 = plot( xGrid, cdf.(dist,xGrid), c=:blue,
11                 xlims=(0,2), ylims=(0,1),
12                 xlabel="x", ylabel="F(x)")
13 p3 = plot( uGrid,quantile.(dist,uGrid), c=:blue,
14                 xlims=(0,1), ylims=(0,2),
15                 xlabel="u", ylabel=(L"F^{-1}(u)"))
16
17 plot(p1, p2, p3, legend=false, layout=(1,3), size=(1200, 400))
```

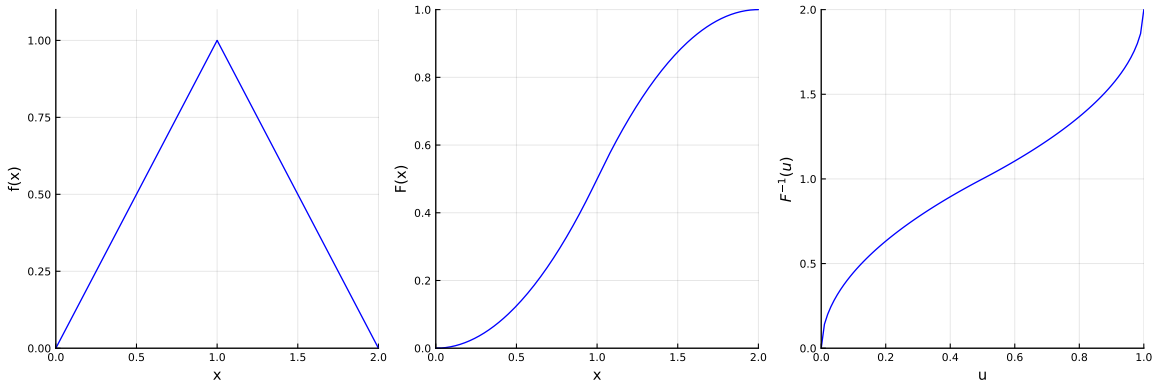

Figure 3.7: The PDF, CDF and inverse CDF a triangular distribution.

In line 3 we use the `TriangularDist()` function to create a distribution type object. The first two arguments are the start and end points of the support, and the third argument is the location of the "peak" (or mode). The essence of this example is in lines 7, 10 and 13 where we use the `pdf()`, `cdf()` and `quantile()` functions respectively. In each case we use `dist` as the first argument and broadcast over the second argument via the '`.`' broadcast operator.

In addition to evaluating functions associated with the distribution, we can also query a distribution object for a variety of properties and parameters. Given a distribution object, you may apply `params()` on it to retrieve the distributional parameters. You may query for the `mean()`, `median()`, `var()` (variance), `std`, (standard deviation), `skewness()`, and `kurtosis()`. You can also query for the minimal and maximal value in the support of the distribution via `minimum()` and `maximum()` respectively. You may also apply `mode()` or `modes()` to either get a single mode (value of x where the PMF or PDF is maximized) or an array of modes where applicable. Listing 3.10 illustrates some of these for our `TriangularDist`.

Listing 3.10: Descriptors of `Distribution` objects

```
1  using Distributions
2  dist = TriangularDist(0,2,1)
3
4  println("Parameters: \t\t\t",params(dist))
5  println("Central descriptors: \t\t",mean(dist),"\t",median(dist))
6  println("Dispersion descriptors: \t", var(dist),"\t",std(dist))
7  println("Higher moment shape descriptors: ",skewness(dist),"\t",kurtosis(dist))
8  println("Range: \t\t\t\t", minimum(dist),"\t",maximum(dist))
9  println("Mode: \t\t\t\t", mode(dist), "\tModes: ",modes(dist))
```

```
Parameters:                     (0.0, 2.0, 1.0)
Central descriptors:            1.0     1.0
Dispersion descriptors:         0.16666666666666666     0.408248290463863
Higher moment shape descriptors: 0.0    -0.6
Range:                          0.0     2.0
Mode:                           1.0     Modes: [1.0]
```

In Listing 3.11 we look at another example, where we generate random observations from a distribution type object via the `rand()` function, and compare the sample mean against the specified mean. Note that two different types of distributions are created here, a continuous distribution and a discrete distribution. These are discussed further in Sections 3.5 and 3.6 respectively.

Listing 3.11: Using `rand()` with `Distributions`

```
1  using Distributions, StatsBase, Random
2  Random.seed!(1)
3
4  dist1 = TriangularDist(0,10,5)
5  dist2 = DiscreteUniform(1,5)
6  theorMean1, theorMean2 = mean(dist1), mean(dist2)
7
8  N = 10^6
9  data1 = rand(dist1,N)
10 data2 = rand(dist2,N)
11 estMean1, estMean2 = mean(data1), mean(data2)
12
13 println("Symmetric Triangular Distiribution on [0,10] has mean $theorMean1
14         (estimated: $estMean1)")
15 println("Discrete Uniform Distiribution on {1,2,3,4,5} has mean $theorMean2
16         (estimated: $estMean2)")
```

```
Symmetric Triangular Distiribution on [0,10] has mean 5.0
        (estimated: 4.999164797766807)
Discrete Uniform Distiribution on {1,2,3,4,5} has mean 3.0
        (estimated: 3.001862)
```

In line 4 we use the `TriangularDist()` function to create a symmetrical triangular distribution about 5, and store this as `dist1`. In line 5 we use the `DiscreteUniform()` function to create a discrete uniform distribution, and store this as `dist2`. Note that observations from this distribution can take on values from $\{1, 2, 3, 4, 5\}$, each with equal probability. In line 6 we evaluate the mean of the two distribution objects created above by applying the function `mean()` to both of them. These methods of `mean()` only use the parameters of the distribution to evaluate the mean. No data manipulation is taking place. In lines 8-11 we estimate the means of the two distributions by randomly sampling from our distributions `dist1` and `dist2`. In lines 9-10 the `Distribution` object is given as a first argument to `rand()`. Lines 13-16 print the results. It can be seen that the estimated means are a good approximation of the actual means.

The Inverse Probability Transform

One may ask how does Julia (or any software package) generate random values from a given distribution? There are a variety of techniques for transforming pseudo-random numbers from a uniform distribution into numbers from a given distribution. An extensive treatment is in [KTB11]. One basic method which stands above the rest is *inverse transform sampling*.

Let X be a random variable distributed with CDF $F(\cdot)$ and inverse CDF $F^{-1}(\cdot)$. Now take U to be a uniform random variable over $[0, 1]$, and let $Y = F^{-1}(U)$. It holds that Y is distributed like X. This useful property is called the *inverse probability transform* and constitutes a generic method for generating random variables from an underlying distribution.

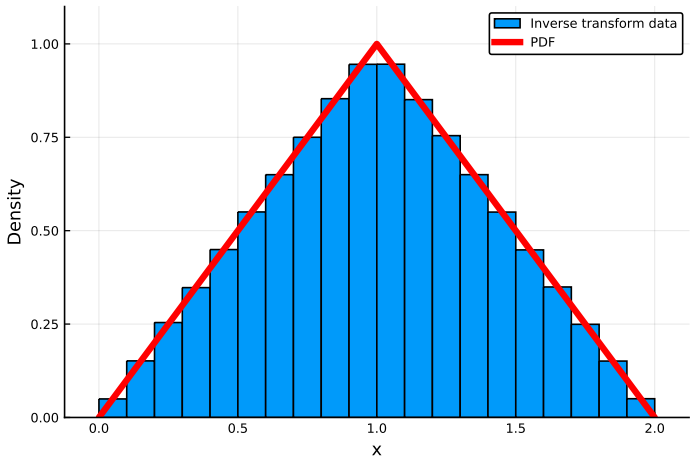

Figure 3.8: A histogram generated using the inverse probability transform compared to the PDF of a triangular distribution.

To see why the method works, consider a uniform random variable U and apply to it the inverse probability transform $F^{-1}(\cdot)$. In such a case, consider the CDF of $Y = F^{-1}(U)$ and see that it is $F(\cdot)$:

$$F_Y(y) = \mathbb{P}\big(Y \leq y\big) = \mathbb{P}\big(F^{-1}(U) \leq y\big) = \mathbb{P}\big(U \leq F(y)\big) = F_U\big(F(y)\big) = F(y).$$

The third equality follows because $F(\cdot)$ is a monotonic function and can be applied to both sides of the inequality. The last step follows because the CDF of uniform $(0, 1)$ random variable is,

$$F_U(z) = \begin{cases} 0 & \text{for } z < 0, \\ z & \text{for } 0 \leq z \leq 1, \\ 1 & \text{for } 1 < z. \end{cases}$$

Keep in mind, that when using the `Distributions` package, we would typically generate random variables using the `rand()` function on a distribution type object, as performed in Listing 3.11 above. The implementation of `rand()` may use the inverse probability transform or alternatively may use a different type of method depending on the distribution at hand. However, in Listing 3.12 below, we illustrate how to use the inverse probability transform with the results presented in Figure 3.8. Observe that we can implement $F^{-1}(\cdot)$ via the `quantile()` function.

Listing 3.12: Inverse transform sampling

```
using Distributions, Plots; pyplot()

triangDist = TriangularDist(0,2,1)
xGrid = 0:0.1:2
N = 10^6
inverseSampledData = quantile.(triangDist,rand(N))

histogram( inverseSampledData, bins=30, normed=true,
        ylims=(0,1.1), label="Inverse transform data")
plot!( xGrid, pdf.(triangDist,xGrid), c=:red, lw=4,
        xlabel="x", label="PDF", ylabel = "Density", legend=:topright)
```

In line 3 we create our triangular distribution `triangDist`. In lines 4 and 5 we define the support over which we plot our data, as well as how many data-points we simulate. In line 6 we generate `N` random observations from a continuous uniform distribution over the domain $[0, 1]$ via the `rand()` function. Then the `quantile()` function, along with the `dot` operator `(.)` to calculate each corresponding quantile of `triangDist`. Lines 8 and 9 plot a histogram of this `inverseSampledData`, using 30 bins. For large `N`, the histogram generated is a close approximation of the PDF of the underlying distribution. Lines 10-11 then plot the analytic PDF of the underlying distribution.

3.5 Families of Discrete Distributions

A *family of probability distributions* is a collection of probability distributions having some functional form that is parameterized by a well-defined set of parameters. In the discrete case, the PMF, $p(x\,;\,\theta) = \mathbb{P}(X = x)$, is parameterized by the *parameter* $\theta \in \Theta$ where Θ is called the *parameter space*. The (scalar or vector) parameter θ then affects the actual form of the PMF, including possibly the support of the random variable. Hence, technically a family of distributions is the collection of PMFs $p(\cdot\,;\,\theta)$ for all $\theta \in \Theta$.

In this section we present some of the most common families of *discrete distributions*. We consider the following: *discrete uniform distribution*, *binomial distribution*, *geometric distribution*, *negative binomial distribution*, *hypergeometric distribution* and *Poisson distribution*. Each of these is implemented in the Julia `Distributions` package. The approach that we take in the code examples of this section is to generate random variables from each distribution using first principles, as opposed to applying `rand()` on a distribution object, as was demonstrated in Listing 3.11 above. Understanding how to generate a random variable from a given distribution using first principles helps strengthen understanding of the associated probability models and processes.

In Listing 3.13 we illustrate how to create a distribution object for each of the discrete distributions that we investigate in this section. As output we print the parameters and the support of each distribution.

Listing 3.13: Families of discrete distributions

```
1  using Distributions
2  dists = [
3      DiscreteUniform(10,20),
4      Binomial(10,0.5),
5      Geometric(0.5),
6      NegativeBinomial(10,0.5),
7      Hypergeometric(30, 40, 10),
8      Poisson(5.5)]
9
10 println("Distribution \t\t\t\t\t\t Parameters \t Support")
11 reshape([dists ;  params.(dists) ;
12                 ((d)->(minimum(d),maximum(d))).(dists) ],
13                 length(dists),3)
```

```
Distribution                                Parameters      Support
6×3 Array{Any,2}:
 DiscreteUniform(a=10, b=20)                  (10, 20)       (10, 20)
 Binomial{Float64}(n=10, p=0.5)               (10, 0.5)      (0, 10)
 Geometric{Float64}(p=0.5)                    (0.5,)         (0, Inf)
 NegativeBinomial{Float64}(r=10.0, p=0.5)     (10.0, 0.5)    (0, Inf)
 Hypergeometric(ns=30, nf=40, n=10)           (30, 40, 10)   (0, 10)
 Poisson{Float64}(λ=5.5)                      (5.5,)         (0, Inf)
```

Lines 2-8 are used to define an array of distribution objects. The help provided by the distributions package is useful. Use `?` «Name» where «Name» may be `DiscreteUniform`, `Binomial`, etc. Lines 10-13 result in output that is a 6×3 array of type `Any`. The first column is the actual distributions object, the second column has the distributional parameters, and the third column represents the support. The parameters and the support for each distribution are presented in more detail later in this section. Note the use of an anonymous function `(d)->(minimum(d),maximum(d))` applied via '`.`' to each element of `dists`. This function returns a tuple. The use of `reshape()` transforms the array of arrays into a matrix of the desired dimensions.

Discrete Uniform Distribution

The *discrete uniform distribution* is simply a probability distribution that places equal probabilities for all equal outcomes. One example is given by the probability of the outcomes of a die toss. The probability of each possible outcome for a fair, six-sided die is given by,

$$\mathbb{P}(X = x) = \frac{1}{6} \qquad \text{for } x = 1, \ldots, 6.$$

Listing 3.14 simulates N tosses of a die, and then calculates and plots the proportion of times each possible outcome occurs, along with the PMF. The plot is in Figure 3.9. For large values of N, the proportion of counts for each outcome converges to 1/6.

Listing 3.14: Discrete uniform die toss

```
1  using StatsBase, Plots; pyplot()
2
3  faces, N = 1:6, 10^6
4  mcEstimate = counts(rand(faces,N), faces)/N
5
6  plot(faces, mcEstimate,
7          line=:stem, marker=:circle,
8          c=:blue, ms=10, msw=0, lw=4, label="MC estimate")
9  plot!([i for i in faces], [1/6 for _ in faces],
10         line=:stem, marker=:xcross, c=:red,
11         ms=6, msw=0, lw=2, label="PMF",
12         xlabel="Face number", ylabel="Probability", ylims=(0,0.22))
```

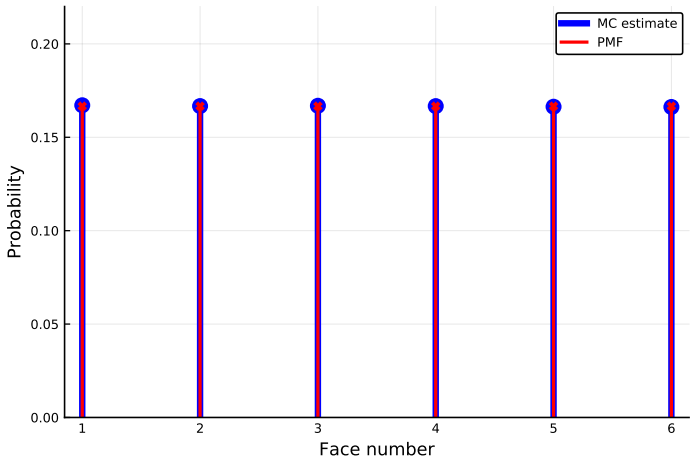

Figure 3.9: A discrete uniform PMF.

In line 3 we define all possible outcomes of our six-sided die, along with how many die tosses we will simulate. Line 4 uniformly and randomly generates `N` observations from our die, and then uses the `counts()` function to calculate proportion of times each outcome occurs. Note that applying `rand(DiscreteUniform(1,6),N)` would yield a statistically identical result to `rand(faces,N)`. Line 5 uses the `stem` function to create a stem plot of the proportion of times each outcome occurs, while line 6 plots the analytic PMF of our six-sided die.

Binomial Distribution

The *binomial distribution* is a discrete distribution which arises where multiple identical and independent yes/no, true/false, success/failure trials (also known as *Bernoulli trials*) are performed. For each trial, there can only be two outcomes, and the probability weightings of each unique trial must be the same.

As an example, consider a two-sided coin, which is flipped n times in a row. If the probability of obtaining a head in a single flip is p, then the probability of obtaining x heads total is given by the PMF,

$$\mathbb{P}(X = x) = \binom{n}{x} p^x (1-p)^{n-x} \qquad \text{for } x = 0, 1, \ldots, n.$$

Listing 3.15 simulates $n = 10$ tosses of a fair coin ($p = 1/2$), `N` times total, with success probability p, and calculates the proportion of times each possible outcome occurs. Observe that in the `Distributions` package, `pdf()` applied to a discrete distribution yields the PMF. In fact, the PMF is often loosely called a PDF (density) in statistics. The result are presented in Figure 3.10.

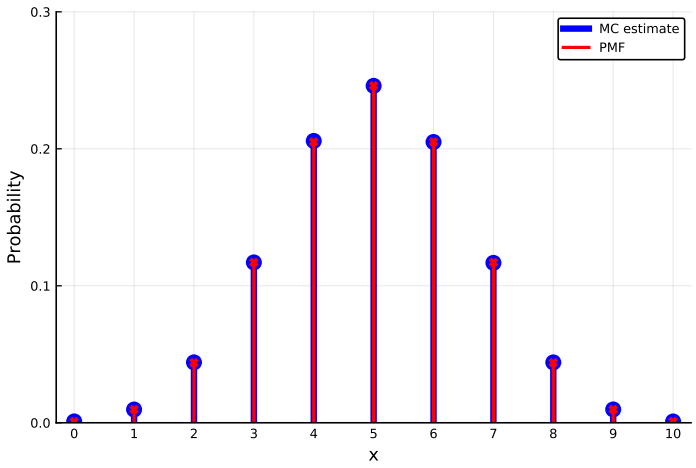

Figure 3.10: Binomial PMF for number of heads in 10 flips each with $p = 0.5$.

Listing 3.15: Coin flipping and the binomial distribution

```
1  using StatsBase, Distributions, Plots; pyplot()
2
3  binomialRV(n,p) = sum(rand(n) .< p)
4
5  p, n, N = 0.5, 10, 10^6
6
7  bDist = Binomial(n,p)
8  xGrid = 0:n
9  bPmf = [pdf(bDist,i) for i in xGrid]
10 data = [binomialRV(n,p) for _ in 1:N]
11 pmfEst = counts(data,0:n)/N
12
13 plot( xGrid, pmfEst,
14         line=:stem, marker=:circle,
15         c=:blue, ms=10, msw=0, lw=4, label="MC estimate")
16 plot!( xGrid, bPmf,
17          line=:stem, marker=:xcross, c=:red,
18          ms=6, msw=0, lw=2, label="PMF", xticks=(0:1:10),
19          ylims=(0,0.3), xlabel="x", ylabel="Probability")
```

In line 3 we define the function `binomialRV()`. It generates a binomial random variable from first principles by creating an array of uniform $[0, 1]$ values of length `n` with `rand(n)`. We then use `.<` to compare each value (element-wise) to `p`. The result is a vector of booleans, with each one set to `true` with probability `p`. Summing up this vector creates the binomial random variable. In line 9 we create a vector incorporating the values of the binomial PMF. Note that in the Julia distributions package, PMFs are created via `pdf()`. Line 10 is where we generate `N` random values. In line 11 we use `counts()` from the `StatsBase` package to count how may times each outcome occurred, for `0:n` heads. We then normalize via division by `N`. The remainder of the code creates the plot.

Note that the Binomial distribution describes part of the fishing example in Section 2.1, where we sample with replacement. This is because the probability of success (i.e. fishing a gold fish) remains unchanged regardless of how many times we have sampled from the pond.

Geometric Distribution

Another distribution associated with Bernoulli trials is the *geometric distribution*. In this case, consider an infinite sequence of independent trials, each with success probability p, and let X be the first trial that is successful. Using first principles it is easy to see that the PMF is,

$$\mathbb{P}(X = x) = p\,(1-p)^{x-1} \qquad \text{for } x = 1, 2, \ldots. \tag{3.14}$$

An alternative version of the geometric distribution is the distribution of the random variable $\tilde{X}$, counting the number of failures until success. Observe that for every sequence of trials, $\tilde{X} = X - 1$. From this it is easy to relate the PMFs of the random variables and see that,

$$\mathbb{P}(\tilde{X} = x) = p(1-p)^{x} \qquad \text{for } x = 0, 1, 2, \ldots.$$

In the Julia `Distributions` package, `Geometric` stands for the distribution of $\tilde{X}$, not X.

We now look at an example involving the popular casino game of roulette. Roulette is a game of chance, where a ball is spun on the inside edge of a horizontal wheel. As the ball loses momentum, it eventually falls vertically down, and lands on one of 37 spaces, numbered 0 to 36. There are 18 black spaces, 18 red, and a single space ('zero') is green. Each spin of the wheel is independent, and each of the possible 37 outcomes is equally likely. Now let us assume that a gambler goes to the casino and plays a series of roulette spins. There are various ways to bet on the outcome of roulette, but in this case he always bets on black (if the ball lands on black he wins, otherwise he loses). Say that the gambler plays until his first win. In this case, the number of plays is a geometric random variable with support $x = 1, 2, \ldots$. Listing 3.16 simulates this scenario and creates Figure 3.11.

Listing 3.16: The geometric distribution

```
using StatsBase, Distributions, Plots; pyplot()

function rouletteSpins(p)
    x = 0
    while true
        x += 1
        if rand() < p
            return x
        end
    end
end

p, xGrid, N = 18/37, 1:7, 10^6

mcEstimate = counts([rouletteSpins(p) for _ in 1:N],xGrid)/N

gDist = Geometric(p)
gPmf = [pdf(gDist,x-1) for x in xGrid]

plot(xGrid, mcEstimate, line=:stem, marker=:circle,
        c=:blue, ms=10, msw=0, lw=4, label="MC estimate")
plot!( xGrid, gPmf, line=:stem, marker=:xcross,
        c=:red, ms=6, msw=0, lw=2, label="PMF",
        ylims=(0,0.5), xlabel="x", ylabel="Probability")
```

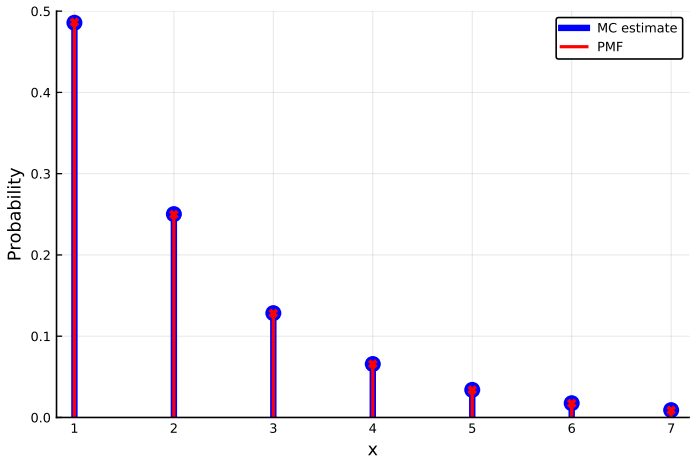

Figure 3.11: A geometric PMF.

The function `rouletteSpins()` defined in lines 3-11 is a straightforward way to generate a geometric random variable with support $1, 2, \ldots$ as X above. Lines 5-10 loop until a value is returned from the function. In each iteration, we increment `x` and check if we have a success (an event happening with probability p) via, `rand() < p`. The remainder of the code is similar to the previous listing. Consider the second argument to `pdf()` in line 18. Here `x-1` is used because the built-in geometric distribution is for the random variable $\tilde{X}$ above, which starts at 0, while we are interested in the geometric random variable starting at 1.

Negative Binomial Distribution

Recall the previous example above of a roulette gambler. Assume now that the gambler plays until he wins for the r'th time (in the previous example $r = 1$). The *negative binomial distribution* describes this situation. That is, a random variable X follows this distribution, if it describes the number of trials until the r'th success. The PMF is given by,

$$\mathbb{P}(X = x) = \binom{x-1}{r-1} p^r (1-p)^{x-r} \quad \text{for} \quad x = r, r+1, r+2, \ldots.$$

Notice that with $r = 1$ the expression reduces to the geometric PMF (3.14). Similarly to the geometric case, there is an alternative version of the negative binomial distribution. Let $\tilde{X}$ denote the number of failures until the r'th success. Here, like in the geometric case, when both random variables are coupled on the same sequence of trials, we have, $\tilde{X} = X - r$. As a result:

$$\mathbb{P}(\tilde{X} = x) = \binom{x+r-1}{x} p^r (1-p)^x \quad \text{for } x = 0, 1, 2, \ldots.$$

To help reinforce this, in Listing 3.17 below we simulate a gambler who bets consistently on black much like in the previous example, and determine the PMF for $r = 5$. That is, we determine the probabilities that x plays will occur up to the 5'th success (or win).

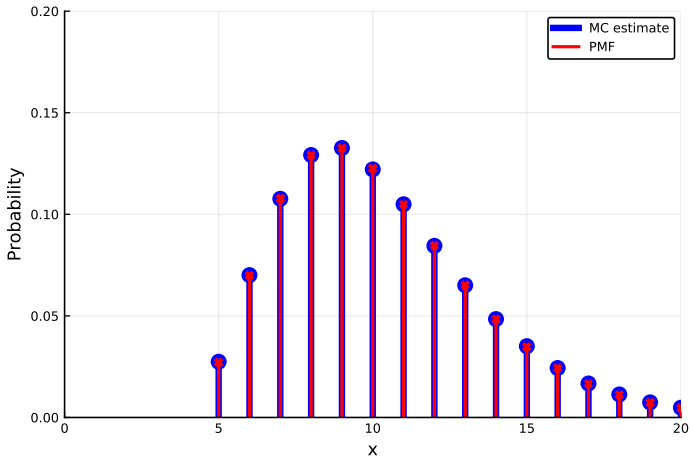

Figure 3.12: The PMF of negative binomial with $r = 5$ and $p = 18/37$.

Listing 3.17: The negative binomial distribution

```
using StatsBase, Distributions, Plots

function rouletteSpins(r,p)
    x = 0
    wins = 0
    while true
        x += 1
        if rand() < p
            wins += 1
            if wins == r
                return x
            end
        end
    end
end

r, p, N = 5, 18/37,10^6
xGrid = r:r+15

mcEstimate = counts([rouletteSpins(r,p) for _ in 1:N],xGrid)/N

nbDist = NegativeBinomial(r,p)
nbPmf = [pdf(nbDist,x-r) for x in xGrid]

plot( xGrid, mcEstimate,
        line=:stem, marker=:circle, c=:blue,
        ms=10, msw=0, lw=4, label="MC estimate")
plot!( xGrid, nbPmf, line=:stem,
         marker=:xcross, c=:red, ms=6, msw=0, lw=2, label="PMF",
         xlims=(0,maximum(xGrid)), ylims=(0,0.2),
         xlabel="x", ylabel="Probability")
```

This code is similar to the previous listing. The main difference is in the function `rouletteSpins()`, which now accepts both `r` and `p` as arguments. It is a straightforward implementation of the negative binomial story. A value is returned in line 11 only once the number of wins equals `r`. In a similar manner to the geometric example notice that in line 23 we use `x-r` for the argument of the `pdf()` function. This is because `NegativeBinomial` in the `Distributions package` stands for a distribution with support, $x = 0, 1, 2, \ldots$ and not $x = r, r+1, r+2, \ldots$ as we desire.

Hypergeometric Distribution

Moving on from Bernoulli trials, we now consider the *hypergeometric distribution*. To put it in context, consider the fishing problem discussed in Section 2.1, specifically the case where we fish without replacement. In this scenario, each time we sample from the population it decreases, and hence the probability of success changes for each subsequent sample. The hypergeometric distribution describes this situation. The PMF given by,

$$p(x) = \frac{\binom{K}{x}\binom{L-K}{n-x}}{\binom{L}{n}} \qquad \text{for } x = \max(0, n+K-L), \ldots, \min(n, K).$$

Here the parameter L is the population size, and K is the number of successes present in the population (this implies that $L - K$ is the number of failures present in the population). The parameter n is the number of samples taken from the population, and the input argument x is the number of successful samples observed. Hence a hypergeometric random variable X with $\mathbb{P}(X = x) = p(x)$ describes the number of successful samples when *sampling without replacement*. Note that the expression for $p(x)$ can be deduced directly via combinatorial counting arguments.

To understand the support of the distribution first consider the least possible value, $\max(0, n + K - L)$. It is either 0, or $n + K - L$ if $n > L - K$. The latter case stems from a situation where the number of samples n, is greater than the number of failures present in the population. That is, in such a case the least possible number of successes that can be sampled is,

$$\text{number of samples } (n) \quad - \quad \text{number of failures in the population } (L - K).$$

As for the upper value of the support, it is $\min(n, K)$ because if $K < n$ then it isn't possible to sample only successes. Note that in general if the sample size n is not "too big" then the support reduces to $x = 0, \ldots, n$.

To help illustrate this distribution, we look at an example where we compare several hypergeometric distributions simultaneously. As before, let us consider a pond which contains a combination of gold and silver fish. In this example, there are $N = 500$ fish total, and we will define the catch of a gold fish a success, and a silver fish a failure. Now say that we sample $n = 30$ fish without replacement. We consider several of these cases, where the only difference between each is the number of successes, K, (gold fish) in the population.

Listing 3.18 below plots the PMF's of 5 different hypergeometric distributions based on the number of successes in the population. The results are shown in Figure 3.13. It can be observed

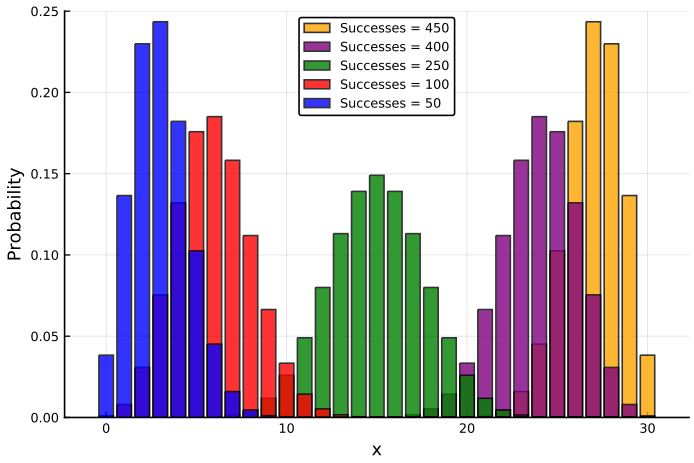

Figure 3.13: A comparison of several hypergeometric distributions for different proportions of successes in a population.

that as the number of successes present in the population increases, the PMF shifts further towards the right. Note that in the Julia `Distributions` package, `Hypergeometric` is parameterized via the number of successes (first argument) and number of failures (second argument), with the third argument being the sample size. This is slightly different to our parameterization above, which uses N, K and n.

Listing 3.18: Comparison of several hypergeometric distributions

```
1  using Distributions, Plots; pyplot()
2
3  L, K, n  = 500, [450, 400, 250, 100, 50], 30
4  hyperDists = [Hypergeometric(k,L-k,n) for k in K]
5  xGrid = 0:1:n
6  pmfs = [ pdf.(dist, xGrid) for dist in hyperDists ]
7  labels = "Successes = " .* string.(K)
8
9  bar( xGrid, pmfs,
10         alpha=0.8, c=[:orange :purple :green :red :blue ],
11         label=hcat(labels...), ylims=(0,0.25),
12         xlabel="x", ylabel="Probability", legend=:top)
```

In line 3 we define the population size, `L`, the sample size `n`, and the array `K`, which contains the number of successes in the population, for each of our 5 scenarios. In line 4 the `Hypergeometric()` constructor is used to create several hypergeometric distributions. The constructor takes three arguments, the number of successes in the population `k`, the number of failures in the population `L-k`, and the number of times we sample from the population without replacement `n`. This constructor is then wrapped in a comprehension in order to create an array of different hypergeometric distributions, `hyperDists`. We then create an array of arrays, `pmfs` in line 6, by applying the `pdf()` function on each distribution. In lines 9-12, the `bar()` function is used to plot a bar chart of the PMF for each hypergeometric distribution in `hyperDists`. Notice the use of `hcat(labels...)` to convert `labels` from `Array{String,1}` to `Array{String,2}` which is required to label the plots plots in `bar()`.

Poisson Distribution and Poisson Process

The *Poisson process* is a *stochastic process* (random process) which can be used to model occurrences of events over time (or more generally in space). It may be used to model the arrival of customers to a system, the emission of particles from radioactive material, or packets arriving to a communication router. The Poisson process is the canonical example of a *point process* capturing the most sensible model for completely random occurrences over time. A full description and analysis of the Poisson process is beyond our scope, however we provide an overview of the basics.

In a Poisson process, during an infinitesimally small time interval, Δt, it is assumed that (as $\Delta t \to 0$) there is an occurrence with probability $\lambda \Delta t$, and no occurrence with probability $1 - \lambda \Delta t$. Furthermore, as $\Delta t \to 0$, it is assumed that the chance of 2 or more occurrences during an interval of length Δt tends to 0. Here $\lambda > 0$ is the *intensity* of the Poisson process, and has the property that when multiplied by an interval of length T, the mean number of occurrences during the interval is λT.

The exponential distribution, discussed in the next section, is closely related to the Poisson process as the times between occurrences in the Poisson process are exponentially distributed. Another closely related distribution is the *Poisson distribution* that we discuss now. For a Poisson process over the time interval $[0, T]$ the number of occurrences satisfy,

$$\mathbb{P}(x \text{ Poisson process occurrences during interval } [0,T]) = e^{-\lambda T} \frac{(\lambda T)^x}{x!} \qquad \text{for } x = 0, 1, \ldots.$$

The PMF $p(x) = e^{-\lambda} \lambda^x / x!$ for $x = 0, 1, 2, \ldots$ describes the Poisson distribution, the mean of which is λ. Hence the number of occurrences in a Poisson process during $[0, T]$ is Poisson distributed with parameter (and mean) λT. Note that in applied statistics, the Poisson distribution is also sometimes taken as a model for occurrences, without explicitly considering a Poisson process. For example, assume that based on previous measurements, on average 5.5 people arrive at a hair salon during rush-hour, then the probability of observing x people during rush-hour can be modeled by the PMF of the Poisson distribution.

The Poisson process possesses many elegant analytic properties, and these sometimes come as an aid when considering Poisson distributed random variables. One such (seemingly magical) property is to consider the random variable $N \geq 0$ such that,

$$\prod_{i=1}^{N} U_i \geq e^{-\lambda} > \prod_{i=1}^{N+1} U_i, \tag{3.15}$$

where $U_1, U_2, \ldots$ is a sequence of i.i.d. uniform$(0, 1)$ random variables and $\prod_{i=1}^{0} U_i \equiv 1$. It turns out that seeking such a random variable N produces an efficient recipe for generating a Poisson random variable. That is, the N defined by (3.15) is Poisson distributed with mean λ. Notice that the recipe dictated by (3.15) is to continue multiplying uniform random variables to a "running product" until the product goes below the desired level $e^{-\lambda}$.

Returning to the hair salon example mentioned above, Listing 3.19 below simulates this scenario, and compares the numerically estimated result against the PMF. The results are presented in Figure 3.14.

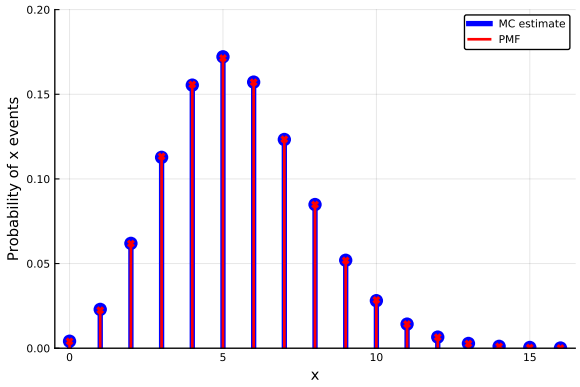

Figure 3.14: The PMF of a Poisson distribution with mean $\lambda = 5.5$.

Listing 3.19: The Poisson distribution

```
1  using StatsBase, Distributions, Plots; pyplot()
2
3  function prn(lambda)
4      k, p = 0, 1
5      while p > MathConstants.e^(-lambda)
6          k += 1
7          p *= rand()
8      end
9      return k-1
10 end
11
12 xGrid, lambda, N = 0:16, 5.5, 10^6
13
14 pDist = Poisson(lambda)
15 bPmf = pdf.(pDist,xGrid)
16 data = counts([prn(lambda) for _ in 1:N],xGrid)/N
17
18 plot( xGrid, data,
19         line=:stem, marker=:circle,
20         c=:blue, ms=10, msw=0, lw=4, label="MC estimate")
21 plot!( xGrid, bPmf, line=:stem,
22         marker=:xcross, c=:red, ms=6, msw=0, lw=2, label="PMF",
23         ylims=(0,0.2), xlabel="x", ylabel="Probability of x events")
```

In lines 3-10 the function `prn()`, standing for "Poisson random number", is defined. It implements (3.15) in a straightforward manner and takes a single argument, the expected arrival rate for our interval `lambda`. Line 16 calls `prn()` a total of `N` times, counts occurrences, and normalizes them by `N` to obtain Monte Carlo estimates of the Poisson probabilities. Lines 18-23 plot these Monte Carlo estimates as well as the PMF.

3.6 Families of Continuous Distributions

Like families of discrete distributions, families of continuous distributions are parametrized by a well-defined set of parameters. Typically the PDF, $f(x\,;\theta)$, is parameterized by the *parameter* $\theta \in \Theta$. Hence, technically a family of continuous distributions is the collection of PDFs $f(\cdot\,;\theta)$ for all $\theta \in \Theta$.

In this section we present some of the most common families of *continuous distributions*. We consider the following: *continuous uniform distribution*, *exponential distribution*, *gamma distribution*, *beta distribution*, *Weibull distribution*, *Gaussian (normal) distribution*, *Rayleigh distribution* and *Cauchy distribution*. As was done with discrete distributions, the approach taken in the code examples involves generating random variables from each distribution using first principles. We also occasionally dive into related concepts that naturally arise in the context of a given distribution. These include the squared coefficient of variation, special functions (gamma and beta), hazard rates, various transformations, and heavy tails.

In Listing 3.20 we illustrate how to create a distribution object for each of the continuous distributions we cover. The listing and its output style is similar to Listing 3.13 used for discrete distributions.

Listing 3.20: Families of continuous distributions

```
using Distributions
dists = [
    Uniform(10,20),
    Exponential(3.5),
    Gamma(0.5,7),
    Beta(10,0.5),
    Weibull(10,0.5),
    Normal(20,3.5),
    Rayleigh(2.4),
    Cauchy(20,3.5)]

println("Distribution \t\t\t Parameters \t Support")
reshape([dists ;  params.(dists) ;
                ((d)->(minimum(d),maximum(d))).(dists) ],
                length(dists),3)
```

```
Distribution                      Parameters      Support
8×3 Array{Any,2}:
 Uniform{Float64}(a=10.0, b=20.0)  (10.0, 20.0)  (10.0, 20.0)
 Exponential{Float64}(θ=3.5)       (3.5,)        (0.0, Inf)
 Gamma{Float64}(α=0.5, θ=7.0)      (0.5, 7.0)    (0.0, Inf)
 Beta{Float64}(α=10.0, β=0.5)      (10.0, 0.5)   (0.0, 1.0)
 Weibull{Float64}(α=10.0, θ=0.5)   (10.0, 0.5)   (0.0, Inf)
 Normal{Float64}(μ=20.0, σ=3.5)    (20.0, 3.5)   (-Inf, Inf)
 Rayleigh{Float64}(σ=2.4)          (2.4,)        (0.0, Inf)
 Cauchy{Float64}(μ=20.0, σ=3.5)    (20.0, 3.5)   (-Inf, Inf)
```

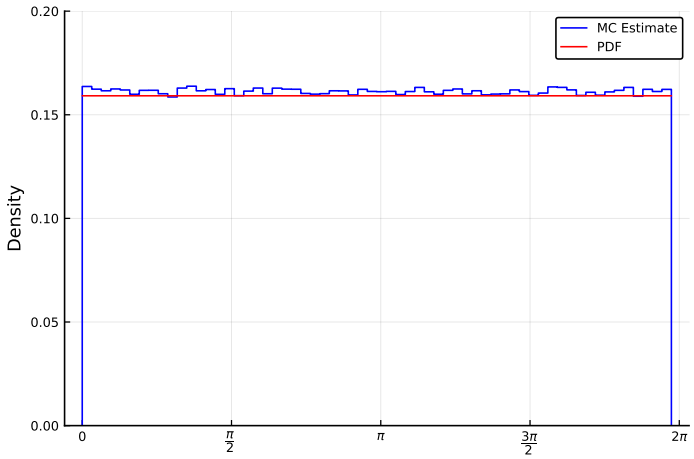

Figure 3.15: The PDF of a continuous uniform distribution over $[0, 2\pi]$.

Continuous Uniform Distribution

The *continuous uniform distribution* describes the case where the outcome of a continuous random variable X has a constant likelihood of occurring over some finite interval. Since the integral of the PDF must equal one, given an interval (a, b), the PDF is given by

$$f(x) = \begin{cases} \dfrac{1}{b-a} & \text{for } a \leqslant x \leqslant b, \\ 0 & \text{for } x < a \text{ or } x > b. \end{cases}$$

As an example, consider the case of a fast spinning circular disk, such as a hard drive. Imagine now there is a small defect on the disk, and we define X as the clockwise angle (in radians) the defect makes with the read head at an arbitrary time. In this case X is modeled by the continuous uniform distribution over $x \in [0, 2\pi]$. Listing 3.21 creates Figure 3.15 where we compare the PDF and a Monte Carlo based estimate.

Listing 3.21: Uniformly distributed angles

```
using Distributions, Plots, LaTeXStrings; pyplot()

cUnif = Uniform(0,2π)
xGrid, N = 0:0.1:2π, 10^6

stephist( rand(N)*2π, bins=xGrid,
        normed=:true, c=:blue,
        label="MC Estimate")
plot!( xGrid, pdf.(cUnif,xGrid),
        c=:red,ylims=(0,0.2),label="PDF", ylabel="Density",xticks=([0:π/2:2π;],
        ["0", L"\dfrac{\pi}{2}", L"\pi", L"\dfrac{3\pi}{2}", L"2\pi"]))
```

In line 3 the `Uniform()` function is used to create a continuous uniform distribution over the domain $[0, 2\pi]$. In Julia you can use the unicode character π or `pi`. In line 6, `rand(N)*2π` is used to generate `N` uniform random values on $[0, 2\pi]$. An alternative would be to use `rand(cUnif,N)`. In our case, we simulate `N` continuous uniform random variables over the domain $[0, 1]$ via the `rand()` function, and then scale each of these by a factor of 2π. A histogram of this data is then plotted using `stephist()`. Notice that the `bins` argument is set to the range `xGrid`. An alternative would be to specify an integer number of bins. Line 9 uses `pdf()` on the distribution object `cUnif` to plot the analytic PDF. Notice the use of `L` from the `LaTexStrings` package in line 11 for creating formulas.

Exponential Distribution

As alluded to in the discussion of the Poisson process above, the *exponential distribution* is often used to model random durations between occurrences. A non-negative random variable X, exponentially distributed with a rate parameter $\lambda > 0$ has PDF,

$$f(x) = \lambda e^{-\lambda x}.$$

As can be verified, the mean is $1/\lambda$, the variance is $1/\lambda^2$, and the CCDF is $\bar{F}(x) = e^{-\lambda x}$. Note that in Julia, the distribution is parameterized by the mean, rather than by λ. Hence to create an exponential distribution object with $\lambda = 0.2$ (for example), one would use `Exponential(5.0)`.

Exponential random variables possess a *lack of memory* property. It can be verified that,

$$\mathbb{P}(X > t + s \mid X > t) = \mathbb{P}(X > s).$$

To show this, expand the conditional probability and use the CCDF. A similar property holds for geometric random variables. This hints at the fact that exponential random variables are the continuous analogs of geometric random variables.

To explore this further, consider a transformation of an exponential random variable X, $Y = \lfloor X \rfloor$, where $\lfloor \cdot \rfloor$ represents the mathematical *floor function*. In this case, Y is no longer a continuous random variable, but is discrete in nature, taking on values in the set $\{0, 1, 2, \ldots\}$.

We can show that the PMF of Y is,

$$p_Y(y) = \mathbb{P}(\lfloor X \rfloor = y) = \int_y^{y+1} \lambda e^{-\lambda x}\, dx = (e^{-\lambda})^y (1 - e^{-\lambda}) \qquad \text{for} \qquad y = 0, 1, 2, \ldots.$$

If we then set $p = 1 - e^{-\lambda}$, we observe that Y is a geometric random variable which starts at 0 and has success parameter p.

In Listing 3.22, we present a comparison between the PMF of the floor of an exponential random variable, and the PMF of the geometric distribution covered in Section 3.5. Remember that in Julia the support of `Geometric()` starts at $x = 0$. The listing creates Figure 3.16.

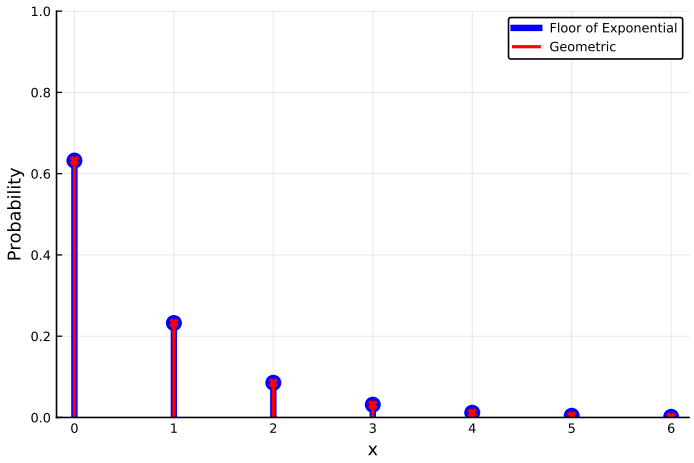

Figure 3.16: The PMF of the floor of an exponential random variable is a geometric distribution.

Listing 3.22: Flooring an exponential random variable

```
1  using StatsBase, Distributions, Plots; pyplot()
2
3  lambda, N = 1, 10^6
4  xGrid = 0:6
5
6  expDist = Exponential(1/lambda)
7  floorData = counts(convert.(Int,floor.(rand(expDist,N))), xGrid)/N
8  geomDist = Geometric(1-MathConstants.e^-lambda)
9
10 plot( xGrid, floorData,
11         line=:stem, marker=:circle,
12         c=:blue, ms=10, msw=0, lw=4,
13         label="Floor of Exponential")
14 plot!( xGrid, pdf.(geomDist,xGrid),
15         line=:stem, marker=:xcross,
16         c=:red, ms=6, msw=0, lw=2,
17         label="Geometric", ylims=(0,1),
18         xlabel="x", ylabel="Probability")
```

In line 6 the `Exponential()` function is used to create the exponential distribution object, `expDist`. Note that the function takes one argument, the inverse of the mean, hence `1/lambda` is used. In line 7 we use the `rand()` function to sample `N` times from the exponential distribution `expDist`. The `floor()` function is then used to round each observation down to the nearest integer, and the `convert()` function is used to convert the values from `Float64` to `Int` type. The function `counts()` is then used to count how many times each integer in `xGrid` occurs, and the proportions are stored in the array `floorData`. In line 8 we use the `Geometric()` function, covered previously, to create a geometric distribution object with probability of success `1-MathConstants.e^-lambda`. Lines 10-18 plot the results where `pdf()` is applied to `geomDist` in line 14.

Gamma Distribution and the Squared Coefficient of Variation

The *gamma distribution* is commonly used for modeling asymmetric non-negative data. It generalizes the exponential distribution and the chi-squared distribution (covered in Section 5.2, in the context of statistical inference). To introduce this distribution, consider the following example, where the lifetimes of light bulbs are exponentially distributed with mean λ^{-1}. Now imagine we are lighting a room continuously with a single light bulb, and that we replace the bulb with a new one when it burns out. If we start at time 0, what is the distribution of time until n bulbs are replaced?

One way to describe this time is by the random variable T, where,

$$T = X_1 + X_2 + \ldots + X_n,$$

and X_i are i.i.d. exponential random variables representing the lifetimes of light bulbs. It turns out that the distribution of T is a gamma distribution. In this case, since it is a sum of i.i.d. exponential random variables it is also called an *Erlang distribution*.

We now introduce the PDF of the gamma distribution. It is a function (in x) proportional to $x^{\alpha-1}e^{-\lambda x}$, where the non-negative parameters λ and α are called the *rate parameter* and *shape parameter* respectively. In order to normalize this function we need to divide by,

$$\int_0^\infty x^{\alpha-1}e^{-\lambda x}\,dx.$$

It turns out that this integral can be represented by $\Gamma(\alpha)/\lambda^\alpha$, where $\Gamma(\cdot)$ is a well known mathematical special function called the *gamma function*, see (3.16). We investigate the gamma function, and the related *beta function* and *beta distribution* below. After using the gamma function for normalization, the PDF of the gamma distribution is,

$$f(x) = \frac{\lambda^\alpha}{\Gamma(\alpha)}x^{\alpha-1}e^{-\lambda x}.$$

In the light bulbs case, we have that $T \sim \text{Gamma}(n, \lambda)$, with shape parameter $\alpha = n$. In general for a gamma random variable, $Y \sim \text{Gamma}(\alpha, \lambda)$, the shape parameter α does not have to be a whole number. It can analytically be evaluated that,

$$\mathbb{E}[Y] = \frac{\alpha}{\lambda}, \qquad \text{and} \qquad \text{Var}(Y) = \frac{\alpha}{\lambda^2}.$$

We also take the opportunity here to introduce another general notion of variability, often used for non-negative random variables, namely the *squared coefficient of variation*,

$$\text{SCV} = \frac{\text{Var}(Y)}{\mathbb{E}[Y]^2}.$$

The SCV is a normalized, or unit-less version of the variance. The lower it is, the less variability in the random variable. It can be seen that for a gamma random variable, the SCV is $1/\alpha$ and for our light bulb example above, $\text{SCV}(T) = 1/n$. Hence for large n, i.e. more light bulbs, there is less variability.

Listing 3.23 considers the three cases of $n = 1$, $n = 10$ and $n = 50$ light bulbs (the case of $n = 1$ is exponential). For each scenario, gamma random variables are simulated by generating sums of

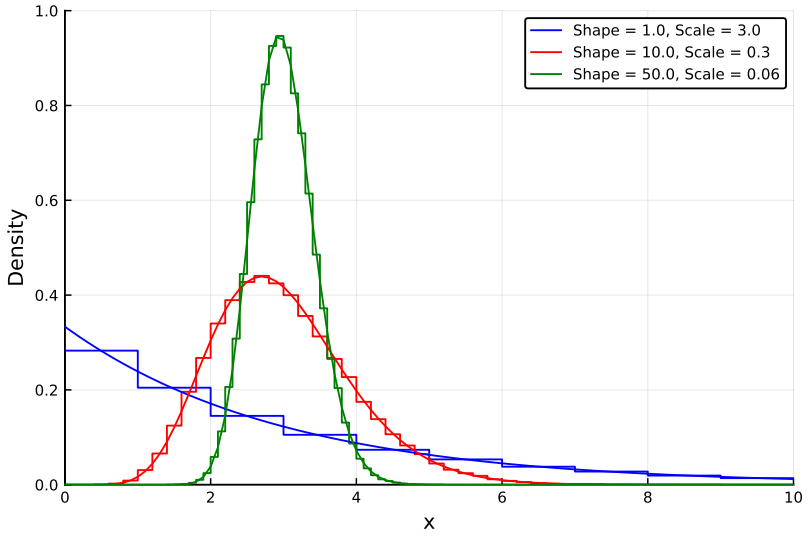

Figure 3.17: Plot of histograms of Monte Carlo simulated gamma observations, against their analytic PDFs.

exponential random variables. In each case, we set the rate parameter for the light bulbs at λn, so that the mean time until all light bulbs run out is $1/\lambda$, independent of n. The resulting histograms are then compared to the theoretical gamma PDF's. Note that the Julia function `Gamma()` is not parametrized by λ, but by $1/\lambda$ in a similar fashion to the `Exponential()` function. This inverse of the rate parameter is called the *scale parameter*.

Listing 3.23: Gamma random variable as a sum of exponentials

```
using Distributions, Plots; pyplot()

lambda, N = 1/3, 10^5
bulbs = [1,10,50]
xGrid = 0:0.1:10
C = [:blue :red :green]
dists = [Gamma(n,1/(n*lambda)) for n in bulbs]

function normalizedData(d::Gamma)
    sh = Int64(shape(d))
    data = [sum(-(1/(sh*lambda))*log.(rand(sh))) for _ in 1:N]
end

L = [ "Shape = "*string.(shape.(i))*", Scale = "*
    string.(round.(scale.(i),digits=2)) for i in dists ]

stephist( normalizedData.(dists), bins=50,
    normed=:true, c=C, xlims=(0,maximum(xGrid)),ylims=(0,1),
    xlabel="x", ylabel="Density", label="")
plot!(xGrid, [pdf.(i,xGrid) for i in dists], c=C, label=reshape(L, 1,:))
```

In lines 3-6 we define the main variables of our problem. In line 4 we create the array `bulbs` which stores the number of bulbs for each case. In line 6 we create an array of colors which are used later for formatting the plots. In line 7 the `Gamma()` function is used along with a comprehension to create a Gamma distribution for each of our cases. The three Gamma distributions are stored in the array `dists`. Lines 9-12 define the function `normalizedData()` which operates on a Gamma distribution as specified via `::Gamma`. The function obtains the shape parameter of the input distribution via `shape()` and converts this to an integer. Then `-log.(rand(sh))` is a raw way of generating a unit mean collection of `sh` exponential random variables using the inverse probability transform. These are then scaled by the scalar, `(1/(sh*lambda))`. Lines 14-15 generate the string array `L` used for the legend. Notice the use of the `round()` function. The remainder of the code plots the histograms and the actual PDFs.

Beta Distribution and Mathematical Special Functions

The *beta distribution* is a commonly used distribution when seeking a parameterized shape over a finite support. It is parametrized by two non-negative parameters, α and β. It has a density proportional to $x^{\alpha-1}(1-x)^{\beta-1}$ for $x \in [0,1]$. By using different positive values of α and β, a variety of shapes can be produced. You may want to try and create such plots yourself to experiment. One common example is $\alpha = 1, \beta = 1$, in which case the distribution defaults to the uniform(0,1) distribution.

As was with the gamma distribution above, in the case of beta, we are also left to seek a normalizing constant K, such that when multiplied by $x^{\alpha-1}(1-x)^{\beta-1}$, the resulting function has a unit integral over $[0,1]$. In our case,

$$K = \frac{1}{\displaystyle\int_0^1 x^{\alpha-1}(1-x)^{\beta-1}\,dx},$$

and hence the PDF is $f(x) = K\,x^{\alpha-1}(1-x)^{\beta-1}$.

We now explore the beta distribution. By focusing on the normalizing constant, we gain further insight into the mathematical gamma function $\Gamma(\cdot)$, which is a component of the gamma distribution covered previously. It turns out that,

$$K = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}.$$

Mathematically, this is called the inverse of the *beta function*, evaluated at α and β. Let us focus solely on the gamma functions, with the purpose of demystifying their use in the gamma and beta distributions. The *gamma function* is a type of *special function*, and is defined as,

$$\Gamma(z) = \int_0^\infty x^{z-1}e^{-x}\,dx. \tag{3.16}$$

It is a continuous generalization of factorial. We know that for positive integer n,

$$n! = n \cdot (n-1)!, \qquad \text{with } 0! \equiv 1.$$

This is the recursive definition of factorial. The gamma function exhibits similar properties, and one can evaluate it via integration by parts,

$$\Gamma(z) = (z-1)\cdot\Gamma(z-1).$$

Note furthermore that, $\Gamma(1) = 1$. Hence we see that for integer values of z,

$$\Gamma(z) = (z-1)!.$$

We now illustrate this in Listing 3.24 and in the process take into consideration the mathematical function beta and the beta PDF. Observe the difference in Julia between lower case `gamma()`, the special mathematical function, and `Gamma()`, the constructor for the distribution. The same applies to `beta()` and `Beta()`.

Listing 3.24: The gamma and beta special functions

```
1  using SpecialFunctions, Distributions
2
3  a,b = 0.2, 0.7
4  x = 0.75
5
6  betaAB1 = beta(a,b)
7  betaAB2 = (gamma(a)gamma(b))/gamma(a+b)
8  betaAB3 = (factorial(a-1)factorial(b-1))/factorial(a+b-1)
9  betaPDFAB1 = pdf(Beta(a,b),x)
10 betaPDFAB2 = (1/beta(a,b))*x^(a-1) * (1-x)^(b-1)
11
12 println("beta($a,$b)     = $betaAB1,\t$betaAB2,\t$betaAB3 ")
13 println("betaPDF($a,$b) at $x = $betaPDFAB1,\t$betaPDFAB2")
```

```
beta(0.2,0.7)     = 5.576463695849875,    5.576463695849875,      5.576463695849877
betaPDF(0.2,0.7) at 0.75 = 0.34214492891381176, 0.34214492891381176
```

We use the `SpecialFunctions` package for `gamma()` and `beta()`. This package also introduces a method for `factorial()` that allows to evaluate $\Gamma(z)$ via `factorial(z-1)` even for non-integer z. In lines 6-8, the `beta()` special function at `a` and `b` is evaluated in three different ways.

Another important property of the gamma function that we encounter later on (in the context of the Chi squared distribution, which we touch on in Section 5.2) is that $\Gamma(1/2) = \sqrt{\pi}$. In Listing 3.25 we show this via numerical integration.

Listing 3.25: The gamma function at 1/2

```
1  using QuadGK, SpecialFunctions
2
3  g(x) = x^(0.5-1) * MathConstants.e^-x
4  quadgk(g,0,Inf)[1], sqrt(pi), gamma(1/2),  factorial(1/2-1)
```

```
(1.7724538355037913, 1.7724538509055159, 1.772453850905516, 1.772453850905516)
```

This example uses the `QuadGK` package, in the same manner as introduced in Listing 3.3. We can see that the numerical integration is in agreement with the analytically expected result.

Weibull Distribution and Hazard Rates

We now explore the *Weibull distribution* along with the concept of the *hazard rate function*, which is often used in *reliability analysis* and *survival analysis*. For a random variable T, representing the lifetime of an individual or a component, an interesting quantity is the instantaneous chance of failure at any time, given that the component has been operating without failure up to time x. This can be expressed as,

$$h(x) = \lim_{\Delta \to 0} \frac{1}{\Delta} \mathbb{P}(T \in [x, x+\Delta) \mid T > x).$$

Alternatively, by using the conditional probability and noticing that the PDF $f(x)$ satisfies $f(x)\Delta \approx \mathbb{P}(x \leq T < x + \Delta)$ for small Δ, we can express the above as,

$$h(x) = \frac{f(x)}{1 - F(x)}. \tag{3.17}$$

Here the function $h(\cdot)$ is called the hazard rate, and it is a common method of viewing the distribution for lifetime random variables T. In fact, we can reconstruct the CDF $F(x)$ by,

$$1 - F(x) = \exp\Big(-\int_0^x h(t)\, dt\Big). \tag{3.18}$$

Hence every continuous non-negative random variable can be described uniquely by its hazard rate. The *Weibull distribution* is naturally defined through the hazard rate by considering hazard rate functions that have a specific simple form. It is a distribution with,

$$h(x) = \lambda x^{\alpha - 1}, \tag{3.19}$$

where λ is positive and α takes on any real value. Notice that the parameter α gives the Weibull distribution different modes of behavior. If $\alpha = 1$ then the hazard rate is constant, in which case the Weibull distribution is actually an exponential distribution with rate λ. If $\alpha > 1$, then the hazard rate increases over time. This depicts a situation of "aging components", i.e. the longer a component has lived, the higher the instantaneous chance of failure. This is sometimes called *Increasing Failure Rate (IFR)*. Conversely, $\alpha < 1$ depicts a situation where the longer a component has lasted, the lower the chance of it failing (as is perhaps the case with totalitarian political regimes). This is sometimes called *Decreasing Failure Rate (DFR)*.

Based on (3.19) and using (3.18) we obtain the CDF and PDF,

$$F(x) = 1 - e^{-\frac{\lambda}{\alpha}x^\alpha}, \qquad \text{and} \qquad f(x) = \lambda x^{\alpha-1} e^{-\frac{\lambda}{\alpha}x^\alpha}. \tag{3.20}$$

Note that in Julia, the distribution is parameterized slightly differently via,

$$f(x) = \frac{\alpha}{\theta}\Big(\frac{x}{\theta}\Big)^{\alpha-1} e^{-(x/\theta)^\alpha} = \alpha\theta^{-\alpha} x^{\alpha-1} e^{-\theta^{-\alpha}x^\alpha},$$

where the bijection from λ to θ is,

$$\lambda = \alpha\theta^{-\alpha}, \qquad \text{and} \qquad \theta = \Big(\frac{\alpha}{\lambda}\Big)^{1/\alpha}. \tag{3.21}$$

In this case, θ is called the scale parameter and α is the shape parameter.

In Listing 3.26 we look at several hazard rate functions for different Weibull distributions using the parameterization (3.20), and show their differences in Figure 3.18. The example also shows how to use the `shape()` and `scale()` functions from the `Distributions` package.

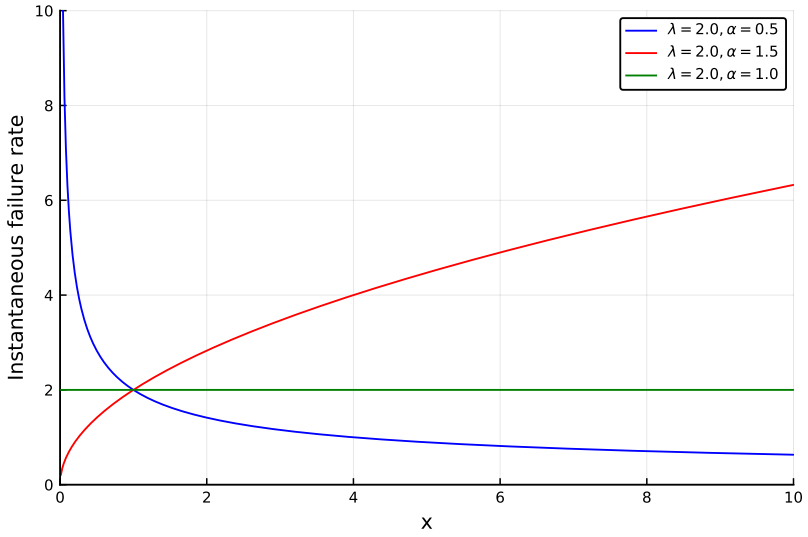

Figure 3.18: Hazard rate functions for different Weibull distributions.

Listing 3.26: Hazard rates and the Weibull distribution

```
using Distributions, Plots, LaTeXStrings; pyplot()

alphas = [0.5, 1.5, 1]
lam = 2

lambda(dist::Weibull) = shape(dist)*scale(dist)^(-shape(dist))
theta(lam,alpha) = (alpha/lam)^(1/alpha)

dists = [Weibull.(a,theta(lam,a)) for a in alphas]

hA(dist,x) = pdf(dist,x)/ccdf(dist,x)
hB(dist,x) = lambda(dist)*x^(shape(dist)-1)

xGrid = 0.01:0.01:10
hazardsA = [hA.(d,xGrid) for d in dists]
hazardsB = [hB.(d,xGrid) for d in dists]

println("Maximum difference between two implementations of hazard: ",
    maximum(maximum.(hazardsA-hazardsB)))

Cl = [:blue :red :green]
Lb = [L"\lambda=" * string(lambda(d)) * ",   " * L"\alpha =" * string(shape(d))
        for d in dists]

plot(xGrid, hazardsA, c=Cl, label=reshape(Lb, 1,:), xlabel="x",
        ylabel="Instantaneous failure rate", xlims=(0,10), ylims=(0,10))
```

```
Maximum difference between two implementations of hazard: 1.7763568394002505e-15
```

In line 6, we define the function `lambda()`, which operates on a `Weibull` type distribution and implements the first equation in (3.21). Note the type specification `::Weibull` and the use of the `shape()` and `scale()` functions. In line 7 we define the function `theta()` which implements the second equation in (3.21). Line 9 constructs three `Weibull` objects in the array `dists`. Lines 11 and 12 implement two alternative ways of calculating the hazard rate function, `hA()` and `hB()`. The first uses (3.17) and the second uses (3.19). Then in lines 18-19, we verify that the two implementations are in agreement. The remainder of the code creates Figure 3.18.

Gaussian (Normal) Distribution

Arguably, the most well known distribution is the *normal distribution*, also known as the *Gaussian distribution*. It is a symmetric "*bell curved*" shaped distribution, which can be found throughout nature. Examples include the distribution of heights among adult humans and noise disturbances of electrical signals. It is commonly exhibited due to the central limit theorem, which is covered in more depth in Section 5.3.

The Gaussian distribution is defined by two parameters, μ and σ^2, which are the mean and variance respectively. The mean μ can take on any value and σ^2 is restricted to be positive. The phrase *standard normal* signifies the case of a normal distribution with $\mu = 0$ and $\sigma^2 = 1$. In the general case, the PDF is given by,

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

The CDF of the normal distribution is not available as a simple expression. However, it is frequently needed and hence statistical tables or software are often used. The CDF of the standard normal random variable is,

$$\Phi(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-\frac{t^2}{2}}\, dt = \frac{1}{2}\Big(1 + \operatorname{erf}\big(\frac{x}{\sqrt{2}}\big)\Big). \tag{3.22}$$

The second expression represents $\Phi(\cdot)$ in terms of the *error function* $\operatorname{erf}(\cdot)$. It is a mathematical special function defined as,

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt.$$

With $\Phi(\cdot)$ (or alternately $\operatorname{erf}(\cdot)$) tabulated, one can move on to a general normal random variable with mean μ and variance σ^2. In this case, the CDF is available via,

$$\Phi\Big(\frac{x-\mu}{\sigma}\Big).$$

As an illustrative example, Listing 3.27 plots the standard normal PDF, along with its first and second derivatives in Figure 3.21. The first derivative is clearly 0 at the PDF's unique maximum at $x = 0$. The second derivative is 0 at the points $x = -1$ and $x = +1$. These are exactly the *inflection points* of the normal PDF (points where the function switches between being locally convex to locally concave or vice-versa). This code example also illustrates the use numerical derivatives from

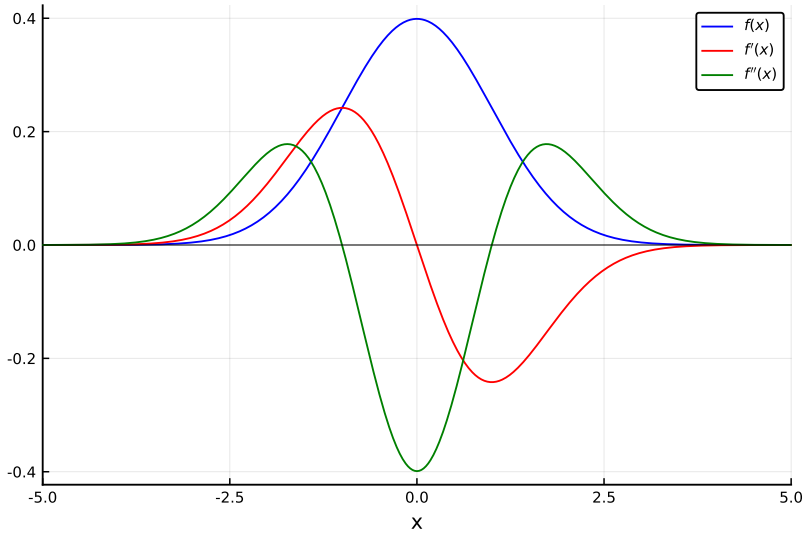

Figure 3.19: Plot of the standard normal PDF and its first and second derivatives.

the `Calculus` package. The code also presents two alternative ways of implementing $\Phi(\cdot)$ of (3.22) and shows they are equivalent. One way uses `cdf()` from the `Distributions` package and the other way uses `erf()` from the `SpecialFunctions` package.

Listing 3.27: Numerical derivatives of the normal density

```
1  using Distributions, Calculus, SpecialFunctions, Plots; pyplot()
2
3  xGrid = -5:0.01:5
4
5  PhiA(x) = 0.5*(1+erf(x/sqrt(2)))
6  PhiB(x) = cdf(Normal(),x)
7
8  println("Maximum difference between two CDF implementations: ",
9              maximum(PhiA.(xGrid) - PhiB.(xGrid)))
10
11 normalDensity(z) = pdf(Normal(),z)
12
13 d0 = normalDensity.(xGrid)
14 d1 = derivative.(normalDensity,xGrid)
15 d2 = second_derivative.(normalDensity, xGrid)
16
17 plot(xGrid, [d0 d1 d2], c=[:blue :red :green],label=[L"f(x)" L"f'(x)" L"f''(x)"])
18 plot!([-5,5],[0,0],  color=:black, lw=0.5, xlabel="x", xlims=(-5,5), label="")
```

```
Maximum difference between two CDF implementations: 1.1102230246251565e-16
```

Lines 5-9 are dedicated to showing the equivalence of the two ways of implementing $\Phi(\cdot)$. In line 11 we define the function `normalDensity()`, which takes an input z, and returns the corresponding value of the PDF of a standard normal distribution. Then in lines 14-15, the functions `derivative()` and `second_derivative()` are used to evaluate the first and second derivatives of `normalDensity` respectively. The curves are plotted in lines 17-18.

Rayleigh Distribution and the Box-Muller Transform

We now consider an exponentially distributed random variable, X, with rate parameter $\lambda = \sigma^{-2}/2$ where $\sigma > 0$. If we set a new random variable, $R = \sqrt{X}$, what is the distribution of R? To work this out analytically, we have for $y \geq 0$,

$$F_R(y) = \mathbb{P}(\sqrt{X} \leq y) = \mathbb{P}(X \leq y^2) = F_X(y^2) = 1 - \exp\left(-\frac{y^2}{2\sigma^2}\right),$$

and by differentiating, we get the density,

$$f_R(y) = \frac{y}{\sigma^2} \exp\left(-\frac{y^2}{2\sigma^2}\right).$$

This is the density of the *Rayleigh Distribution* with parameter σ. We see it is related to the exponential distribution via a square root transformation. Hence the implication is that since we know how generate exponential random variables via $-\frac{1}{\lambda}\log(U)$ where $U \sim \text{uniform}(0,1)$, then by applying a square root we can generate Rayleigh random variables.

The Rayleigh distribution is important because of another distributional relationship. Consider two independent normally distributed random variables, N_1 and N_2, each with mean 0 and standard deviation σ. In this case, it turns out that $\tilde{R} = \sqrt{N_1^2 + N_2^2}$ is Rayleigh distributed just as R above. As we see in the next example this property yields a method for generating normal random variables. It also yields a statistical model often used in radio communications called *Rayleigh fading*.

Listing 3.28 demonstrates three alternative ways of generating Rayleigh random variables. It generates R and $\tilde{R}$ as above, as well as by applying `rand()` to a `Rayleigh` object from the `Distributions` package. The mean of a Rayleigh random variable is $\sigma\sqrt{\pi/2}$ and is approximately 2.1306 when $\sigma = 1.7$, as in the code below.

Listing 3.28: Alternative representations of Rayleigh random variables

```
using Distributions, Random
Random.seed!(1)

N = 10^6
sig = 1.7

data1 = sqrt.(-(2* sig^2)*log.(rand(N)))

distG = Normal(0,sig)
data2 = sqrt.(rand(distG,N).^2 + rand(distG,N).^2)

distR = Rayleigh(sig)
data3 = rand(distR,N)

mean.([data1, data2, data3])
```

```
3-element Array{Float64,1}:
 2.1309969895700465
 2.1304634508886053
 2.1292020616665392
```

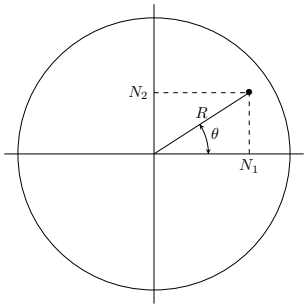

Figure 3.20: Geometry of the Box-Muller transform.

Line 7 generates `data1`, according to R above. Note the use of element wise mapping of `sqrt()` and `log()`. Lines 9 and 10 generate `data2`, as in $\tilde{R}$ above. Here we use `rand()` applied to the normal distribution object `distG`. Lines 12 and 13 use `rand()` applied to a `Rayleigh` distribution object. Line 15 produces the output by applying `mean()` to `data1`, `data2` and `data3` individually. Observe that the sample mean is very similar to the theoretical mean presented above.

A common way to generate normal random variables, called the *Box-Muller Transform*, is to use the relationship between the Rayleigh distribution and a pair of independent zero mean normal random variables, as mentioned above. Consider Figure 3.20 representing the relationship between the pair (N_1, N_2) and their *polar coordinate* counterparts, R and θ. Assume now that the Cartesian coordinates of the point (N_1, N_2) are identically normally distributed, with N_1 independent of N_2 and set $\sigma = 1$. In this case, by representing N_1 and N_2 in polar coordinates (θ, R) we have that the angle θ is uniformly distributed on $[0, 2\pi]$ and that the radius R is distributed as a Rayleigh random variable.

Given this, a recipe for generating N_1 and N_2 is to first generate θ and R and then transform them via,

$$N_1 = R\cos(\theta), \qquad N_2 = R\sin(\theta).$$

Often, N_2 is not needed. Hence in practice, given two independent uniform(0,1) random variables U_1 and U_2 we set, $Z = \sqrt{-2\ln U_1}\cos(2\pi U_2)$. Here Z has a standard Normal distribution. Listing 3.29 uses this method to generate normal random variables and compares their histogram to the standard normal PDF. The output is presented in Figure 3.21.

Listing 3.29: The Box-Muller transform

```
1  using Random, Distributions, Plots; pyplot()
2  Random.seed!(1)
3
4  Z() = sqrt(-2*log(rand()))*cos(2*pi*rand())
5  xGrid = -4:0.01:4
6
7  histogram([Z() for _ in 1:10^6], bins=50,
8                  normed=true, label="MC estimate")
9  plot!(xGrid, pdf.(Normal(),xGrid),
10           c=:red, lw=4, label="PDF",
11           xlims=(-4,4), ylims=(0,0.5), xlabel="x", ylabel="f(x)")
```

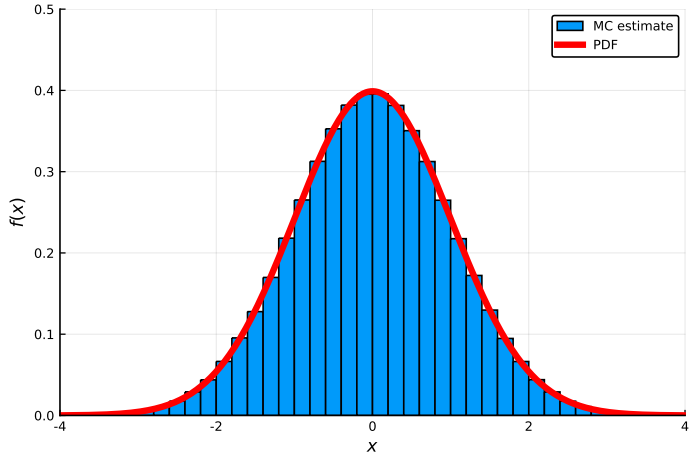

Figure 3.21: The Box-Muller transform can be used to generate a normally distributed random variable.

In line 4 we define a function `Z()`, which implements the Box-Muller transform and generates a single standard normal random variable. The remaining lines plot a histogram of 10^6 random variables from `Z()`, and compares the standard normal PDF. Notice the use of the `L` macro for latex formatting in line 11.

Cauchy Distribution

We now introduce the *Cauchy distribution*, also known as the *Lorentz distribution*. At first glance a plot of the PDF looks very similar to the normal distribution. However, it is fundamentally different as its mean and standard deviation are undefined. The PDF of the Cauchy distribution is given by,

$$f(x) = \frac{1}{\pi\gamma\left(1 + \left(\frac{x - x_0}{\gamma}\right)^2\right)}, \tag{3.23}$$

where x_0 is the location parameter at which the peak is observed and γ is the scale parameter.

In order to better understand the context of this type of distribution we will develop a physical example of a Cauchy distributed random variable. Consider a drone hovering stationary in the sky at unit height. A pivoting laser is attached to its undercarriage, which pivots back and forth as it shoots pulses at the ground. At any point the laser fires, it makes an angle θ from the vertical ($-\pi/2 \leq \theta \leq \pi/2$) as is illustrated in Figure 3.22.

Since the laser fires at a high frequency as it is pivoting, we can assume that the angle θ is distributed uniformly on $[-\pi/2, \pi/2]$. For each shot from the laser, a point can be measured, X, horizontally on the ground from the point above which the drone is hovering. We can now consider

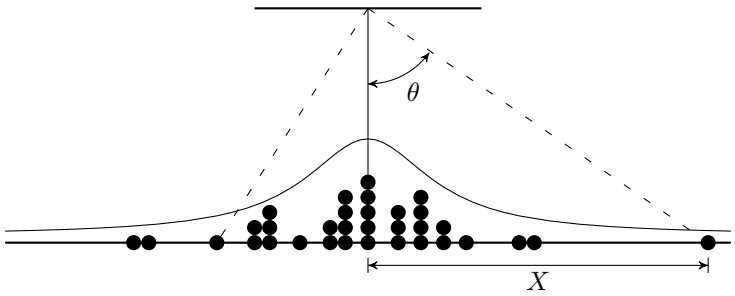

Figure 3.22: Indicative distribution of laser shots from a hovering drone.

this horizontal measurement as a new random variable, X. Hence the CDF is,

$$F_X(x) = \mathbb{P}\big(\tan(\theta) \leqslant x\big) = \mathbb{P}\big(\theta \leqslant \operatorname{atan}(x)\big) = F_\theta\big(\operatorname{atan}(x)\big) = \begin{cases} 0, & \operatorname{atan}(x) \leq -\pi/2, \\ \frac{1}{\pi}\operatorname{atan}(x), & \operatorname{atan}(x) \in (-\pi/2, \pi/2), \\ 1, & \pi/2 \leq \operatorname{atan}(x). \end{cases}$$

Now since it always holds that $\operatorname{atan}(x) \in (-\pi/2, \pi/2)$ we can obtain the density by taking the derivative of $\frac{1}{\pi}\operatorname{atan}(x)$ which evaluates to,

$$f(x) = \frac{1}{\pi(1+x^2)}.$$

This is a special case of the more complicated density (3.23), with $x_0 = 0$ and $\gamma = 1$. Importantly, the expectation integral,

$$\int_{-\infty}^{\infty} x f(x)\, dx,$$

is not defined since each of the one sided improper integrals does not converge. Hence a Cauchy random variable is an example of a *distribution without a mean*.

You may now revisit the law of large numbers (Section 3.2) and ask what happens to sample averages of such random variables. That is, would the sequence of sample averages converge to anything? The answer is no. We illustrate this in Listing 3.30 and the associated Figure 3.23. In this example, occasional large values create huge spikes due to angles near $-\pi/2$ or $\pi/2$. There is no strong law of large numbers in this case since the mean is not defined.

Listing 3.30: The law of large numbers breaks down with very heavy tails

```
using Random, Plots; pyplot()
Random.seed!(808)

n = 10^6
data = tan.(rand(n)*pi .- pi/2)
averages = accumulate(+,data)./collect(1:n)

plot( 1:n, averages,
        c=:blue, legend=:none,
        xscale=:log10, xlims=(1,n), xlabel="n", ylabel="Running average")
```

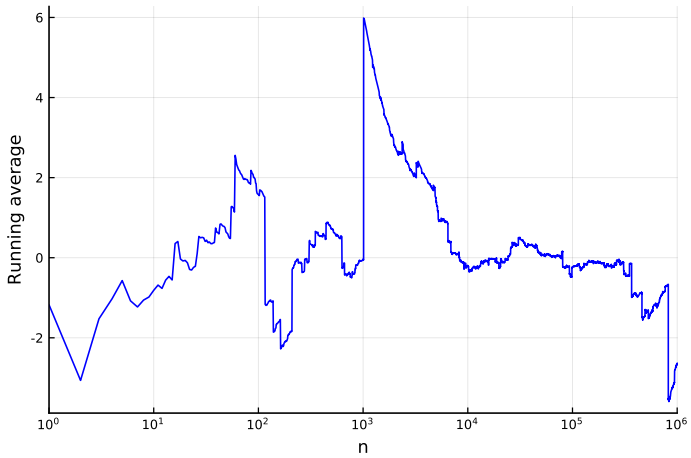

Figure 3.23: Cumulative average of Cauchy distributed random variables.

In line 2 the seed of the random number generator is set, so that the same stream of random numbers is generated each time. In line 5 we create `data`, an array of `n` Cauchy random variables constructed through the angle mechanism described and illustrated in Figure 3.22. In line 6 we use the `accumulate()` function to create a running sum, and then divide this element wise via `./` by the array `collect(1:n)`. Notice that '+' is used as the first argument to `accumulate()`. Here the addition operator is treated as a function. The remainder of the code plots the running average.

3.7 Joint Distributions and Covariance

We now consider pairs and vectors of random variables. In general, in a probability space, we may define multiple random variables, $X_1, \ldots, X_n$ where we consider the vector or tuple, $\mathbf{X} = (X_1, \ldots, X_n)$ as a *random vector*. A key question deals with representing and evaluating probabilities of the form $\mathbb{P}\big(\mathbf{X} \in B\big)$, where B is some subset $\mathbb{R}^n$. Our focus here is on the case of a pair of random variables (X, Y), which are continuous and have a density function. The probability distribution of (X, Y) is called a *bivariate distribution* and more generally, the probability distribution of $\mathbf{X}$ is called a *multi-variate distribution*.

The Joint PDF

A function, $f_{\mathbf{X}} : \mathbb{R}^n \to \mathbb{R}$ is said to be a *joint probability density function* (PDF) if for any input, $x_1, \ldots, x_n$, it holds that $f_{\mathbf{X}}(x_1, x_2, \ldots, x_n) \geq 0$ and,

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} f_{\mathbf{X}}(x_1, x_2, \ldots, x_n)\, dx_1 dx_2 \ldots dx_n = 1. \tag{3.24}$$

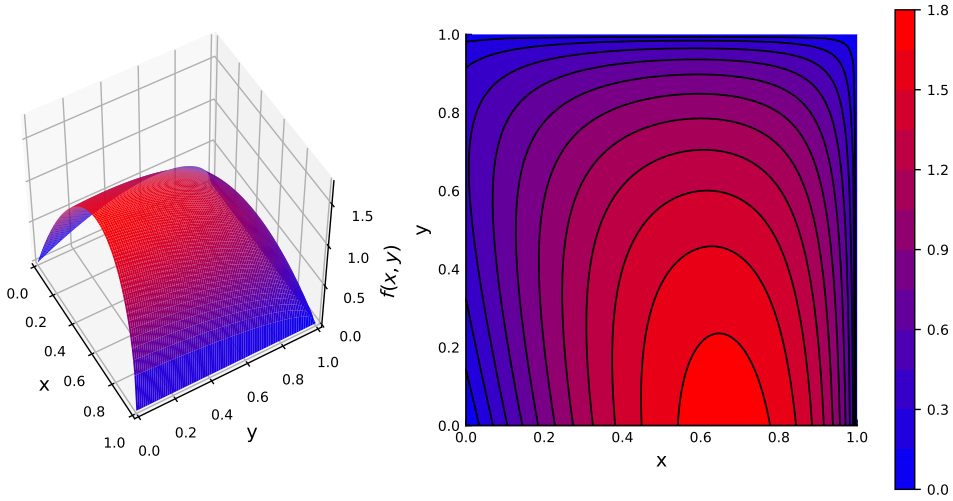

Figure 3.24: A contour plot and a three dimensional surface plot of $f(x, y)$.

Hence if we consider now $B \subset \mathbb{R}^n$, then the probabilities of a random vector $\mathbf{X}$, distributed with density $f_{\mathbf{X}}$, can be evaluated via,

$$\mathbb{P}\Big(\mathbf{X} \in B\Big) = \int_B f_{\mathbf{X}}(\mathbf{x})\, d\,\mathbf{x}.$$

As an example let $\mathbf{X} = (X, Y)$ and consider the joint density,

$$f(x, y) = \begin{cases} \frac{9}{8}(4x + y)\sqrt{(1 - x)(1 - y)}, & x \in [0, 1],\ y \in [0, 1], \\ 0, & \text{otherwise.} \end{cases}$$

This PDF is plotted in Figure 3.24. We may now obtain all kinds of probabilities. For example, set $B = \{(x, y) \mid x < y\}$, then,

$$\mathbb{P}\Big((x, y) \in B\Big) = \int_{x=0}^{1} \int_{y=x}^{1} f(x, y)\, dy\, dx = \frac{31}{80} = 0.3875. \tag{3.25}$$

The joint distribution of X and Y allows us to also obtain related distributions. We may obtain the *marginal densities* of X and Y, denoted $f_X(\cdot)$ and $f_Y(\cdot)$, via,

$$f_X(x) = \int_{y=0}^{1} f(x, y)\, dy \qquad \text{and} \qquad f_Y(y) = \int_{x=0}^{1} f(x, y)\, dx.$$

For our example by explicitly integrating we obtain,

$$f_X(x) = \frac{3}{10}\sqrt{1 - x}(1 + 10x) \qquad \text{and} \qquad f_Y(y) = \frac{3}{20}\sqrt{1 - y}(8 + 5y).$$

In general, the random variables X and Y are said to be *independent* if, $f(x, y) = f_X(x) f_Y(y)$. In our current example, this is not the case. Furthermore, whenever we have two densities of scalar random variables, we may multiply them to make the joint distribution of the random vector composed of independent random variables. That is, if we take our $f_X(\cdot)$ and $f_Y(\cdot)$ above, we may create $\tilde{f}(x, y)$ via,

$$\tilde{f}(x, y) = f_X(x) f_Y(y) = \frac{9}{200}\sqrt{(1 - x)(1 - y)}(1 + 10x)(8 + 5y).$$

Observe that $\tilde{f}(x,y) \neq f(x,y)$. Hence we see that while both bivariate distributions have the same marginal distribution, they are different bivariate distributions and hence describe different relationships between X and Y.

Of further interest is the *conditional density* of X given Y, and vice-versa. It is denoted by $f_{X\,|\,Y=y}(x)$ and describes the distribution of the random variable X, given the specific value $Y = y$. It can be obtained from the joint density via,

$$f_{X\,|\,Y=y}(x) = \frac{f(x,y)}{f_Y(y)} = \frac{f(x,y)}{\int_{x=0}^{1} f(x,y)\,dx}.$$

In Listing 3.31 we generate Figure 3.24, and in addition use crude Riemann sums to approximate the integral (3.25) as well as the integral over the total density.

Listing 3.31: Visualizing a bivariate density

```
1  using Plots, LaTeXStrings, Measures; pyplot()
2
3  delta = 0.01
4  grid = 0:delta:1
5  f(x,y) = 9/8*(4x+y)*sqrt((1-x)*(1-y))
6  z = [f(x,y) for y in grid, x in grid]
7
8  densityIntegral = sum(z)*delta^2
9  println("2-dimensional Riemann sum over density: ", densityIntegral)
10
11 probB = sum([sum([f(x,y)*delta for y in x:delta:1])*delta for x in grid])
12 println("2-dimensional Riemann sum to evaluate probability: ", probB)
13
14 p1 = surface(grid, grid, z,
15         c=cgrad([:blue, :red]), la=1, camera=(60,50),
16         ylabel="y", zlabel=L"f(x,y)", legend=:none)
17 p2 = contourf(grid, grid, z,
18         c=cgrad([:blue, :red]))
19 p2 = contour!(grid, grid, z,
20         c=:black, xlims=(0,1), ylims=(0,1), ylabel="y", ratio=:equal)
21
22 plot(p1, p2, size=(800, 400), xlabel="x", margin=5mm)
```

```
2-dimensional Riemann sum over density: 1.0063787264382458
2-dimensional Riemann sum to evaluate probability: 0.3932640388868346
```

In line 5 we define the bivariate density function, `f()`. In line 6 we evaluate the density over a grid of x and y values. This grid is then used to obtain a crude approximation of the integral in line 8 with the result printed in line 9. Similarly, the nested integral (3.25) is approximated via two Riemann sums in line 11 with the result printed in line 12. The remainder of the code creates Figure 3.24.

Covariance and Vectorized Moments

Given two random variables, X and Y, with respective means, μ_X and μ_Y, the *covariance* is defined by,

$$\mathrm{Cov}(X,Y) = \mathbb{E}\big[(X-\mu_X)(Y-\mu_Y)\big] = \mathbb{E}[X\,Y] - \mu_x\mu_y.$$

The second formula follows by expansion. Notice also that $\mathrm{Cov}(X,X) = \mathrm{Var}(X)$ by comparing with (3.3). The covariance is a common measure of the relationship between the two random variables, where if $\mathrm{Cov}(X,Y) = 0$, we say the random variables are *uncorrelated*. Furthermore, if $\mathrm{Cov}(X,Y) \neq 0$, the its sign gives an indication of the relationship.

Another important concept is the *correlation coefficient*,

$$\rho_{XY} = \frac{\mathrm{Cov}(X,Y)}{\sqrt{\mathrm{Var}(X)\mathrm{Var}(Y)}}. \tag{3.26}$$

It is a normalized form of the covariance with $-1 \leq \rho_{XY} \leq 1$. Values nearing ± 1 indicate a very strong *linear relationship* between X and Y, whereas values near or at 0 indicate a lack of a linear relationship.

Note that if X and Y are independent random variables, then $\mathrm{Cov}(X,Y) = 0$ and hence $\rho_{XY} = 0$. However, the opposite case does not always hold, since in general $\rho_{XY} = 0$ does not imply independence. Nevertheless as described below, for jointly normal random variables it does.

Consider now a random vector $\mathbf{X} = (X_1, \ldots, X_n)$, taken as a column vector. It can be described by moments in an analogous manner to a scalar random variable as was detailed in Section 3.2. A key quantity is the *mean vector*,

$$\mu_X := \big[\mathbb{E}[X_1], \mathbb{E}[X_2], \ldots, \mathbb{E}[X_n]\big]^T.$$

Furthermore, the *covariance matrix* is the matrix defined by the expectation (taken element wise) of the (*outer product*) random matrix given by $(X-\mu_X)(X-\mu_X)^T$, and is expressed as

$$\Sigma_X = \mathrm{Cov}(X) = \mathbb{E}[(X-\mu_x)(X-\mu_x)^T]. \tag{3.27}$$

As can be verified, the i,j'th element of Σ_X is $\mathrm{Cov}(X_i, X_j)$ and hence the diagonal elements are the variances.

Linear Combinations and Transformations

We now consider *linear transformations* applied to random vectors. For any collection of random variables,

$$\mathbb{E}[X_1 + \ldots + X_n] = \mathbb{E}[X_1] + \ldots + \mathbb{E}[X_n].$$

For uncorrelated random variables,

$$\mathrm{Var}(X_1 + \ldots + X_n) = \mathrm{Var}(X_1) + \ldots + \mathrm{Var}(X_n).$$

More generally if we allow the random variables to be correlated, then,

$$\mathrm{Var}(X_1 + \ldots + X_n) = \mathrm{Var}(X_1) + \ldots + \mathrm{Var}(X_n) + 2\sum_{i<j} \mathrm{Cov}(X_i, X_j). \tag{3.28}$$

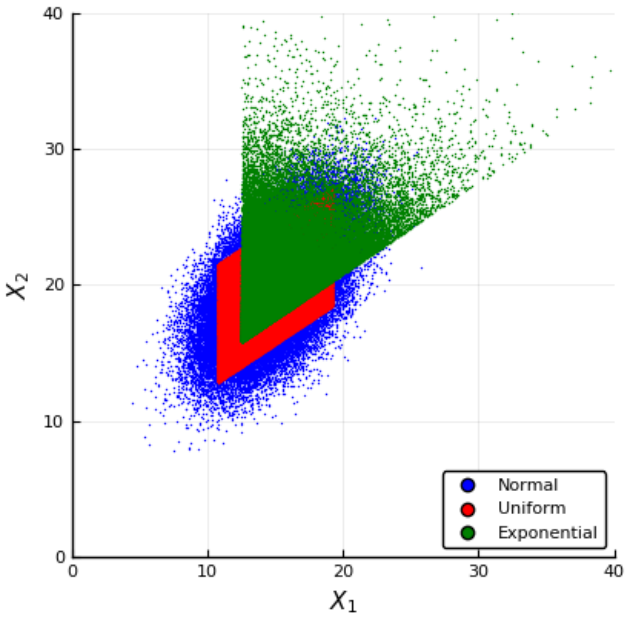

Figure 3.25: Random vectors from three different distributions, each sharing the same mean and covariance matrix.

Note that the right hand side of (3.28) is the sum of the elements of the matrix $\mathrm{Cov}\big((X_1,\ldots,X_n)\big)$. This is a special case of a more general *affine transformation*, where we take a random vector $\mathbf{X} = (X_1,\ldots,X_n)$ with covariance matrix $\Sigma_{\mathbf{X}}$, and an $m \times n$ matrix A and m vector $\mathbf{b}$. We then set,

$$\mathbf{Y} = A\,\mathbf{X} + \mathbf{b}. \tag{3.29}$$

In this case, the new random vector $\mathbf{Y}$ exhibits mean and covariance,

$$\mathbb{E}[\mathbf{Y}] = A\,\mathbb{E}[\mathbf{X}] + \mathbf{b} \qquad \text{and} \qquad \mathrm{Cov}(\mathbf{Y}) = A\Sigma_{\mathbf{X}}A^T. \tag{3.30}$$

Now to retrieve (3.28), we use the $1 \times n$ matrix $A = [1,\ldots,1]$ and observe that $A\Sigma_{\mathbf{X}}A^T$ is a sum of all of the elements of $\Sigma_{\mathbf{X}}$.

The Cholesky Decomposition and Generating Random Vectors

Say now that you wish to create an n dimensional random vector $\mathbf{Y}$ with some specified mean vector $\mu_{\mathbf{Y}}$ and covariance matrix $\Sigma_{\mathbf{Y}}$. That is, $\mu_{\mathbf{Y}}$ and $\Sigma_{\mathbf{Y}}$ are known.

The formulas in (3.30) yield a potential recipe for such a task if we are given a random vector $\mathbf{X}$ with zero mean and identity covariance matrix ($\Sigma_{\mathbf{X}} = I$). For example, in the context of Monte Carlo random variable generation, creating such a random vector $\mathbf{X}$ is trivial – just generate a sequence of n i.i.d. normal(0,1) random variables.

Now apply the affine transformation (3.29) on $\mathbf{X}$ with $\mathbf{b} = \mu_{\mathbf{Y}}$ and a matrix A that satisfies,

$$\Sigma_{\mathbf{Y}} = AA^T. \tag{3.31}$$

Now (3.30) guarantees that $\mathbf{Y}$ has the desired $\mu_{\mathbf{Y}}$ and $\Sigma_{\mathbf{Y}}$.

The question is now how to find a matrix A that satisfies (3.31). For this the *Cholesky decomposition* comes as an aid. As an example assume we wish to generate a random vector $\mathbf{Y}$ with,

$$\mu_{\mathbf{Y}} = \begin{bmatrix} 15 \\ 20 \end{bmatrix} \qquad \text{and} \qquad \Sigma_{\mathbf{Y}} = \begin{bmatrix} 6 & 4 \\ 4 & 9 \end{bmatrix}.$$

Listing 3.32 generates random vectors with these mean vector and covariance matrix using three alternative forms of zero-mean, identity-covariance matrix random variables. As you can see from Figure 3.25, such distributions can be very different in nature even though they share the same first and second order characteristics. The output also presents mean and variance estimates of the random variables generated, showing they agree with the specifications above.

Listing 3.32: Generating random vectors with desired mean and covariance

```
using Distributions, LinearAlgebra, LaTeXStrings, Random, Plots; pyplot()
Random.seed!(1)

N = 10^5

SigY = [ 6 4 ;
         4 9]
muY = [15 ;
       20]
A = cholesky(SigY).L

rngGens = [()->rand(Normal()),
           ()->rand(Uniform(-sqrt(3),sqrt(3))),
           ()->rand(Exponential())-1]

rv(rg) = A*[rg(),rg()] + muY

data = [[rv(r) for _ in 1:N] for r in rngGens]

stats(data) = begin
    data1, data2 = first.(data),last.(data)
    println(round(mean(data1),digits=2), "\t",round(mean(data2),digits=2),"\t",
            round(var(data1),digits=2), "\t", round(var(data2),digits=2), "\t",
            round(cov(data1,data2),digits=2))
end

println("Mean1\tMean2\tVar1\tVar2\tCov")
for d in data
    stats(d)
end

scatter(first.(data[1]), last.(data[1]), c=:blue, ms=1, msw=0, label="Normal")
scatter!(first.(data[2]), last.(data[2]), c=:red, ms=1, msw=0, label="Uniform")
scatter!(first.(data[3]), last.(data[3]), c=:green, ms=1, msw=0, label="Exponential",
        xlims=(0,40), ylims=(0,40), legend=:bottomright, ratio=:equal,
    xlabel=L"X_1", ylabel=L"X_2")
```

```
Mean1   Mean2   Var1    Var2    Cov
14.99   19.99   6.01    9.0     4.0
15.0    20.0    6.01    8.96    3.97
15.0    19.98   6.03    8.85    4.01
```

We define the covariance matrix `SigY` and the mean vector `muY` in lines 6-9. In line 10 we use `cholesky()` from `LinearAlgebra` together with `.L` to compute a lower triangular matrix A that satisfies (3.31). In lines 12-14 we define an array of functions, `rngGens`, where each element is a function that generates a scalar random variable with zero mean and unit variance. The first entry is a standard normal, the second entry is a uniform on $[-\sqrt{3}, \sqrt{3}]$ and the third entry is a unit exponential shifted by -1. The function we define in line 16, `rv()`, assumes an input argument which is a function to generate a random value and then implements the transformation (3.29). In line 18 we create an array of 3 arrays, with each internal array consisting of `N` 2-dimensional random vectors. We then define a function `stats()` in lines 20-25 which calculates and prints first and second order statistics. Note the use of `begin` and `end` to define the function. The function is then used in lines 27-30 for printing output. The remainder of the code creates Figure 3.25 using `data`.

Bivariate Normal

One of the most ubiquitous families of multi-variate distributions is the *multi-variate normal distribution*. Similarly to the fact that a scalar (*univariate*) normal distribution is parametrized by the mean μ and the variance σ^2, a multi-variate normal distribution is parametrized by the mean vector $\mu_{\mathbf{X}}$ and the covariance matrix $\Sigma_{\mathbf{X}}$.

We begin first with the *standard multi-variate* having $\mu_{\mathbf{X}} = \mathbf{0}$ mean and $\Sigma_{\mathbf{X}} = I$. In this case, the PDF for the random vector $\mathbf{X} = (X_1, \ldots, X_n)$ is,

$$f(\mathbf{x}) = (2\pi)^{-n/2} e^{-\frac{1}{2}\mathbf{x}^T\mathbf{x}}. \tag{3.32}$$

Listing 3.33 illustrates numerically that this is a valid PDF for increasing dimensions. The example also illustrates how to use numerical integration. The integral (3.24) is carried out. As is observed from the output, the integral is accurate for dimensions $n = 1, \ldots, 8$ after which accuracy is lost for the given level of computational effort specified (up to 10^7 function evaluations).

Listing 3.33: Multidimensional integration

```
1  using HCubature
2
3  M = 4.5
4  maxD = 10
5
6  f(x) = (2*pi)^(-length(x)/2) * exp(-(1/2)*x'x)
7
8  for n in 1:maxD
9      a = -M*ones(n)
10     b = M*ones(n)
11     I,e = hcubature(f, a, b, maxevals = 10^7)
12     println("n = $(n), integral = $(I), error (estimate) = $(e)")
13 end
```

```
n = 1, integral = 0.9999932046537506, error (estimate) = 4.365848932375016e-10
n = 2, integral = 0.9999864091389514, error (estimate) = 1.487907641465839e-8
n = 3, integral = 0.9999796140804286, error (estimate) = 1.4899542976517278e-8
n = 4, integral = 0.9999728074508313, error (estimate) = 4.4447365681340567e-7
n = 5, integral = 0.999965936103044, error (estimate) = 2.3294669134930872e-5
```

```
n = 6, integral = 0.9999639124757695, error (estimate) = 0.0003937954462609516
n = 7, integral = 1.0001623151630603, error (estimate) = 0.0031506650163379375
n = 8, integral = 1.0074827348433588, error (estimate) = 0.023275741664597824
n = 9, integral = 1.2233043761463287, error (estimate) = 0.3731125349186617
n = 10, integral = 0.42866209316161175, error (estimate) = 0.22089760603668285
```

We use the `HCubature` package. In line 3 we define `M`. Then the integration is performed over a square of width twice of `M`, centered at the origin. In line 4 we define `maxD` as the number of dimensions up to which we wish to carry out integration. The function definition in line 6 implements (3.32). We loop over the dimensions in lines 8-13, each time computing the integral in line 11 where we specify `maxevals` as the maximum number of evaluations. The result is a tuple of the integral value and error, which are assigned to `I` and `e` respectively.

Now in general, using an affine transformation like (3.29), it can be shown that for arbitrary $\mu_{\mathbf{X}}$ and $\Sigma_{\mathbf{X}}$ (positive definite),

$$f(\mathbf{x}) = |\Sigma_{\mathbf{X}}|^{-1/2}(2\pi)^{-n/2}e^{-\frac{1}{2}(\mathbf{x}-\mu_{\mathbf{X}})^T\Sigma_{\mathbf{X}}^{-1}(\mathbf{x}-\mu_{\mathbf{X}})},$$

where $|\cdot|$ is the determinant. In the case of $n = 2$, this becomes the *bivariate normal distribution* with a density represented as,

$$f_{XY}(x, y; \sigma_X, \sigma_Y, \mu_X, \mu_Y, \rho) = \frac{1}{2\pi\sigma_X\sigma_Y\sqrt{1-\rho^2}} \times \exp\left\{\frac{-1}{2(1-\rho^2)}\left[\frac{(x-\mu_X)^2}{\sigma_X^2} - \frac{2\rho(x-\mu_X)(y-\mu_Y)}{\sigma_X\sigma_Y} + \frac{(y-\mu_Y)^2}{\sigma_Y^2}\right]\right\}.$$

Here the elements of the mean and covariance matrix are spelled out via,

$$\mu_{\mathbf{X}} = \begin{bmatrix}\mu_X \\ \mu_Y\end{bmatrix} \qquad \text{and} \qquad \Sigma_{\mathbf{Y}} = \begin{bmatrix}\sigma_X^2 & \sigma_X\sigma_Y\rho \\ \sigma_X\sigma_Y\rho & \sigma_Y^2\end{bmatrix}.$$

Note that $\rho \in (-1, 1)$ is the correlation coefficient as defined in (3.26).

In Section 4.2 we fit the five parameters of a bivariate normal to weather data and keep the results as assignment statements to `meanVect` and `covMat` in the file `mvParams.jl`. The example below, illustrates a plot of random vectors generated from a distribution matching these parameters. Here we use the `MvNormal()` constructor from `Distributions` to create a multi-variate normal distribution object. The listing generates Figure 3.26.

Listing 3.34: Bivariate normal data

```
1  using Distributions, Plots; pyplot()
2
3  include("../data/mvParams.jl")
4  biNorm = MvNormal(meanVect,covMat)
5
6  N = 10^3
7  points = rand(MvNormal(meanVect,covMat),N)
8
9  support = 15:0.5:40
10 z = [ pdf(biNorm,[x,y]) for y in support, x in support ]
11
12 p1 = scatter(points[1,:], points[2,:], ms=0.5, c=:black, legend=:none)
13 p1 = contour!(support, support, z,
14                 levels=[0.001, 0.005, 0.02], c=[:blue, :red, :green],
15                 xlims=(15,40), ylims=(15,40), ratio=:equal, legend=:none,
16                 xlabel="x", ylabel="y")
17 p2 = surface(support, support, z, lw=0.1, c=cgrad([:blue, :red]),
18                  legend=:none, xlabel="x", ylabel="y",camera=(-35,20))
19
20 plot(p1, p2, size=(800, 400))
```

In line 3 we include another Julia file defining `meanVect` and `covMat`. This file is generated in Listing 4.12 of Chapter 4. In line 4 we create an `MvNormal` distribution object representing the bivariate distribution. In line 7 we use `rand()` with a method provided via the `Distributions` package to generate random points. The rest of the code plots Figure 3.26. Notice the call to `contour()` in lines 13-16, with specified `levels`. In lines 17-18 the parameters supplied via `camera` are horizontal rotation and vertical rotation in degrees.

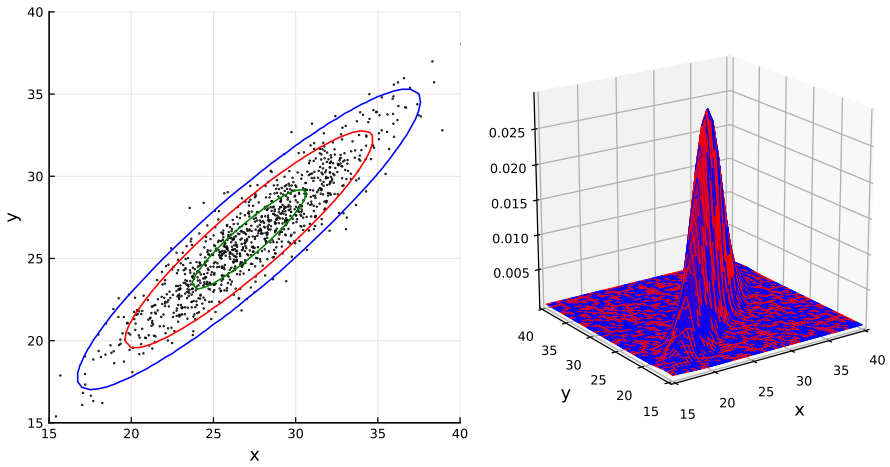

Figure 3.26: Contour lines and a surface plot for a bivariate normal distribution with randomly generated points on the contour plot.

Chapter 4

Processing and Summarizing Data - DRAFT

In this chapter we introduce methods and techniques for processing and summarizing data. In statistics nomenclature, the act of summarizing data is known as *descriptive statistics*. In data-science nomenclature such activities take the names of *analytics* and *dash-boarding*, while the process of manipulating and pre-processing data is sometimes called *data cleansing*, or *data cleaning*.

The statistical techniques and tools that we introduce include summary statistics and methods for data visualization, sometimes called *Exploratory Data Analysis (EDA)*. We introduce several Julia tools for this, including the `DataFrames` package which allows for the storage of datasets that contain non-homogeneous data and includes support for missing entries. We also use the `Statistics` and `StatsBase` packages, which contain useful functions for summarizing data.

In practice statisticians and data-scientists often collect data in various ways, including *experimental studies*, *observational studies*, *longitudinal studies*, *survey sampling*, and *data scraping*. Then to gain insight from the data, one may consider different *data configurations* such as:

Single sample: A case where all observations are considered to represent items from a homogeneous population. The configuration of the data takes the form: $x_1, x_2, \ldots, x_n$.

Single sample over time (*time-series*): The configuration of the data takes the form: $x_{t_1}, x_{t_2}, \ldots, x_{t_n}$ with time points $t_1 < t_2 < \ldots < t_n$.

Two samples: Similar to the single sample case, only now there are two populations (x's and y's). The configuration of the data takes the form: $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$.

Generalizations from two samples to k samples (each of potentially different sample size, $n_1, \ldots, n_k$).

Observations in pairs (2-tuples): In this case, although similar to the two sample case, each observation is a tuple of points, (x, y). Hence the configuration of data is $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.

Generalizations from pairs to vectors of observations. $(x_{11}, \ldots, x_{1p}), \ldots, (x_{n1}, \ldots, x_{np})$.

Other configurations including relationship data (graphs of connections), images, and many more.

This chapter is structured as follows: In Section 4.1 we see how to manipulate tabular data via data frames in Julia. In Section 4.2 we deal with methods of summarizing data including basic elements of descriptive statistics. We then move on to plotting where in Section 4.3 we present a variety of methods for plotting single sample data. In Section 4.4 we present plots for comparing samples. Section 4.5 presents plots for multivariate and high-dimensional data. We then present more simplistic business style plots in Section 4.6. The chapter closes with Section 4.7, where we show several ways of handling files using Julia as well as how to interact with a server side database.

For readers who wish to better understand the concepts of copies and mutability used in Section 4.1, the subsection below provides an optional overview. It can be skipped on a first reading.

Mutability, References, Shallow Copies and Deep Copies in Julia

When using any programming language, it is useful to have a basic understanding of how data is organized and referenced in memory. For this reason we now briefly overview the differences between *mutable* types, *immutable* types, *reference copying*, *shallow copies* and *deep copies* in Julia. We also introduce the basic programming concepts of 'call by value' and 'call by reference'. This basic understanding is important in its own right, however it may also help readers better understand certain aspects of Julia's `DataFrame` package, described in the sequel.

As a starting point, we review the difference between two mechanisms for passing variables to functions. Assume you have a variable `x`, a function `f()`, and then you then execute `f(x)`. One can envision two general mechanisms by which this can take place. The first is named *call by value* and describes a situation where the code implementing `f()` gets a copy of the variable `x`. As `f()` executes, even if its code appears to modify `x`, it is actually modifying a local copy. The second mechanism is named *call by reference* and describes a situation where `f()` obtains a *memory reference* (or *pointer*) to `x`. In such a case, as `f()` executes, if it modifies `x`, then it actually modifies values in the original memory location of `x`.

In Julia, both mechanisms exist under a unified umbrella called *pass by sharing*. This means that variables are not copied when passed to functions. However, if a value is about to be changed within a function then depending on the mutability attribute of its type, either of the mechanisms may be employed. If the variable type is *immutable* then a local copy is made and the behavior follows the 'call by value' type. However, if the type is *mutable* then the called function does not create a local copy. Instead, it can modify the original variable according to the 'call by reference' mechanism. Hence the variable's property, mutable or immutable, determines which function calling mechanism is exhibited.

As a general rule, primitive types such as `Int64` or `Float32` are immutable. The same goes for composite types defined using the `struct` keyword. An exception to this is for composite types that are explicitly defined as `mutable struct`. Note that the code examples in this book seldom define types - however many of the types we use from packages are composite types. While not often used, if you wish to programmatically check if the type of a variable is immutable or not, you can use the `isimmutable()` function.

Importantly, arrays are mutable. Listing 4.1 implements two different methods for the function `f()`. The first method is for `Int`, a primitive type (immutable), and the second is for `Array{Int}`

(mutable). It then demonstrates the 'call by value' behavior exhibited for the primitive type, while the 'call by reference' behavior is exhibited for the array.

Listing 4.1: Call by value vs. call by reference

```
1  f(z::Int) = begin z = 0 end
2  f(z::Array{Int}) = begin z[1] = 0 end
3
4  x = 1
5  @show typeof(x)
6  @show isimmutable(x)
7  println("Before call by value: ", x)
8  f(x)
9  println("After call by value: ", x,"\n")
10
11 x = [1]
12 @show typeof(x)
13 @show isimmutable(x)
14 println("Before call by reference: ", x)
15 f(x)
16 println("After call by reference: ", x)
```

```
typeof(x) = Int64
isimmutable(x) = true
Before call by value: 1
After call by value: 1

typeof(x) = Array{Int64,1}
isimmutable(x) = false
Before call by reference: [1]
After call by reference: [0]
```

In line 1 we implement a method of `f()` for integer types. The code `z = 0` will operate on a local copy of `z`. In line 2 we implement a method of `f()` for arrays. Here the code `z[1] = 0` will modify the first entry of the input argument `z`. Lines 4-9 use the first method, passing the variable `x` into `f()`. As can be see from the output, the operation of the function `f()` does not modify `x`. Also note the use of the `@show` macro, useful for debugging or understanding code. Lines 11-16 invoke the method of `f()` for arrays of integers (this is multiple dispatch). The key point is that `f(x)` in line 15 modifies the original `x` from global scope.

Ideally, for performance reasons, the level of actual copying of memory should be kept to a minimum. This is the underlying motivation for having a default 'pass by reference' mechanism when working with arrays, as you can give functions references to huge data arrays without any memory duplication. However, this entails some level of danger because function calls may modify variables that are passed to them as arguments. For this reason, Julia offers explicit functions for creating copies of variables, namely `copy()` and `deepcopy()`. The former creates a 'shallow copy' of the variable and copies all entries, but does not do it recursively. The latter recursively produces a copy until a completely independent copy of the variable is created.

We demonstrate the different type of copies and their interaction with mutability in Listing 4.2. The basic example on which we apply a deep copy is a doubly nested array, e.g. `[[10]]`. In this case, using `copy()` will not be applied to the inner array `[10]`, however using `deepcopy()` recursively copies all mutable entries.

Listing 4.2: Deep copy and shallow copy

```
1  println("Immutable:")
2  a = 10
3  b = a
4  b = 20
5  @show a
6
7  println("\nNo copy:")
8  a = [10]
9  b = a
10 b[1] = 20
11 @show a
12
13 println("\nCopy:")
14 a = [10]
15 b = copy(a)
16 b[1] = 20
17 @show a
18
19 println("\nShallow copy:")
20 a = [[10]]
21 b = copy(a)
22 b[1][1] = 20
23 @show a
24
25 println("\nDeep copy:")
26 a = [[10]]
27 b = deepcopy(a)
28 b[1][1] = 20
29 @show a;
```

```
Immutable:
a = 10

No copy:
a = [20]

Copy:
a = [10]

Shallow copy:
a = Array{Int64,1}[[20]]

Deep copy:
a = Array{Int64,1}[[10]]
```

Lines 1-5 exhibit no surprise due to immutability. The `Int64` `a` is assigned to `b` and `b` is modified in line 4. At this point Julia creates a copy because the variable is immutable. Lines 7-11 demonstrate different behavior. The array `a` is mutable and hence after `b` is assigned to `a` in line 9, the modification of `b` in line 10 also modifies `a`. Lines 13-17 show a case where a `copy()` of `a` is created. In this case modification of `b` in line 16 does not alter `a`. Lines 19-23 are similar, however in this case the fact that `copy()` is only a shallow copy matters. The variable `b` has a new outer array, however the inner array is still shared with `a`. Hence the modification in line 22 modifies the inner array of `a` as well. Finally, in lines 25-29 this is resolved by creating a `deepcopy()`.

4.1 Working with Data Frames

In cases where data is homogeneous, arrays, matrices, and tensors are popular ways of organizing data. However, more commonly datasets are heterogeneous in nature, or contain incomplete or missing entries. In addition, datasets are often large, and commonly require "cleaning". In such cases, more advanced data storage mechanisms are needed.

The Julia `DataFrames` package introduces a data storage structure known as a `DataFrame`, which is aimed at overcoming these challenges. It can be used to store columns of different types, and also introduces the `missing` variable type which, as the name suggests, is used in place of missing entries. The `missing` type has an important property, in that it 'poisons' other types it interacts with. For example, if `x` represents a value, then `x + missing == missing`. This ensures that missing values do not 'infect' and skew results when operations are performed on data. For example, if `mean()` is used on a column with a `missing` value present, the result will evaluate as `missing`. We show ways of dealing with missing values in Listing 4.7.

Data frames are easy to work with. They can be created manually or data can be imported from a `csv` or `txt` file. Columns and rows can be referenced by their position index, name (i.e. symbol), or according to a set of user-defined rules. We now explore some of their functionally. See `http://juliadata.github.io/DataFrames.jl/stable/` for further documentation.

Introducing the Data Frame

We now introduce data frames through the exploration and formatting of an example dataset. The dataset has four fields; `Name`, `Date`, `Grade` and `Price`. In addition, as is often the case with real datasets, there are missing values present. Therefore, before analysis can start, some *data cleaning* must be performed.

Any variable in a dataset can be classified as either a *numerical variable*, or *categorical variable*. A numerical variable is a variable in which the location of the measurement on the number line is meaningful. Examples include height and weight. A categorical variable communicates some information based on categories, or characteristics via grouping. Categorical variables can be further split into *nominal variables*, such as blood type, or names, and *ordinal variables*, in which some order is communicated, such as grades on a test, A to E. In our example, `Price` is a numerical variable, while `Name` is a nominal categorical variable. Since, in our example, `Grade` can be thought of as a rating (A being best, and E being worst) it is an ordinal categorical variable.

Having covered types of variables, we begin a step by step example of using data frames. In Listing 4.3 we load the data from the file `purchaseData.csv` into a data frame and inspect its contents. Often *comma separated values files* (csv files) contain a *header row* which gives details of each column. However in other cases, no header row appears. In our case, the file's first row is is a header row and it contains the names of the columns. Hence the first row of the file is:

```
Name, Date, Grade, Price
```

Listing 4.3: Creating and inspecting a `DataFrame`

```
1  using DataFrames, CSV
2  data = CSV.read("../data/purchaseData.csv", copycols = true)
3
4  println(size(data),"\n")
5  println(names(data),"\n")
6  println(first(data, 6),"\n")
7  println(describe(data),"\n")
```

```
(200, 4)

Symbol[:Name, :Date, :Grade, :Price]

6×4 DataFrame
| Row | Name     | Date       | Grade   | Price   |
|     | String   | String     | String  | Int64   |
-------------------------------------------------
| 1   | MARYANNA | 14/09/2008 | A       | 79700   |
| 2   | REBECCA  | 11/03/2008 | B       | missing |
| 3   | ASHELY   | 5/08/2008  | E       | 24311   |
| 4   | KHADIJAH | 2/09/2008  | missing | 38904   |
| 5   | TANJA    | 1/12/2008  | C       | 47052   |
| 6   | JUDIE    | 17/05/2008 | D       | 34365   |

4×8 DataFrame
| Row | variable | mean    | min       | median  | max       | nunique | nmissing | eltype
|
|     | Symbol   | Union   | Any       | Union   | Any       | Union   | Int64    | Union
|
-----------------------------------------------------------------------------------------------------------------------
| 1   | Name     |         | ABBEY     |         | ZACHARY   | 182     | 17       | Union{Missing, String} |
| 2   | Date     |         | 1/07/2008 |         | 9/10/2008 | 141     | 4        | Union{Missing, String} |
| 3   | Grade    |         | A         |         | E         | 5       | 13       | Union{Missing, String} |
| 4   | Price    | 39702.0 | 8257      | 38045.5 | 79893     |         | 14       | Union{Missing, Int64}
|
```

In line 1 we specify use of the `DataFrames` package, which allows us to use `DataFrame` type objects. We also use the `CSV` package for reading `csv` files. In line 2 `CSV.read()` is used to create a data frame object, populated with data from the file specified. Note that our file has a header row, however in cases where there isn't a header use `header = false`. We use `copycols = true` to create a data frame with mutable columns (the default is `false`). If the default was used, each column would be of the read-only `CSV.Column` type. In line 4 the `size()` function is used to return the number of rows and columns of the data frame as a tuple. Two other useful functions not shown here are `nrow()` and `ncol()`, which return the number of rows and number of columns respectively. In line 5 the `names()` function is used to return an array of all column names as symbols. In line 6 the `first()` function is used to display the first six lines of the data frame, as specified by the second argument. Note that `last()` can be used to display the last several rows instead. In line 7 `describe()` is used to create a data frame with a summary of the data in each column of the input data frame (`data` in our case). On inspection, by looking at the `nmissing` column, one can see there are missing values present, and we return to this problem in Listing 4.7.

Referencing Data

We now look at ways in which entries within a data frame can be referenced. Individual entries can be referenced by both row and column index. Columns can be referenced by their name represented as a symbol, or by their index. Multiple rows or multiple columns can be referenced via a collection of symbols or indices. We demonstrate several aspects of this in Listing 4.4 below.

Listing 4.4: Referencing data in a `DataFrame`

```
1  using DataFrames, CSV
2  data = CSV.read("../data/purchaseData.csv", copycols = true)
3
4  println("Grade of person 1: ", data[1, 3],
5          ", ", data[1,:Grade],
6          ", ", data.Grade[1], "\n")
7  println(data[[1,2,4], :], "\n")
8  println(data[13:15, :Name], "\n")
9  println(data.Name[13:15], "\n")
10 println(data[13:15, [:Name]])
```

```
Grade of person 1: A, A, A

3×4 DataFrame
| Row | Name     | Date       | Grade   | Price   |
|     | String   | String     | String  | Int64   |
-----------------------------------------------------
| 1   | MARYANNA | 14/09/2008 | A       | 79700   |
| 2   | REBECCA  | 11/03/2008 | B       | missing |
| 3   | KHADIJAH | 2/09/2008  | missing | 38904   |

Union{Missing, String}["SAMMIE", missing, "STACEY"]

Union{Missing, String}["SAMMIE", missing, "STACEY"]

3×1 DataFrame
| Row | Name    |
|     | String  |
-----------------
| 1   | SAMMIE  |
| 2   | missing |
| 3   | STACEY  |
```

In lines 4-6 we see different ways of accessing the element from the first row and third column labeled `:Date`. In line 7 the rows and columns to be extracted are designated by the first and second arguments, `[1,2,4]`, and '`:`' respectively. Note that '`:`' can be used to select either all rows, or all columns. Line 8 is somewhat similar, but here a unit range is used to select rows 13-15, while the symbol `:Name` is used so that only the `Name` column is extracted. Alternatively, the column could have been referenced by its index, i.e. `data[13:15, 1]`, or the syntax `data.Name[13:15]` could have been used instead. Note that although lines 9 and 10 look similar, there is an important difference. The code in line 9 creates an array, while that of line 10 creates a data frame object, due to the extra set of `[]`. If one wanted, additional columns could also be selected by including them in `[]`, separated by '`,`'.

Modifying Data

In general, entries of a data frame can be updated like entries of a matrix, however in certain cases care is required. Functions performed on a data frame will return a copy of that data frame. In other words, no underlying change will be made to the data frame object, but rather a shallow copy will be made and returned as output. Often one wants to change the values within a data frame. However, by default the columns of a data frame are *immutable*, which means that the values within them cannot be changed. In order to make changes to a column, the column must first be mutable. One way to do this is by including `copycols=true` when a data frame is created from a `csv` file. This argument has the effect of making all columns mutable. Another way is by using `!` when referencing the rows of a data frame. For example, `df[!, :X]` references the underlying data in column `:X`, while `df[:, :X]` simply references a shallow copy. In Listing 4.5 below we show how these approaches work.

Listing 4.5: Editing and copying a `DataFrame`

```
using DataFrames, CSV
data1 = CSV.read("../data/purchaseData.csv")
data2 = CSV.read("../data/purchaseData.csv", copycols=true)

try data1[1, :Name] = "YARDEN" catch; @warn "Cannot: data1 is immutable" end

data2[1, :Name] = "YARDEN"
println("\n", first(data2, 3), "\n")

data1[!, :Price] ./= 1000
rename!(data1, :Price=>Symbol("Price(000's)"))
println(first(data1, 3), "\n")

replace!(data2[!,:Grade], ["E"=>"F", "D"=>"E"]...)
println(first(data2, 3),"\n")
```

```
Warning: Cannot: data1 is immutable

| Row | Name    | Date       | Grade   | Price   |
|     | String  | String     | String  | Int64   |
-----------------------------------------------
| 1   | YARDEN  | 14/09/2008 | A       | 79700   |
| 2   | REBECCA | 11/03/2008 | B       | missing |
| 3   | ASHELY  | 5/08/2008  | E       | 24311   |

3×4 DataFrame
| Row | Name     | Date       | Grade   | Price(000's) |
|     | String   | String     | String  | Float64      |
-------------------------------------------------------
| 1   | MARYANNA | 14/09/2008 | A       | 79.7         |
| 2   | REBECCA  | 11/03/2008 | B       | missing      |
| 3   | ASHELY   | 5/08/2008  | E       | 24.311       |

3×4 DataFrame
| Row | Name    | Date       | Grade   | Price   |
|     | String  | String     | String  | Int64   |
-----------------------------------------------
| 1   | YARDEN  | 14/09/2008 | A       | 79700   |
| 2   | REBECCA | 11/03/2008 | B       | missing |
| 3   | ASHELY  | 5/08/2008  | F       | 24311   |
```

In lines 2-3 two dataframes are created, the first `data1` has immutable columns, while `data2` has mutable columns due to the second argument in `CSV.read()`. In line 5 we try to change the value of the first row and first column in `data1`. This is done within the `try/catch` structure. If an error occurs within `try`, the code jumps to `catch` and continues. Since `data1` is immutable, an error is returned, and so the code after `catch` runs. Here we use the `@warn` macro to return a warning. In line 7 we try the same change for `data2`, and since this data frame is mutable, we are able to make the change. In line 10 we perform division on every row element in the `:Price` column of `data1` by using `!` to reference all rows. By using this syntax, the underlying `:Price` column data is referenced, and the column changed to mutable, which then allows us to make make the change. The actual change is done via the combination of the broadcast '`.`' operator, which extends the in-place division via `\=` to each row. Note the column type changes from `Int64` to `Float64`. In line 11 `rename!()` is used to rename the `:Price` column as shown, with a pair of values, separated via `=>`, given as the second argument. Finally, in line 14, `replace!()` is used to replace all `D` and `E` entries in the `:Grade` column to `E` and `F` respectively. Note that `replace!()` operates on an iterable, hence the use of the `...` splat operator, and finally note that the order of replacement does not matter, as the replacement does not advance one after the other sequentially. Again note that `!` was used for row referencing.

Copying a Data Frame

When copying a data frame, the same rules and principles that are relevant for other Julia types apply. These were discussed at the start of this chapter and demonstrated in Listing 4.2. We now show how `copy()` and `deepcopy()` can be used with data frames in Listing 4.6.

Listing 4.6: Using `copy()` and `deepcopy()` with a `DataFrame`

```
1  using DataFrames, CSV
2  data1 = CSV.read("../data/purchaseData.csv", copycols=true)
3  println("Original value: ", data1.Name[1],"\n")
4
5  data2 = data1
6  data2.Name[1] = "EMILY"
7  @show data1.Name[1]
8
9  data1 = CSV.read("../data/purchaseData.csv", copycols=true)
10 data2 = copy(data1)
11 data2.Name[1] = "EMILY"
12 @show data1.Name[1]
13 println()
14
15 data1 = DataFrame()
16 data1.X = [[0,1],[100,101]]
17 data2 = copy(data1)
18 data2.X[1][1] = -1
19 @show data1.X[1][1]
20
21 data1 = DataFrame(X = [[0,1],[100,101]])
22 data2 = deepcopy(data1)
23 data2.X[1][1] = -1
24 @show data1.X[1][1];
```

```
Original value: MARYANNA

data1.Name[1] = "EMILY"
data1.Name[1] = "MARYANNA"

(data1.X[1])[1] = -1
(data1.X[1])[1] = 0
```

We first create a data frame from a csv file where `data1.Name[1]` is the string `MARYANNA`. Then in lines 5-7, setting `data2 = data1` simply implies that `data2` refers to the same object as `data1`. Hence modifying `data2` in line 6 results in a modification of `data1`. In lines 9-13 we circumvent such a situation by using the `copy()` function. In this case setting the new name into `data2`, `EMILY`, does not affect `data1`. However, in other cases a shallow copy isn't enough for separating data frames. This is the case in lines 15-19 where we create a data frame with a column named `X` comprised of arrays. In this case, the copied data frame, `data2`, still refers to the original entries (arrays), because these are mutable and were not copied via `copy()` in line 17. The consequence is that modifying a specific entry of `data2` as in line 18 actually modifies `data1`. This is then circumvented by using `deepcopy()` as in lines 21-24.

Handling Missing Entries

We now look more closely at the case when `missing` values are present in a data frame. As discussed at the start of this section, `missing` 'poisons' other types on interaction, and this property ensures that missing values do not 'infect' and skew results when operations are performed on a dataset. The `DataFrames` package comes with several useful functions for dealing with missing entries. The `Missing.jl` package also provides extra functionality for dealing with missing values. In Listing 4.7 below we elaborate on some of the functions useful for dealing with missing values.

Listing 4.7: Handing `missing` entries

```
1  using Statistics, DataFrames, CSV
2  data = CSV.read("../data/purchaseData.csv", copycols=true)
3
4  println(mean(data.Price),"\n")
5  println(mean(skipmissing(data.Price)),"\n")
6  println(coalesce.(data.Grade, "QQ")[1:4],"\n")
7  println(first(dropmissing(data,:Price), 4),"\n")
8  println(sum(ismissing.(data.Name)),"\n")
9  println(findall(completecases(data))[1:4])
```

```
missing

39702.01075268817

["A", "B", "E", "QQ"]

4×4 DataFrame
| Row | Name      | Date       | Grade   | Price |
|     | String    | String     | String  | Int64 |
-------------------------------------------------
| 1   | MARYANNA | 14/09/2008 | A       | 79700 |
```

```
| 2   | ASHELY   | 5/08/2008  | E       | 24311 |
| 3   | KHADIJAH | 2/09/2008  | missing | 38904 |
| 4   | TANJA    | 1/12/2008  | C       | 47052 |

17

[1, 3, 5, 6]
```

In line 4 we attempt to calcualte the mean of the `:Price` column of `data`, however `missing` is returned as this column contains missing values. By comparison, in line 5 `skipmissing()` is first used to return a copy of the data from the `:Price` column which has no missing entries, and after this `mean()` applied. In line 6 `data.Grade` is used to obtain a reference to the `:Grade` column, and then the `coalesce()` function is used to replace all `missing` values with the string '`QQ`'. The first four values are accessed via `[1:4]` to verify the replacement has occurred. In line 7 `dropmissing()` is used to drop all rows which have `missing` in the `:Price` column. If no second argument is given, `dropmissing()` will drop all rows that contain `missing`. In line 8 `ismissing()` is used with the broadcast operator to check if values in the `:Name` column are `missing`. If they are, `true` is returned, else `false`. Then `sum()` is used to calculate how many `missing` entries are present. The result, `17`, can be verified from the output of Listing 4.3 where we see the number of missing entries. In line 9 `completecases()` is used to check if each row contains fully completed fields, i.e. no `missing` values. If no `missing` values are present, `true` is returned, else `false`, for each row. Then `findall()` is used on this array to return an array of row indexes which have no `missing` values, and to shorten the output, the first four values of this array are printed.

Reshaping, Joining and Manipulating Data Frames

When working with data it is not uncommon to want to perform operations such as merges or joins between several data sets, or to split or reshape the structure of a data set. The `DataFrames` package makes this easy, as it provides many useful functions to do these types of operations. In Listing 4.8 we present brief examples of some of the more useful functions for merging and joining data frames.

Listing 4.8: Reshaping, joining and merging data frames

```
using DataFrames, CSV
data = CSV.read("../data/purchaseData.csv", copycols = true)

newCol = DataFrame(Validated=ones(Int, size(data,1)))
newRow = DataFrame([["JOHN", "JACK"] [123456, 909595]], [:Name, :PhoneNo])
newData = DataFrame(Name=["JOHN", "ASHELY","MARYANNA"],
                    Job=["Lawyer", "Doctor","Lawyer"])

data = hcat(data, newCol)
println(first(data, 3), "\n")

data = vcat(data, newRow, cols=:union)
println(last(data, 3), "\n")

data = join(data, newData, on=:Name)
println(data, "\n")

select!(data,[:Name,:Job])
println(data, "\n")

unique!(data,:Job)
println(data)
```

```
3×5 DataFrame
| Row | Name     | Date       | Grade   | Price   | Validated |
|     | String   | String     | String  | Int64   | Int64     |
-------------------------------------------------------------
| 1   | MARYANNA | 14/09/2008 | A       | 79700   | 1         |
| 2   | REBECCA  | 11/03/2008 | B       | missing | 1         |
| 3   | ASHELY   | 5/08/2008  | E       | 24311   | 1         |

3×6 DataFrame
| Row | Name | Date       | Grade   | Price   | Validated | PhoneNo |
|     | Any  | String     | String  | Int64   | Int64     | Any     |
-----------------------------------------------------------------
| 1   | RIVA | 30/12/2008 | E       | 21842   | 1         | missing |
| 2   | JOHN | missing    | missing | missing | missing   | 123456  |
| 3   | JACK | missing    | missing | missing | missing   | 909595  |

3×7 DataFrame
| Row | Name     | Date       | Grade   | Price   | Validated | PhoneNo | Job    |
|     | Any      | String     | String  | Int64   | Int64     | Any     | String |
-------------------------------------------------------------------------------
| 1   | MARYANNA | 14/09/2008 | A       | 79700   | 1         | missing | Lawyer |
| 2   | ASHELY   | 5/08/2008  | E       | 24311   | 1         | missing | Doctor |
| 3   | JOHN     | missing    | missing | missing | missing   | 123456  | Lawyer |

3×2 DataFrame
| Row | Name     | Job    |
|     | Any      | String |
--------------------------
| 1   | MARYANNA | Lawyer |
| 2   | ASHELY   | Doctor |
| 3   | JOHN     | Lawyer |

2×2 DataFrame
| Row | Name     | Job    |
```

```
|     | Any      | String |
-------------------------
| 1   | MARYANNA | Lawyer |
| 2   | ASHELY   | Doctor |
```

In line 2 we create `data` in the same manner as the previous listings. In lines 4-6 we create three separate data frames. The first, `newCol`, consists of a single column `:Validated` with the same number of rows as `data`. The second, `newRow`, consists of two rows with `:Name` and `:PhoneNumber` columns. The third, `newData`, has two rows and two columns, `:Name` and `:Job`. In line 9 `hcat()` is used to horizontally concatenate `newCol` to `data`. In line 12 `vcat()` is used to vertically concatenate `data` and `newRow`, with the new row appended to the bottom of the data frame. Note `cols=:union` is used so that all columns from both data frames are kept, and `missing` entries recorded where applicable. Alternatively, `:equal` or `:intersect` could have been used, or an array of columns to be kept instead. In line 15 `join()` is used to join `data` and `newData` together, based on the `:Name` column. Note that `join()` can be used in several different ways. The functions `select!()` and `unique!()` are demonstrated in lines 18-22. Another function not shown here is `stack()`, which can be used to stack a data frame from a wide format to a long format. We recommend the reader consult the `DataFrames` manual for further information on each of the functions listed here.

Useful Operations for Data Frames

We have already covered some of the many useful functions available in the `DataFrames` package, such as `replace!()` and `rename!()`, both introduced in Listing 4.5. We now provide insight into several more concepts, including sorting, changing a column of strings to `Date` types, how to make a column `Categorical` (useful when constructing models, as covered in 8.4), and finally how to split, apply, and combine data all in one via the `by()` function. Listing 4.9 demonstrates these.

Listing 4.9: Manipulating `DataFrame` objects

```
1  using DataFrames, CSV, Dates, Statistics
2  data = dropmissing(CSV.read("../data/purchaseData.csv", copycols=true))
3
4  data[!,:Date] = Date.(data[!,:Date], "d/m/y")
5  println(first(sort(data, :Date), 3),"\n")
6
7  println(first(filter(row -> row[:Price] > 50000, data),3 ),"\n")
8
9  categorical!(data, :Grade)
10 println(first(data, 3), "\n")
11
12 println(
13    by(data, :Grade, :Price =>
14        x -> ( NumSold=length(x), AvgPrice=mean(x)) )
15    )
```

```
3×4 DataFrame
| Row | Name      | Date       | Grade  | Price |
|     | String    | Date       | String | Int64 |
-------------------------------------------------
| 1   | STEPHEN   | 2008-02-11 | D      | 33155 |
| 2   | JACKELINE | 2008-02-12 | E      | 8257  |
| 3   | ARDELL    | 2008-03-03 | C      | 46911 |

3×4 DataFrame
| Row | Name     | Date       | Grade  | Price |
|     | String   | Date       | String | Int64 |
------------------------------------------------
| 1   | MARYANNA | 2008-09-14 | A      | 79700 |
| 2   | NOE      | 2008-08-15 | A      | 79344 |
| 3   | SAMMIE   | 2008-11-05 | B      | 61730 |

3×4 DataFrame
| Row | Name     | Date       | Grade        | Price |
|     | String   | Date       | Categorical  | Int64 |
-----------------------------------------------------
| 1   | MARYANNA | 2008-09-14 | A            | 79700 |
| 2   | ASHELY   | 2008-08-05 | E            | 24311 |
| 3   | TANJA    | 2008-12-01 | C            | 47052 |

5×3 DataFrame
| Row | Grade        | NumSold | AvgPrice |
|     | Categorical  | Int64   | Float64  |
-----------------------------------------------
| 1   | A            | 15      | 76606.7  |
| 2   | B            | 19      | 59873.9  |
| 3   | C            | 33      | 45285.8  |
| 4   | D            | 35      | 34656.8  |
| 5   | E            | 51      | 20492.5  |
```

In line 2 `dropmissing()` is used so that all rows with missing entries are excluded. This is done as some of the functions here require all values to be non-missing. In line 4 the `Date()` function from the `Dates` package is applied to every row from the `:Date` column, converting each entry from a string to a `Date` type, according to the string formatting given as the second argument. In line 5 `sort()` is used to sort by the `:Date` column. In line 7 `filter()` is used to return only rows which have a `:Price` greater than `50000`. In line 9 the type of the `:Grade` column is changed to categorical via `categorical!()`. In lines 13-14 the powerful `by()` function is demonstrated. Here `data` is split according to `:Grade`. The third argument is where calculations are defined. The columns to be referenced in the calculations are put to the left of '=>', in our case only `:Price` is used. The calculations are specified by the anonymous function in line 14. Note that => is used to define a `Pair` and -> is used to define an anonymous function. The anonymous function creates a `NamedTuple` defining two new columns, `:NumSold` and `:AvgPrice`. For the first, the total number of each `:Grade` is calculated based on the length, i.e. number of entries in the price column. For the second, the average price is calculated via `mean()`. Note that the `by()` function can be used in many ways, and calculations can be done over data in more than one column. There are also several other related functions not touched on here, including `mapcols()`, which can be used to transform all values in a data frame, and `aggregate()`, which has functionality similar to `by()`. Further functionality is also available via the `DataFramesMeta` package which provides a macro-based framework to interface with data frames, such as via the `@linq` macro and the `|>` operator. Consult the documentation for further information.

A Cleaning and Imputation Example

In practice, it is not uncommon for data sets to contain many missing values, or require a certain amount of cleaning before one can use the data, see for example [TKNS20]. Furthermore it may not always be practical to simply exclude every observation which has a missing value. For example, if we were to simply exclude all rows with missing values in `purchaseData.csv`, then we would loose almost 25% of the data set.

Instead of simply deleting rows, one way of dealing with `missing` values is to use *imputation*. This involves substituting missing entries with values based on either the data observed, or according to some other logic. Various methods can be used to impute missing values, and care must be taken when imputing, as it can lead to bias in the data. The exact type of imputation scheme used should ideally take into account both the nature of the data and the eventual statistical analysis. A comprehnsive treatment is in [Van12].

We now present an example of one way in which one might consider cleaning and imputing a data set. In Listing 4.10 below we clean the data and impute missing values. First, we replace all missing names with the string '`QQ`' and all missing dates with the string '`31/06/2008`'. We then calculate the average price of each grade based on the data available, and use these averages to impute missing entries in both the price and grade columns.

Listing 4.10: Cleaning and imputing data

```
using DataFrames, CSV, Statistics
data = CSV.read("../data/purchaseData.csv")

rowsKeep = .!(ismissing.(data.Grade) .& ismissing.(data.Price))
data = data[rowsKeep, :]

replace!(x -> ismissing(x) ? "QQ" : x, data.Name)
replace!(x -> ismissing(x) ? "31/06/2008" : x, data.Date)

grPr = by(dropmissing(data), :Grade, :Price=>x ->
        AvgPrice = round(mean(x), digits=-3))

d = Dict(grPr[:,1] .=> grPr[:,2])
nearIndx(v, x) = findmin(abs.(v.-x))[2]
for i in 1:nrow(data)
    if ismissing(data[i, :Price])
        data[i, :Price] = d[data[i, :Grade]]
    end
    if ismissing(data[i, :Grade])
        data[i, :Grade] = grPr[ nearIndx(grPr[:,2], data[i, :Price]), :Grade]
    end
end

println(first(data, 5), "\n")
println(describe(data))
```

```
5×4 DataFrame
| Row | Name     | Date       | Grade   | Price  |
|     | String   | String     | String  | Int64  |
-------------------------------------------------
| 1   | MARYANNA | 14/09/2008 | A       | 79700  |
| 2   | REBECCA  | 11/03/2008 | B       | 60000  |
| 3   | ASHELY   | 5/08/2008  | E       | 24311  |
| 4   | KHADIJAH | 2/09/2008  | D       | 38904  |
| 5   | TANJA    | 1/12/2008  | C       | 47052  |

4×8 DataFrame
| Row | variable | mean    | min       | median  | max       | nunique | nmissing | eltype
|
|     | Symbol   | Union   | Any       | Union   | Any       | Union   | Int64    | Union
|
-------------------------------------------------------------------------------------------------------------------
| 1   | Name     |         | ABBEY     |         | ZACHARY   | 183     | 0        | Union{Missing, String} |
| 2   | Date     |         | 1/07/2008 |         | 9/10/2008 | 142     | 0        | Union{Missing, String} |
| 3   | Grade    |         | A         |         | E         | 5       | 0        | Union{Missing, String} |
| 4   | Price    | 40037.9 | 8257      | 38045.5 | 79893     |         | 0        | Union{Missing, Int64}
|
```

In lines 4-5 we check if there are any rows with missing values in both the `:Grade` and `:Price` columns, and we remove them if present. First `ismissing()` is applied element wise over all values in each column, `.&` is then used to evaluate to `true` if both columns contain `missing`, and finally the preceding `.!` is used to flip the result, evaluating to `true` if the row should be kept. In our example there are no rows with missing values in both columns, so all rows are kept. In lines 7-8 we replace all missing names with the strings `"QQ"` and `"31/06/2008"` respectively via `replace!()`. In lines 10-11 `dropmissing()` and `by()` are used to calculate the mean price of each group, excluding rows with `missing` values. The results are rounded to the nearest thousand (`digits = -3`) and stored as the data frame `grPr`. In line 14 the dictionary `d` is created based on the values from `grPr`, with grade the key, and average price the value. In line 14 the `nearIndx()` function is created. It takes a value as input, `x`, and then finds the index of the nearest value from a given vector of values, `v`. In lines 15-22 we loop over each row in the data frame, and impute missing values in the price and grade columns. In lines 16-18 if the price entry is missing, then the grade is used to return the corresponding value stored in the dictionary `d`. Similarly, in lines 19-21 if the grade entry is missing, then the `nearIndx()` function is used to find the index of the closest value in `grPr` based on the price in `data`, and then `missing` is replaced by the corresponding grade. In lines 24-25 the first several rows of the data frame are printed, along with a summary of the cleaned data frame. At this point, no `missing` values are present.

4.2 Summarizing Data

Now that we have introduced data frames and methods of data processing we can explore basic methods of *descriptive statistics* to obtain data summaries. We focus on numerical data in a single sample, observations in pairs, and observations in vectors.

Single Sample

Given a set of observations, $x_1, \ldots, x_n$, we can compute a variety of descriptive statistics. The *sample mean* is often the most basic and informative *measure of centrality*. It is denoted by $\overline{x}$ and

is given by,

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}.$$

It is the *arithmetic mean* of the observations. However, this term, 'arithmetic mean', is not often used, unless we want to disambiguate it with the *geometric mean* or the *harmonic mean*, each respectively calculated via,

$$\overline{x}_g = \sqrt[n]{\prod_{i=1}^{n} x_i}, \qquad \text{and} \qquad \overline{x}_h = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}.$$

These two other *Pythagorean means* are not as popular in statistics as the arithmetic mean, however they are occasionally useful.

The geometric mean is useful for averaging growths factors. For example if $x_1 = 1.03$, $x_2 = 1.05$ and $x_3 = 1.07$ are growth factors, the geometric mean, $\overline{x}_g = 1.049714$ is a good summary statistic of the 'average growth factor'. This is because the growth factor obtained by $\overline{x}_g^3$ equals the growth factor $x_1 \cdot x_2 \cdot x_3$. Hence we if we start with an original base level of say 100 units (e.g. dollars) and exhibit growths of x_1, x_2, and x_3 above in three consecutive periods, then after three periods we have:

$$\text{Value after three periods} = 100 \cdot x_1 \cdot x_2 \cdot x_3 = 100 \cdot \overline{x}_g^3 = 115.7205.$$

Here the average growth factor is $\overline{x}_g$. Using the arithmetic mean, $\overline{x} = 1.05$ in such a case would yield 115.7625, which is slightly off from the correct value. Hence, in such scenarios, using the arithmetic mean to describe 'average growth' is not adequate.

The harmonic mean is useful for averaging rates or speeds. For example assume that you are on a brisk hike, walking 5 km up a mountain and then 5 km back down. Say your speed going up is $x_1 = 5$ km/h and your speed going down is $x_2 = 10$ km/h. What is your 'average speed' for the whole journey? You travel up for 1 hour and down for 0.5 hours and hence your total travel time is 1.5 hours. Hence the average speed is $10/1.5 = 6.6\overline{6}$ km/h. This is not the arithmetic mean which is 7.5 km/h but rather exactly equals the harmonic mean.

Also note that for any dataset $\overline{x}_h \leq \overline{x}_g \leq \overline{x}$. Here the inequalities become equalities only if all observations are equal. For $n = 2$, the second inequality can be obtained by manipulating the basic inequality $0 \leq (x_1 - x_2)^2$. Then for higher n it can be obtained by induction. The first inequality can then be obtained from the second inequality, since the harmonic mean is the reciprocal of the arithmetic mean of reciprocals.

A different breed of descriptive statistics is based on *order statistics*. This term is used to describe the *sorted sample*, and is sometimes denoted by,

$$x_{(1)} \leq x_{(2)} \leq \ldots \leq x_{(n)}.$$

Based on the order statistics we can define a variety of statistics such as the *minimum*, $x_{(1)}$, the *maximum*, $x_{(n)}$, and the *median*, which in the case of n being odd is $x_{((n+1)/2)}$ and in case of n being even is the arithmetic mean of $x_{(n/2)}$ and $x_{(n/2+1)}$. Like the sample mean, the median is a

measure of centrality. It is often preferable due to the fact that it isn't influenced by very high or very low measurements.

Related statistics are the α-*quantile*, for $\alpha \in [0,1]$ which is effectively $x_{(\widetilde{\alpha n})}$, where $\widetilde{\alpha n}$ denotes a rounding of αn to the nearest element of $\{1,\ldots,n\}$, or alternatively an interpolation similar to the case of the median with n even. For $\alpha = 0.25$ and $\alpha = 0.75$, these values are known as the *first quartile* and *third quartile* respectively. Finally the *inter quartile range (IQR)* is the difference between these two quartiles and the *range* is $x_{(n)} - x_{(1)}$.

The range and the IQR are *measures of dispersion*, meaning that the greater their magnitude, the more spread in the data. When dealing with measures of dispersion, the most popular and useful measure is the *sample variance*,

$$s^2 = \frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n-1} = \frac{\sum_{i=1}^{n} x_i^2 - n\,\overline{x}^2}{n-1}.$$

The sample variance is approximately the arithmetic mean of squared deviations from the sample mean, but isn't exactly because we divide by $n-1$ and not n (the latter is sometimes called *population variance*). If all observations are constant then $s^2 = 0$, otherwise $s^2 > 0$, and the bigger it is, the more dispersion we have in the data. A related quantity is the *sample standard deviation* s where $s := \sqrt{s^2}$. Also of interest is the *standard error* $s/\sqrt{n}$. Variances, standard deviations, and standard errors play a major role in the chapters that follow.

In Julia, functions for these descriptive statistics are implemented in the built-in `Statistics` package, with some additional functionality in the `StatsBase` package. Listing 4.11 illustrates their usage.

Listing 4.11: Summary statistics

```
using CSV, Statistics, StatsBase
data = CSV.read("../data/temperatures.csv")[:,4]

println("Sample Mean: ", mean(data))
println("Harmonic <= Geometric <= Arithmetic ",
        (harmmean(data), geomean(data), mean(data)))
println("Sample Variance: ",var(data))
println("Sample Standard Deviation: ",std(data))
println("Minimum: ", minimum(data))
println("Maximum: ", maximum(data))
println("Median: ", median(data))
println("95th percentile: ", percentile(data, 95))
println("0.95 quantile: ", quantile(data, 0.95))
println("Interquartile range: ", iqr(data),"\n")

summarystats(data)
```

```
Sample Mean: 27.1554054054054
Harmonic <= Geometric <= Arithmetic (26.52, 26.84, 27.155)
Sample Variance: 16.12538955837281
Sample Standard Deviation: 4.015643106449178
Minimum: 16.1
Maximum: 37.6
```

```
Median: 27.7
95th percentile: 33.0
0.95 quantile: 33.0
Interquartile range: 6.100000000000001

Summary Stats:
Length:         777
Missing Count:  0
Mean:           27.155405
Minimum:        16.100000
1st Quartile:   24.000000
Median:         27.700000
3rd Quartile:   30.100000
Maximum:        37.600000
```

In line 2 we load the data and select the fourth column. This sets `data` to be an array of `Float64`. In line 4 we compute and print the sample mean using `mean()`. We then compare it to the harmonic mean and geometric mean, computed via `harmmean()` and `geomean()` respectively. In line 7 we compute the sample variance using `var()` and then in line 8 the sample standard deviation via `std()`. In lines 9-14 we compute different statistics associated with order statistics including the min, max, median, and quartiles. Finally, in line 16 we use the `summarystats()` function which yields similar output.

Observations in Pairs

When data is configured in the form of pairs, $(x_1, y_1), \ldots, (x_n, y_n)$, we often consider the *sample covariance*, which is given by,

$$\widehat{\text{cov}}_{x,y} = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{n-1}, \tag{4.1}$$

where $\overline{x}$ and $\overline{y}$ are the sample means of $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_n)$ respectively. A positive covariance indicates a *positive linear relationship* meaning that when x is larger than its mean, we expect y to be larger than its mean, and similarly when x is small, y is small. A negative covariance indicates a *negative linear relationship* meaning that when x is large then y is small, and when x is small then y is large. If the covariance is 0 or near 0, it is an indication that no such relationship holds.

However, like the variance, the covariance is not a normalized quantity and hence depends on the units of measurement. For example say we were measuring x and y using kilograms and meters respectively and obtain $\widehat{\text{cov}}_{x,y} = 0.003$. Assume that we then decided to modify the data and represent x in grams by multiplying the original x values by 1000. From (4.1) you can observe that the covariance would change to $\widehat{\text{cov}}_{x,y} = 3$. If one was to naively interpret these numbers, in the first case it may appear that there is almost no positive linear relationship, while in the second case it appears that a positive linear relationship holds. However, nothing changed in the data except for the units of measurement and any relationship existing in the first data set (kilograms vs. meters) should also exist in the modified one (grams vs. meters).

For this reason we define another useful statistic, the *sample correlation (coefficient)*:

$$\hat{\rho}_{x,y} := \frac{\widehat{\text{cov}}_{x,y}}{s_x\, s_y},$$

where s_x and s_y are the sample standard deviations of the samples $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ respectively. Using the *Cauchy-Schwartz* inequality, we can show that $\hat{\rho}_{x,y} \in [-1, 1]$. The sign of $\hat{\rho}_{x,y}$ agrees with the sign of $\widehat{\text{cov}}_{x,y}$, however importantly its magnitude is meaningful. Having $|\hat{\rho}_{x,y}|$ near 0 implies little or no linear relationship, while $|\hat{\rho}_{x,y}|$ closer to 1 implies a stronger linear relationship, which is either positive or negative depending on the sign of $\hat{\rho}_{x,y}$. Also note that that if $(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$ then the sample covariance is simply the sample variance. That is $\widehat{\text{cov}}_{x,x} = s_x^2$.

It is often useful to represent the variances and covariances in the *sample covariance matrix* as follows,

$$\hat{\Sigma} = \begin{bmatrix} \widehat{\text{cov}}_{x,x} & \widehat{\text{cov}}_{x,y} \\ \widehat{\text{cov}}_{x,y} & \widehat{\text{cov}}_{y,y} \end{bmatrix} = \begin{bmatrix} s_x^2 & \hat{\rho}_{x,y}\, s_x\, s_y \\ \hat{\rho}_{x,y}\, s_x\, s_y & s_y^2 \end{bmatrix}. \tag{4.2}$$

In Listing 4.12, we import a weather observation dataset containing pairs of temperature observations (see Section 3.7). We then estimate the elements of the covariance matrix, and then store the results in the file `mvParams.jl`. Note that this file is used as input to Listing 3.34 at the end of Chapter 3.

Listing 4.12: Estimating elements of a covariance matrix

```
using DataFrames, CSV, Statistics

data = CSV.read("../data/temperatures.csv", copycols=true)
brisT = data.Brisbane
gcT = data.GoldCoast

sigB = std(brisT)
sigG = std(gcT)
covBG = cov(brisT, gcT)

meanVect = [mean(brisT) , mean(gcT)]
covMat = [sigB^2  covBG
          covBG   sigG^2]

outfile = open("../data/mvParams.jl","w")
write(outfile,"meanVect = $meanVect \ncovMat = $covMat")
close(outfile)
print(read("../data/mvParams.jl", String))
```

```
meanVect = [27.1554, 26.1638]
covMat = [16.1254 13.047; 13.047 12.3673]
```

In lines 3-5 we import the data and store the temperatures for Brisbane and Gold Coast as the arrays `brisT` and `gcT` respectively. In lines 7-8 the standard deviations of our temperature observations are calculated, and in line 9 the `cov()` function is used to estimate the covariance. In line 11 the means of our temperatures are calculated and stored as the array `meanVect`. In lines 12-13 the covariance matrix is calculated and assigned to the variable `covMat`. In lines 15-17 we save `meanVect` and `covMat` to the new Julia file, `mvParams.jl`. Note that this file is used as input for our calculations in Listing 3.34. First, in line 15 the `open()` function is used (with the argument `w`) to create the file `mvParams.jl` in write mode. Note that `open()` creates an input-output stream, `outfile`, which can then be written to. Then in line 16 `write()` is used to write to the input-output stream `outfile`. In line 17 the input-output stream `outfile` is closed. In line 18 the content of the file `mvParams.jl` is printed via the `read()` and `print()` functions.

Observations in Vectors

We now consider data that consists of n vectors. The i'th data vector represents a tuple of values, $(x_{i1}, \ldots, x_{ip})$. In this case, the data can be represented by a $n \times p$ *data matrix*, X, where the rows are observations (data vectors) and each column represents a different *variable*, *feature* or *attribute*. Such a layout is natural if considering the data as part of a data frame, see Section 4.1. However, in other cases, you may see this data matrix transposed such that each observation is a column vector of features.

In summarizing the data matrix X, a few basic objects arise. These include the *sample mean vector*, *sample standard deviation vector*, *sample covariance matrix*, and the *sample correlation matrix*. We now describe these.

The sample mean vector is simply a vector of length p where the j'th entry, $\overline{x}_j$ is the sample mean of $(x_{1j}, \ldots, x_{nj})$, based on the j'th column of X. Similarly the sample standard deviation vector has a j'th entry, s_j, which is the sample standard deviation of $(x_{1j}, \ldots, x_{nj})$.

With these we often *standardize* (also called *normalize*) the data by creating a new $n \times p$ matrix Z, with entries,

$$z_{ij} = \frac{x_{ij} - \overline{x}_j}{s_j}, \quad i = 1, \ldots, n, \quad j = 1, \ldots, p, \tag{4.3}$$

sometimes called *z-scores*. It can be created via, $Z = (X - \mathbf{1}\overline{x}^T)\,\mathrm{diag}(s)^{-1}$, where $\mathbf{1}$ is a column vector of 1's, $\overline{x}$ is the mean vector, s is the standard deviation vector, and $\mathrm{diag}(\cdot)$ creates a diagonal matrix from a vector. The standardized data has the attribute that each column, $(z_{1j}, \ldots, z_{nj})^T$, has a 0 sample mean and a unit standard deviation. Hence first and second order information of the j'th feature is lost when moving from the data matrix X to the standardized matrix Z. Nevertheless, relationships between features are still captured in Z and can be easily calculated. Most notably, the sample correlation between feature j_1 and feature j_2, denoted by $\hat{\rho}_{j_1,j_2}$ is simply calculated via,

$$\hat{\rho}_{j_1 j_2} = \frac{\sum_{i=1}^{n} z_{ij_1} z_{ij_2}}{n-1} = \Big[\frac{1}{n-1} Z^T Z\Big]_{j_1\, j_2}. \tag{4.4}$$

Here the second expression means taking the j_1, j_2 entry from the $p \times p$ sample correlation matrix $Z^T Z/(n-1)$. In Julia this can be performed via the `cor()` function.

Without resorting to standardization, it is often of interest to calculate the $p \times p$ *sample covariance matrix* $\hat{\Sigma}$ generalizing (4.2). Here the j_1, j_2 entry is the covariance between the j_1 and j_2 variables. The matrix can be computed in several ways. For example,

$$\hat{\Sigma} = \frac{1}{n-1}(X - \mathbf{1}\bar{x}^T)^T(X - \mathbf{1}\bar{x}^T) = \frac{1}{n-1}X^T(I - n^{-1}\mathbf{1}\mathbf{1}^T)X. \tag{4.5}$$

In the second expression, I is the identity matrix and as before $\mathbf{1}$ is a vector of 1's and hence $\mathbf{1}\mathbf{1}^T$ is a matrix of 1's. Note that $(I - n^{-1}\mathbf{1}\mathbf{1}^T)X$ is the *de-meaned* data. Also note that the symmetric matrix $(I - n^{-1}\mathbf{1}\mathbf{1}^T)$ multiplied by itself is itself. In Julia, this calculation can be performed via the `cov()` function. We now illustrate several alternative ways for computing the sample covariance and sample correlation in Listing 4.13 below. In addition to the `cov()`, `cor()`, `mean()`, and `std()` functions, the listing also illustrates the use of the `zscore()` function from `StatsBase`.

Listing 4.13: Sample covariance

```
using Statistics, StatsBase, LinearAlgebra, DataFrames, CSV
df = CSV.read("../data/3featureData.csv",header=false)
n, p = size(df)
println("Number of features: ", p)
println("Number of observations: ", n)
X = convert(Array{Float64,2},df)
println("Dimensions of data matrix: ", size(X))

xbarA = (1/n)*X'*ones(n)
xbarB = [mean(X[:,i]) for i in 1:p]
xbarC = sum(X,dims=1)/n
println("\nAlternative calculations of (sample) mean vector: ")
@show(xbarA), @show(xbarB), @show(xbarC)

Y = (I-ones(n,n)/n)*X
println("Y is the de-meaned data: ", mean(Y,dims=1))

covA = (X .- xbarA')'*(X .- xbarA')/(n-1)
covB = Y'*Y/(n-1)
covC = [cov(X[:,i], X[:,j]) for i in 1:p, j in 1:p]
covD = [cor(X[:,i], X[:,j])*std(X[:,i])*std(X[:,j]) for i in 1:p, j in 1:p]
covE = cov(X)
println("\nAlternative calculations of (sample) covariance matrix: ")
@show(covA), @show(covB), @show(covC), @show(covD), @show(covE)

ZmatA = [(X[i,j] - mean(X[:,j]))/std(X[:,j]) for i in 1:n, j in 1:p ]
ZmatB = hcat([zscore(X[:,j]) for j in 1:p]...)
println("\nAlternate computation of Z-scores yields same matrix: ",
        maximum(norm(ZmatA-ZmatB)))
Z = ZmatA

corA = covA ./ [std(X[:,i])*std(X[:,j]) for i in 1:p, j in 1:p]
corB = covA ./ (std(X,dims = 1)'*std(X,dims = 1))
corC = [cor(X[:,i],X[:,j]) for i in 1:p, j in 1:p]
corD = Z'*Z/(n-1)
corE = cov(Z)
corF = cor(X)
println("\nAlternative calculations of (sample) correlation matrix: ")
@show(corA), @show(corB), @show(corC), @show(corD), @show(corE), @show(corF);
```

```
Number of features: 3
Number of observations: 7
Dimensions of data matrix: (7, 3)

Alternative calculations of (sample) mean vector:
xbarA = [1.05714, 2.08571, 3.5]
xbarB = [1.05714, 2.08571, 3.5]
xbarC = [1.05714, 2.08571, 3.5]
Y is the de-meaned data: [6.74064e-17 3.24889e-16 2.85486e-16]

Alternative calculations of (sample) covariance matrix:
covA = [0.119524 -0.087381 0.44; -0.087381 0.121429 -0.715; 0.44 -0.715 8.03333]
covB = [0.119524 -0.087381 0.44; -0.087381 0.121429 -0.715; 0.44 -0.715 8.03333]
covC = [0.119524 -0.087381 0.44; -0.087381 0.121429 -0.715; 0.44 -0.715 8.03333]
covD = [0.119524 -0.087381 0.44; -0.087381 0.121429 -0.715; 0.44 -0.715 8.03333]
covE = [0.119524 -0.087381 0.44; -0.087381 0.121429 -0.715; 0.44 -0.715 8.03333]

Alternate computation of Z-scores yields same matrix: 2.220446049250313e-16

Alternative calculations of (sample) correlation matrix:
corA = [1.0 -0.725319 0.449032; -0.725319 1.0 -0.723932; 0.449032 -0.723932 1.0]
corB = [1.0 -0.725319 0.449032; -0.725319 1.0 -0.723932; 0.449032 -0.723932 1.0]
corC = [1.0 -0.725319 0.449032; -0.725319 1.0 -0.723932; 0.449032 -0.723932 1.0]
corD = [1.0 -0.725319 0.449032; -0.725319 1.0 -0.723932; 0.449032 -0.723932 1.0]
corE = [1.0 -0.725319 0.449032; -0.725319 1.0 -0.723932; 0.449032 -0.723932 1.0]
corF = [1.0 -0.725319 0.449032; -0.725319 1.0 -0.723932; 0.449032 -0.723932 1.0]
```

In line 2 we read the data with `header = false` since there isn't a line in the csv for the variable (or feature) names. In line 3 we use the `size()` function to set the number of observations, `n`, and number of features `p`. The `convert()` function is used in line 6 to extract a data matrix out of the data frame `df`. Lines 9-13 show alternative ways of computing the sample mean vector. Note the use of `dims=1` in the `sum()` function in line 11, indicating to sum over columns. In line 15 we create the de-meaned data Y, and show in line 16 that the mean is 0 (effectively 0 in the output). Lines 18-24 illustrate a variety of ways to calculate the sample covariance matrix using several forms of (4.5). Lines 26-30 deal with standardized data as in (4.3). The printout of the maximum of the norm in line 29 is a way for seeing that the two matrices `ZmatA` and `ZmatB` are identical. Finally, lines 32-39 compute the correlation matrix in a variety of ways. Observe line 35 implementing (4.4). Also observe that the covariance of Z is the correlation of X as shown in lines 36-37.

4.3 Plots for Single Samples and Time Series

In this section we deal with plots focused on a single collection of observations (numbers), $x_1, \ldots, x_n$, where in certain cases we plot several such single collections jointly for comparison. If the observations are obtained by randomly sampling a population, then the order of the observations is inconsequential. In this case we say the data is a single sample and in general, plotting the observations one after the other isn't particularly useful. However, if the observations represent measurement over time then we call the dataset a *time-series*, and in this case plotting the observations one after the other is the standard way for considering temporal patterns in the data.

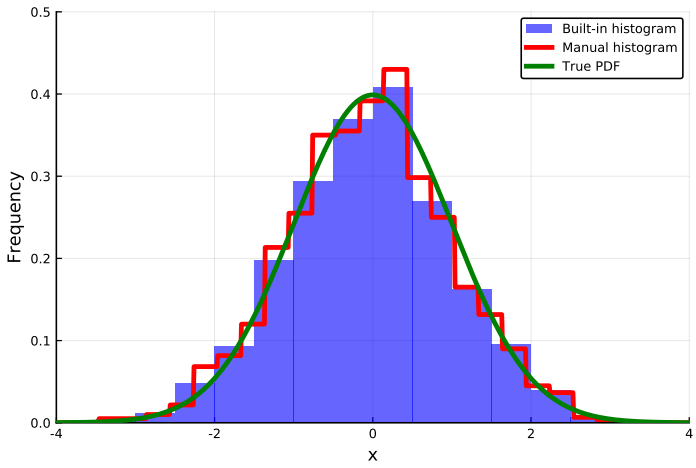

Figure 4.1: A manually created histogram with the same number of bins as `histogram()`. Both are compared to original PDF.

Histograms

In both the single sample and time series case, considering frequencies of occurrences is generally an insightful way to visualize the data. The most standard mechanism for plotting frequencies is the *histogram*, already used extensively in previous chapters (see for example Listing 1.10). Mathematically, a histogram can be defined as follows. First denote the support of the observations via $[\ell, m]$ where ℓ is the minimal observation and m is the maximal observation. Then the interval $[\ell, m]$ is partitioned into a finite set of *bins* $\mathcal{B}_1, \ldots, \mathcal{B}_L$, and the frequency in each bin is recorded via,

$$f_j = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}\{x_i \in \mathcal{B}_j\}, \qquad \text{for} \quad j = 1, \ldots, L. \tag{4.6}$$

Here $\mathbf{1}\{\cdot\}$ is 1 if $x_i \in \mathcal{B}_j$ or 0 if $x_i \notin \mathcal{B}_j$. We have that $\sum f_j = 1$ and hence $f_1, \ldots, f_L$ is a discrete probability distribution.

A histogram is then just a visual representation of this discrete probability distribution (the frequencies). One way to plot the frequencies is via a *stem plot* (see for example Figure 3.10 illustrating a binomial distribution). However such a plot would not represent the widths of the bins. Hence an alternative representation is via a *histogram function* $h(x)$ which is a scaled plot of the frequencies $f_1, \ldots, f_L$. The function $h(x)$ is defined for any $x \in [\ell, m]$. It is constructed by staying constant on all values $x \in \mathcal{B}_j$, at a height of $f_j/|\mathcal{B}_j|$ where $|\mathcal{B}_j|$ is the width of bin j. This ensures the total area under the plot is 1. Hence $h(x)$ is actually a probability density function, and can hence be compared to probability densities. Mathematically,

$$h(x) = \sum_{j=1}^{L} \frac{f_j}{|\mathcal{B}_j|} \mathbf{1}\{x \in \mathcal{B}_j\} = \frac{f_{b(x)}}{|\mathcal{B}_{b(x)}|}, \tag{4.7}$$

where $b(x)$ is the bin of x, that is $x \in \mathcal{B}_{b(x)}$.

Note that there are a multitude of ways for choosing the number of bins L and the actual bins $\mathcal{B}_1, \ldots, \mathcal{B}_L$. Different histogram implementations will use different *bin selection heuristics*. We don't

discuss these methods here. Throughout this book we use the `histogram()` function from `Plots` and it contains a default bin selection heuristic. In certain cases we specify the number of bins using the keyword argument, `bins`, often making a judicious choice based on the appearance for a specific dataset. It is important to keep in mind that histograms are clearly not unique representations of the data, because there is not a unique way to choose the bins.

For demonstration purposes we implement a manual histogram in Listing 4.14 and compare it to `histogram()` from plots. The results are in Figure 4.1. The demonstration illustrates that a histogram is not a unique representation of the data. Both the manually created histogram and the built-in histogram have `L` bins, however a different choice of actual bins creates different histograms. Note that if in line 21, `L` is replaced with `first.(bins)`, then the two histograms will agree because they use the exact same bins. Also note that our implementation is not an efficient one, but rather aims to illustrate the use of the above equations directly. A related classic plot that we do not survey here is the *stem and leaf plot*.

Listing 4.14: Creating a manual histogram

```
1  using Plots, Distributions, Random; pyplot()
2  Random.seed!(0)
3
4  n = 2000
5  data = rand(Normal(),n)
6  l, m = minimum(data), maximum(data)
7
8  delta = 0.3;
9  bins = [(x,x+delta) for x in l:delta:m-delta]
10 if last(bins)[2] < m
11     push!(bins,(last(bins)[2],m))
12 end
13 L = length(bins)
14
15 inBin(x,j) = first(bins[j]) <= x && x < last(bins[j])
16 sizeBin(j) = last(bins[j]) - first(bins[j])
17 f(j) = sum([inBin(x,j)  for x in data])/n
18 h(x) = sum([f(j)/sizeBin(j) * inBin(x,j) for j in 1:L])
19
20 xGrid = -4:0.01:4
21 histogram(data,normed=true, bins=L,
22     label="Built-in histogram",
23     c=:blue, la=0, alpha=0.6)
24 plot!(xGrid,h.(xGrid), lw=3, c=:red, label="Manual histogram",
25     xlabel="x",ylabel="Frequency")
26 plot!(xGrid,pdf.(Normal(),xGrid),label="True PDF",
27     lw=3, c=:green, xlims=(-4,4), ylims=(0,0.5))
```

In lines 4-6 we deal with the data. It is artificially sampled from a standard normal distribution. Lines 8-13 detail our choice of bins. In this case, `L` is implicitly defined based on the bin width `delta`. The statement in line 11 is executed when `l-m` is not a multiple of `delta` and adds an additional final bin (potentially smaller than the rest of the bins). The function `inBin()` implements $\mathbf{1}\{x \in \mathcal{B}_j\}$. The function `sizeBin()` implements $|\mathcal{B}_j|$. The function `f()` implements f_j as in (4.6). We then use these in line 18 to implement $h(x)$ as in the first representation of (4.7). Lines 21-23 plot the histogram using `histogram()` where we specify `L` bins. Lines 24-25 plot our manual implementation of the histogram via `h()`. For comparison, we also plot the PDF of the data in lines 26-27.

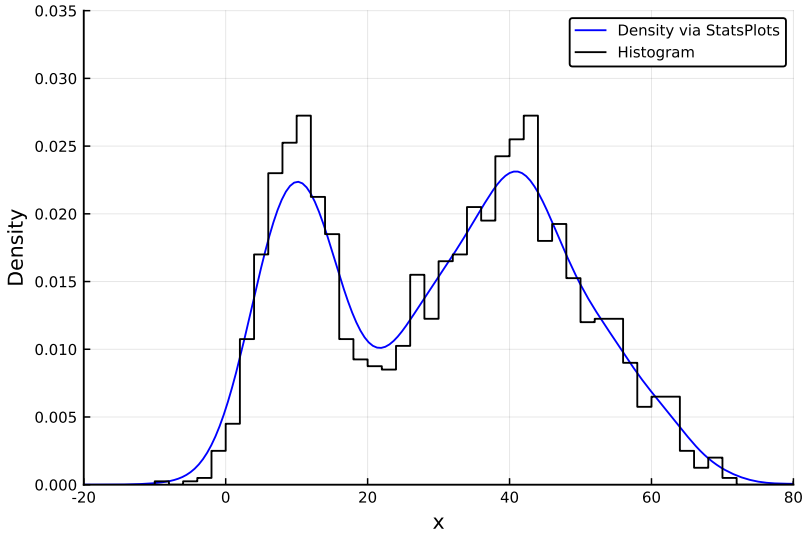

Figure 4.2: Histogram of the underlying data, and the KDE, as generated from the `density()` function in `StatsPlots`.

Density Plots and Kernel Density Estimation

A more modern and visually appealing alternative to histograms is the *smoothed histogram*, also known as a *density plot*, often generated via a *kernel density estimate*. Before we describe and detail kernel density estimation, let's see how to use it to create a smoothed histogram as in Figure 4.2. The figure is generated by Listing 4.15, using the `density()` function from the `StatsPlots` package. The plot is compared to a histogram. Its usage is similar to the `histogram()` or `stephist()` functions from `Plots`, however the result is strikingly different, yielding a smooth curve.

This example and the next are based on synthetic data from a *mixture model*. Such models are useful for situations where we sample from populations made up of heterogeneous sub-populations. Each sub-population has its own probability distribution and these are "mixed" in the process of sampling. At first a *latent* (un-observed) random variable determines which sub-population is used, and then a sample is taken from that sub-population. In terms of random variable generation, creating a mixture simply involves first randomly selecting which probability distribution is used, and then generating an observation from it. Also, the probability density function of the mixture is a convex combination of the probability density functions of each of the sub-populations. That is, if the M sub-populations have densities $g_1(x), \ldots, g_M(x)$ with weights, $p_1, \ldots, p_M$ and $\sum p_i = 1$, then the density of the mixture is,

$$f(x) = \sum_{i=1}^{M} p_i \, g_i(x).$$

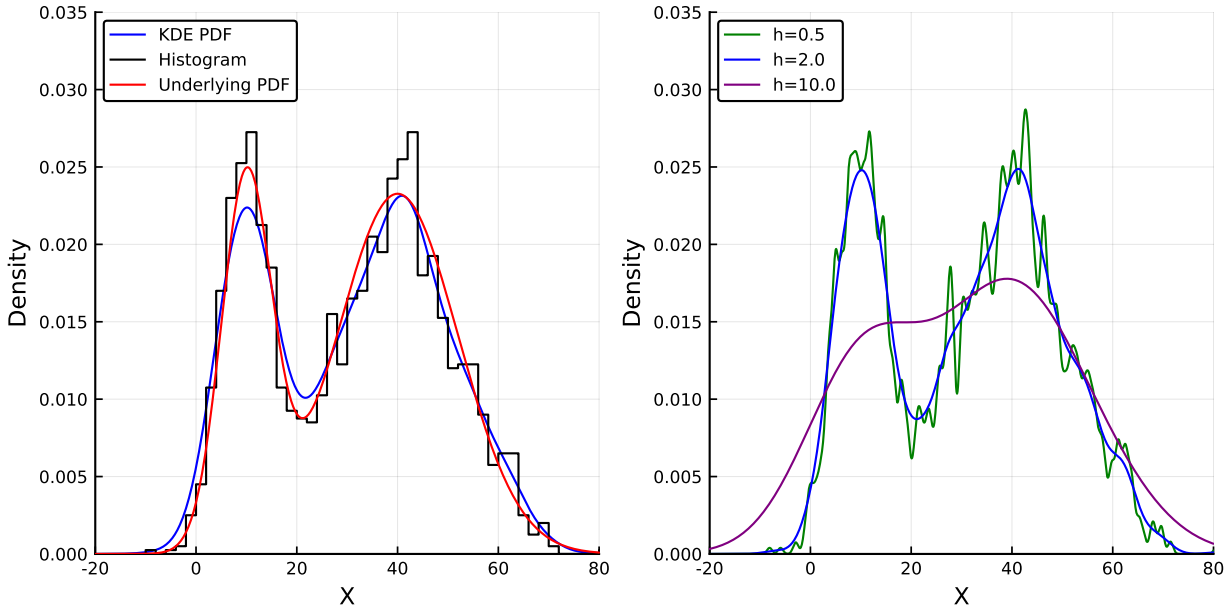

Figure 4.3: Left: KDE compared to the actual underlying PDF as well as a histogram. Right: KDE obtained via several bandwidths settings.

Listing 4.15: Classic vs. smooth histograms

```
1  using Random, Distributions, StatsPlots; pyplot()
2  Random.seed!(0)
3
4  mu1, sigma1 = 10, 5
5  mu2, sigma2 = 40, 12
6  dist1, dist2 = Normal(mu1,sigma1), Normal(mu2,sigma2)
7  p = 0.3
8  mixRv() = (rand() <= p) ? rand(dist1) : rand(dist2)
9
10 n = 2000
11 data = [mixRv() for _ in 1:n]
12
13 density(data, c=:blue, label="Density via StatsPlots",
14         xlims=(-20,80), ylims=(0,0.035))
15 stephist!(data, bins=50, c=:black, norm=true,
16     label="Histogram", xlabel="x", ylabel = "Density")
```

Lines 4-8 deal with the mixture random variable. It is a mixture of two normal distributions, each with parameters as specified in lines 4-5. The mixture places a probability of `p = 0.3` of being from the first distribution and hence a probability of 0.7 of being from the second. Line 8 defines the function that generates the mixture random variable. It evaluates to `rand(dist1)` with probability 0.3 and `rand(dist2)` with probability 0.7. Lines 10-11 generate data samples from this mixture distribution. Lines 13-14 create the density plot. Lines 15-16 plot a histogram for comparison.

How is a density plot created? The typical way is via *kernel density estimation* (*KDE*), which is a way of fitting a probability density function to data. When we used the `density()` function from `StatsPlots` above, KDE was implicitly invoked. However, in certain cases, we may wish to have

access to the estimated density. For this we use the `kde()` function from the `KernelDensity` package. Let us first explain how kernel density estimation works.

Given a set of observations, $x_1, \ldots, x_n$, the KDE is the function,

$$\hat{f}(x) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{h} K\Big(\frac{x - x_i}{h}\Big), \tag{4.8}$$

where $K(\cdot)$ is some specified *kernel function* and $h > 0$ is the *bandwidth* parameter. The kernel function is a function that satisfies the properties of a PDF. A typical example is the Gaussian kernel.

$$K(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}.$$

With such a kernel (or any other) the estimator $\hat{f}(x)$ is a PDF because it is a weighted superposition of scaled kernel functions centered about each of the observations. Like histograms, KDEs are not unique as they depend on the type of kernel function used and more importantly on the bandwidth parameter. A very small bandwidth implies that the density,

$$\frac{1}{h} K\Big(\frac{x - x_i}{h}\Big), \tag{4.9}$$

is very concentrated around x_i. This in turn implies that the KDE (4.8) is comprised of a superposition of very concentrated functions, one for each observation. In contrast, a very large bandwidth implies that the density (4.9) around each observation has a very wide spread. This will make the KDE 'smear' over a wide range. Hence ideally, the bandwidth is not too small nor too large. The right hand plot of Figure 4.3 illustrates KDE with different choices of bandwidth. As can be seen when $h = 0.5$ the KDE appears to have multiple spikes. At the other extreme, when $h = 10$ the KDE is very 'smeared'.

For any value of h, it can be proved under general conditions that if the data is distributed according to some density $f(\cdot)$, then $\hat{f}(\cdot)$ converges to $f(\cdot)$ when the sample size grows. Nevertheless, in practice, choosing the bandwidth is a key issue in the application of KDE. A default classic rule is *Silverman's rule*, which is based on the sample standard deviation of the sample, s. The rule is,

$$h = \Big(\frac{4}{3}\Big)^{1/6} s\, n^{-1/5} \approx 1.06\, s\, n^{-1/5}.$$

There is some theory justifying this h in certain cases, and in other cases more advanced rules such as [BGK10] perform better. Listing 4.16 carries out KDE for the same synthetic data as the previous example. It generates Figure 4.3 where the left plot compares the KDE to the underlying PDF of the mixture and the right plot presents the effect of changing the bandwidths.

Listing 4.16: Kernel density estimation

```
1  using Random, Distributions, KernelDensity, Plots; pyplot()
2  Random.seed!(0)
3
4  mu1, sigma1 = 10, 5
5  mu2, sigma2 = 40, 12
6  dist1, dist2 = Normal(mu1,sigma1), Normal(mu2,sigma2)
7  p = 0.3
8  mixRv() = (rand() <= p) ? rand(dist1) : rand(dist2)
9  mixPDF(x) = p*pdf(dist1,x) + (1-p)*pdf(dist2,x)
10
11 n = 2000
12 data = [mixRv() for _ in 1:n]
13
14 kdeDist = kde(data)
15
16 xGrid = -20:0.1:80
17 pdfKDE = pdf(kdeDist,xGrid)
18
19 plot(xGrid, pdfKDE, c=:blue, label="KDE PDF")
20 stephist!(data, bins=50, c=:black, normed=:true, label="Histogram")
21 p1 = plot!(xGrid, mixPDF.(xGrid), c=:red, label="Underlying PDF",
22     xlims=(-20,80), ylims=(0,0.035), legend=:topleft,
23     xlabel="X", ylabel = "Density")
24
25 hVals = [0.5,2,10]
26 kdeS = [kde(data,bandwidth=h) for h in hVals]
27 plot(xGrid, pdf(kdeS[1],xGrid), c = :green, label= "h=$(hVals[1])")
28 plot!(xGrid, pdf(kdeS[2],xGrid), c = :blue, label= "h=$(hVals[2])")
29 p2 = plot!(xGrid, pdf(kdeS[3],xGrid), c = :purple, label= "h=$(hVals[3])",
30     xlims=(-20,80), ylims=(0,0.035), legend=:topleft,
31     xlabel="X", ylabel = "Density")
32 plot(p1,p2,size = (800,400))
```

The first 12 lines are similar to the previous code example with an exception of line 9 that defines the function `mixPDF()` which is the PDF of the mixture distribution. In line 14 we invoke the function `kde()` to generate a KDE type object `kdeDist`, based on `data`. The `KernelDensity` package supplies methods for the `pdf()` function that can be applied to `UnivariateKDE` objects such as `kdeDist`. This is used in line 17 to create the array `pdfKDE` over `xGrid`. Lines 19-23 plot the KDE, a histogram of the data, and the actual PDF. These plots make up `p1` which is the left hand of the figure. The right hand side `p2` is created in lines 25-32.

Empirical Cumulative Distribution Function

While KDE is a useful way to estimate the PDF of the unknown underlying distribution given some sample data, the *Empirical Cumulative Distribution Function* (ECDF) may be viewed as an estimate of the underlying CDF. In contrast to histograms and KDEs, ECDFs provide a unique representation of the data not dependent on tuning parameters, such as the bins for histograms, or the bandwidth and kernel function for KDE.

The ECDF is a stepped function which, given n data points, increases by $1/n$ at each point.

Mathematically, given the sample, $x_1, \ldots, x_n$ the ECDF is given by,

$$\hat{F}(t) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}\{x_i \leqslant t\} \qquad \text{where } \mathbf{1}\{\cdot\} \text{ is the indicator function.}$$

In the case of i.i.d. data from an underlying distribution with CDF $F(\cdot)$, the *Glivenko-Cantelli theorem* ensures that the ECDF $\hat{F}(\cdot)$ approaches $F(\cdot)$ as the sample size grows.

Constructing an ECDF is possible in Julia through the `ecdf()` function contained in the `StatsBase` package. In Listing 4.17 we use synthetic data from the same mixture distribution as in the two previous examples. We obtain the ECDF for a sample of size $n = 30$ and then again for $n = 100$. We compare the ECDFs to the underlying actual CDF of the mixture distribution. See Figure 4.4.

Listing 4.17: Empirical cumulative distribution function

```
1  using Random, Distributions, StatsBase, Plots; pyplot()
2  Random.seed!(0)
3
4  mu1, sigma1 = 10, 5
5  mu2, sigma2 = 40, 12
6  dist1, dist2 = Normal(mu1,sigma1), Normal(mu2,sigma2)
7  p = 0.3
8  mixRv() = (rand() <= p) ? rand(dist1) : rand(dist2)
9  mixCDF(x) = p*cdf(dist1,x) + (1-p)*cdf(dist2,x)
10
11 n = [30, 100]
12 data1 = [mixRv() for _ in 1:n[1]]
13 data2 = [mixRv() for _ in 1:n[2]]
14
15 empiricalCDF1 = ecdf(data1)
16 empiricalCDF2 = ecdf(data2)
17
18 xGrid = -10:0.1:80
19 plot(xGrid,empiricalCDF1.(xGrid), c=:blue, label="ECDF with n = $(n[1])")
20 plot!(xGrid,empiricalCDF2.(xGrid), c=:red, label="ECDF with n = $(n[2])")
21 plot!(xGrid, mixCDF.(xGrid), c=:black, label="Underlying CDF",
22     xlims=(-10,80), ylims=(0,1),
23     xlabel="x", ylabel="Probability", legend=:topleft)
```

The first few lines of the code block are similar to the previous examples using a mixture distribution. A difference is that in line 9 we define the function `mixCDF()` which is the CDF of the mixture distribution. We then generate two samples in lines 12-13, of varying sample sizes. In lines 15-16 we invoke the `ecdf()` function from `StatsBase`. The returned object can then be used as a function, evaluating $\hat{F}(\cdot)$ at any point. This is done in lines 19-20 where we plot the ECDFs evaluated on `xGrid`. Then lines 21-23 plot the actual CDF.

Normal Probability Plot

We now introduce the *normal probability plot*. This plot can be used to indicate if it is likely that a data set has come from a population that is normally distributed or not. It works by plotting

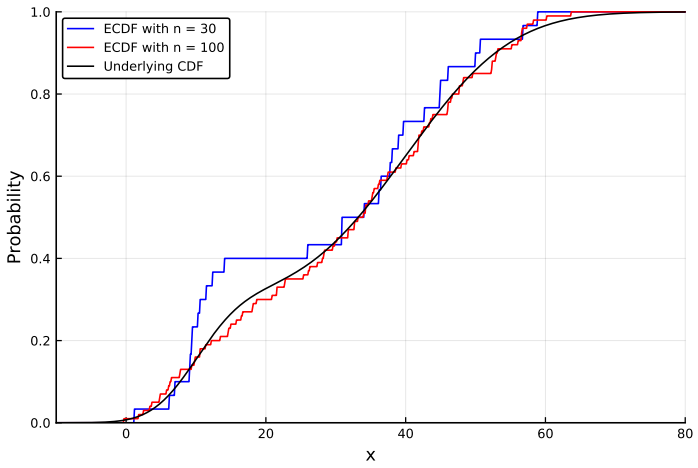

Figure 4.4: The ECDF from a sample compared to the population CDF.

the quantiles of the dataset in question against the theoretical quantiles that one would expect if the sample data came from a normal distribution, and checking if the plot is linear. The normal probability plot is actually a special case of the more generalized *Q-Q plot*, or *quantile-quantile plot* that is described in more detail in the next section.

In order to create a normal probability plot, the data points are first sorted in ascending order, $x_1, \ldots, x_n$, then the quantiles of each data point calculated. Finally, n equally-spaced quantiles of the standard normal distribution are calculated, and each ascending quantile pair is then plotted. If the data comes from a normal distribution then we can expect the normal probability plot to follow a straight line, otherwise not. An alternative view of the normal probability plot is to think of the ECDF of the data, plotted with the y-axis stretched according to the inverse of the CDF of the normal distribution.

In Listing 4.30 we create Figure 4.5 which presents two normal probability plots. It is based on two synthetic data sets that have a similar mean and a similar variance. The first comes from a normal distribution and the second from an exponential distribution. As can be seen, the "non-normality" of the data coming from the exponential distribution is very apparent.

Listing 4.18: Normal probability plot

```
using Random, Distributions, StatsPlots, Plots; pyplot()
Random.seed!(0)

mu = 20
d1, d2 = Normal(mu,mu), Exponential(mu)

n = 100
data1 = rand(d1,n)
data2 = rand(d2,n)

qqnorm(data1, c=:blue, ms=3, msw=0, label="Normal Data")
qqnorm!(data2, c=:red, ms=3, msw=0, label="Exponential Data",
        xlabel="Normal Theoretical Quantiles",
        ylabel="Data Quantiles", legend=true)
```

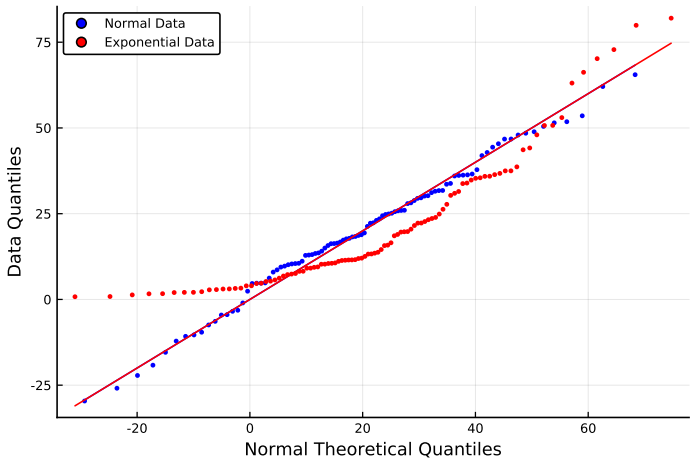

Figure 4.5: Comparing two normal probability plots. One from a normal population and one from an exponential population.

The distributions for the two synthetic data sets are defined in lines 4-5. You can check that they have the same theoretical mean and variance by using `mean()` and `var()` on `d1` and `d2`. The samples are then generated in lines 7-9. Lines 11-14 then plot the normal probability plots via the `qqnorm()` and `qqnorm!()` functions from `StatsPlots`. The second function has a `!` in the name similar to other plotting functions that add onto an existing plot.

Visualizing Time-Series

Moving from single sample data to time-series, we now illustrate a basic example. We create time-series plots of two time-series together with an associated histogram. Later, we also show how a radial plot can help visualize cyclic temporal patterns on the same data. In general, when confronted with time series data, simply plotting a histogram of the data can be misleading because the frequencies in the histogram can be greatly affected by trends or cyclic patterns in the data. For example, the gross domestic product (GDP) of China has risen in the past 20 years from around a trillion US dollars (USD) to roughly 12.5 trillion USD. It does not make sense to plot a histogram of this data, because it would not capture the distribution of the GDP.

Nevertheless, in cases where the time-series data appears to be *stationary*, then a histogram is immediately insightful. Broadly speaking, a stationary sequence is one in which the distributional law of observations does not depend on the exact time. This means that there isn't an apparent trend nor a cyclic component. To illustrate these concepts we present Figure 4.6. The top left plot presents two time-series of temperature data in the adjacent locations of Brisbane and Gold Coast Australia. As apparent from the plot, the sequences are clearly non-stationary. This is because of seasonality. The top right plot shows a zoomed in view of a specific fortnight. The bottom left plot is a time-series of the differences in temperatures between Brisbane and Gold Coast. On initial inspection, this time-series appears to be stationary. Hence for the difference we plot a histogram in the bottom right. The code for generating Figure 4.6 is in Listing 4.19.

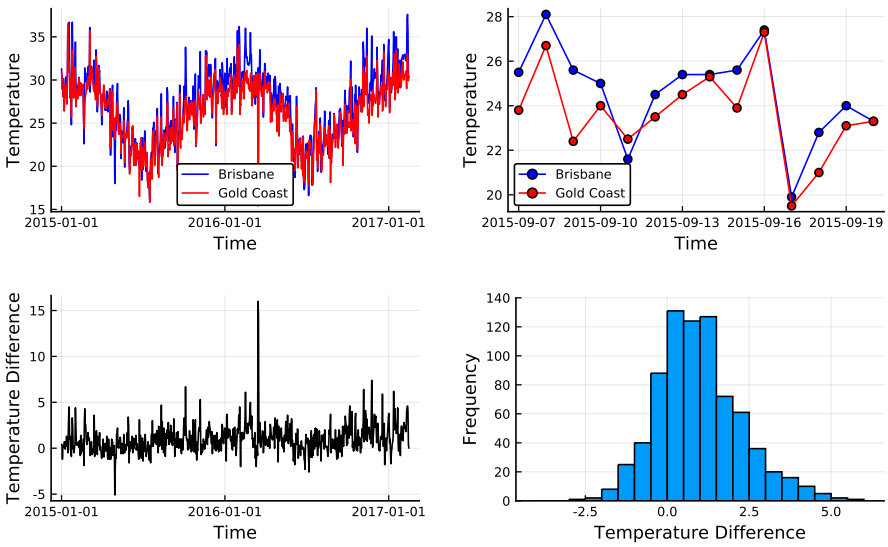

Figure 4.6: Simple plots for time-series data. Top left: Temperatures over time. Top Right: Zooming in on a specific week. Bottom left: Differences in temperature. Bottom right: Histogram of the difference in temperature.

Listing 4.19: Multiple simple plots for a time series

```
using DataFrames, CSV, Statistics, Dates, Plots, Measures; pyplot()

data = CSV.read("../data/temperatures.csv")
brisbane = data.Brisbane
goldcoast = data.GoldCoast

diff = brisbane - goldcoast
dates = [Date(
            Year(data.Year[i]),
            Month(data.Month[i]),
            Day(data.Day[i])
        ) for i in 1:nrow(data)]

fortnightRange = 250:263
brisFortnight = brisbane[fortnightRange]
goldFortnight = goldcoast[fortnightRange]

default(xlabel="Time", ylabel="Temperature")
default(label=["Brisbane" "Gold Coast"])

p1 = plot(dates, [brisbane goldcoast],
        c=[:blue :red])
p2 = plot(dates[fortnightRange], [brisFortnight goldFortnight],
        c=[:blue :red], m=(:dot, 5, Plots.stroke(1)))
p3 = plot(dates, diff,
        c=:black, ylabel="Temperature Difference",legend=false)
p4 = histogram(diff, bins=-4:0.5:6,
        ylims=(0,140), legend = false,
        xlabel="Temperature Difference", ylabel="Frequency")
plot(p1,p2,p3,p4, size = (800,500), margin = 5mm)
```

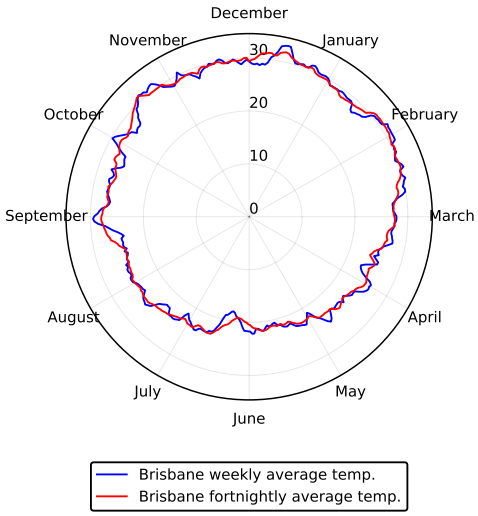

Figure 4.7: A radial plot of (time) averaged weekly and fortnightly temperatures for Brisbane in 2015.

In lines 3-5 we read the data and create the arrays `brisbane` and `goldcoast` describing the temperatures in these respective locations. In line 7 we create the array `diff` made up of temperature differences. In lines 8-12 we create the array `dates` which contains `Date` objects mapped to the days of temperature measurement. It is constructed based on the `Year`, `Month`, and `Day` columns of the data frame by using the respective functions from the `Dates` package. In line 14 we define a range of days spanning a fortnight, `fortnightRange`. This is then used to splice that fortnight from the temperature data into `brisFortnight` and `goldFortnight`. In this plotting example we use the `default()` function from `Plots` to set some default argument for each subplot. This is in lines 18-19. We then create the plots in lines 21-30, overriding the defaults in certain cases.

Radial Plot

It is often useful to plot time-series data, or cyclic data, on a so called *radial plot*. Such a plot involves plotting data on a polar coordinate system. See Figure 4.7. This plot can be used to help visualize the nature of a dataset by comparing the distances of each data point radially from the origin. A variation of the radial plot is the *radar plot*, which is often used to visualize the levels of different categorical variables on the one plot.

For our example of a radial plot we use the Brisbane temperature data in 2015, similar to the data plotted in the previous listing. This time, we present the effect of different forms of *smoothing* on the time-series data. For this we carry out a *moving average* on the data. Roughly, this transforms the original data sequence $x_1, \ldots, x_n$ into a smoother sequence $\tilde{x}_1, \ldots, \tilde{x}_n$ via,

$$\tilde{x}_i = \frac{1}{L} \sum_{j=0}^{L-1} x_{i-j}.$$

Hence each $\tilde{x}_i$ is the average of the L observations $x_{i-L+1}, \ldots, x_i$ neighbouring time i in the original sequence. A critical parameter is the *window size* L which determines "how much smoothing" is

to be performed. With $L = 1$ no smoothing takes places and as L is increased more smoothing takes place. There are also minor details that we don't specify here associated with shifting the smoothing window and with edge effects.

In Listing 4.20 we use the `TimeSeries` package to carry out such smoothing, comparing two window size values, $L = 7$ (weekly) and $L = 14$ (fortnightly). This package contains a variety of more advanced time series manipulation functions. Refer to the documentation for more examples. The listing then plots the smoothed data in Figure 4.7. Since Brisbane is in the southern hemisphere, you can observe that September - March temperatures are significantly higher than the temperatures during April-August.

Listing 4.20: Radial plot

```
1  using DataFrames, CSV, Dates, StatsBase, Plots, TimeSeries; pyplot()
2
3  data = CSV.read("../data/temperatures.csv",copycols = true)
4  brisbane = data.Brisbane
5  dates = [Date(
6              Year(data.Year[i]),
7              Month(data.Month[i]),
8              Day(data.Day[i])
9          ) for i in 1:nrow(data)]
10
11 window1, window2 = 7, 14
12 d1 = values(moving(mean,TimeArray(dates,brisbane),window1))
13 d2 = values(moving(mean,TimeArray(dates,brisbane),window2))
14
15 grid = (2pi:-2pi/365:0) .+ pi/2
16 monthsNames = Dates.monthname.(dates[1:31:365])
17
18 plot(grid, d1[1:365],
19     c=:blue, proj=:polar, label="Brisbane weekly average temp.")
20 plot!(grid, d2[1:365],
21     xticks=([mod.((11pi/6:-pi/6:0) .+ pi/2,2pi) ;], monthsNames),
22     c=:red, proj=:polar,
23     label="Brisbane fortnightly average temp.", legend=:outerbottom)
```

Lines 3-9 are similar to the previous listing setting up `brisbane` as an array of temperature readings and `dates` as an array of dates. In line 11 we define `window1` and `window2` which specify the width of the moving average smoothing to be performed. Then lines 12-13 use several functions from the `TimeSeries` package to perform moving average smoothing. We first create `TimeArray` objects, we then use the `moving()` function with first argument `mean`, we then extract the values using the `values()` function. The results are in the arrays `d1` and `d2`. In line 15 we specify the polar plotting grid. Note the use of `.+ pi/2` shifting the range by 90 degrees. In line 16 we use the `monthname()` function from package `Dates` to get an array of month names for labels. The radial plots are generated in lines 18-23 using the argument `proj=:polar`. Notice the specification of `xticks` in line 21 where we broadcast the `mod()` function with second argument `2pi`. This ensures all angles are standardized to lie in the interval $[0, 2\pi]$.

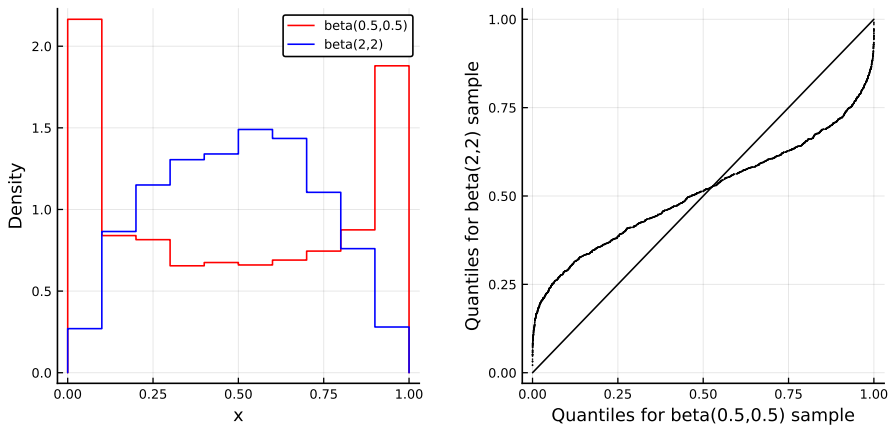

Figure 4.8: Left: Samples from two beta distributions. Right: A Q-Q plot comparing the samples.

4.4 Plots for Comparing Two or More Samples

Having covered plots for single sample data, we now introduce plots that are primarily designed for comparing two or more samples. As described at the start of this chapter, in the case of two samples, we denote the data $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$.

Quantile-Quantile (Q-Q) Plot

The *Quantile-Qunatile* or *Q-Q plot* checks if the distributional shape of two samples is the same or not. For this plot we require that that the sample sizes are the same. Then the ranked quantiles of the first sample are plotted against the ranked quantile of the second sample. While mathematically a Q-Q plot is a parametric curve, in practice since sample sizes are finite, the points plotted are according to the points of the first sample - on the horizontal axis. In the case where the samples have a similar distributional shape, the resulting plot appears like a collection of increasing points along a straight line. However, in cases where the distributional shape varies, other patterns appear. Hence, Q-Q plots serve as a mechanism to compare the distributional shapes of two samples.

A different variant of Q-Q plots is when the quantiles of a single sample are plotted against the quantiles of a theoretical distribution. One such plot is the normal probability plot covered in the previous section. In general, one can create such a plot of a single sample against any theoretical distribution. Refer to the documentation of `qqplot()` from `StatsPlots` for more information. Another variant is the *Probability-Probability* or *P-P plot*. Here, cumulative probabilities are used on the axes instead of quantiles.

In Listing 4.21 we generate Figure 4.8 which considers two synthetic samples from beta distributions. The left plot presents histograms and the right a Q-Q plot. You may modify the parameters in the code and see how this affects the plot.

Listing 4.21: Q-Q Plots

```
1  using Random, Distributions, StatsPlots, Plots, Measures; pyplot()
2  Random.seed!(0)
3
4  b1, b2 = 0.5 , 2
5  dist1, dist2, = Beta(b1,b1), Beta(b2,b2)
6
7  n = 2000
8  data1 = rand(dist1,n)
9  data2 = rand(dist2,n)
10
11 stephist(data1, bins=15, label = "beta($b1,$b1)", c = :red, normed = true)
12 p1 = stephist!(data2, bins=15, label = "beta($b2,$b2)",
13         c = :blue, xlabel="x", ylabel="Density",normed = true)
14
15 p2 = qqplot(data1, data2, c=:black, ms=1, msw =0,
16         xlabel="Quantiles for beta($b1,$b1) sample",
17         ylabel="Quantiles for beta($b2,$b2) sample",
18         legend=false)
19
20 plot(p1, p2, size=(800,400), margin = 5mm)
```

Lines 4-5 define the distributions of the synthetic data and their parameters. Lines 7-9 create the sample data. Lines 11-13 create the histograms. Lines 15-18 call the `qqplot()` function from `StatsPlots` and create the Q-Q plot.

Box Plot

The *box plot*, also known as a *box and whisker plot*, is commonly used to visually draw conclusions of, and to compare two or more single sample datasets. It displays the first and third quartiles along with the median, i.e. the 'box', along with calculated upper and lower bounds of the data, i.e. the 'whiskers', hence the name. The location of the whiskers is typically given by

$$\text{minimum} = Q1 - 1.5IQR, \qquad \text{maximum} = Q3 + 1.5IQR,$$

where IQR is the inter-quartile range (see Section 4.2). Observations that lie outside this range are called *outliers*.

In Listing 4.22, we present an example of the box plot, where we compare three datasets. The files `machine1.csv`, `machine2.csv`, and `machine3.csv` represent sample measurements of the diameter of identical pipes produced by three different machines. The diameters of the pipes vary due to imprecision of each machine but also potentially due to variability between the machines. Statistical analysis of this example via ANOVA is presented in Chapter 7, Section 7.3. The listing produces Figure 4.9, and from this figure we can visually compare the three sample populations.

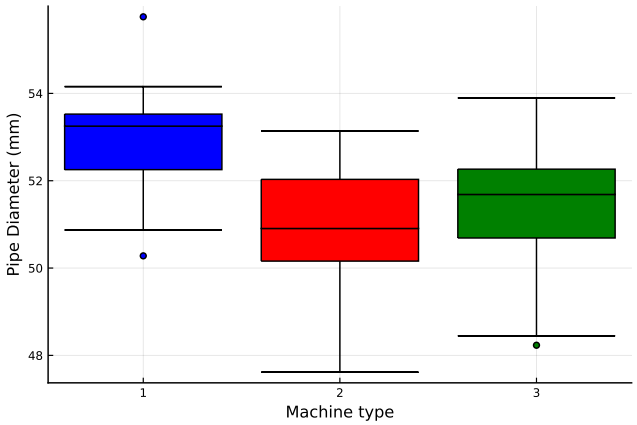

Figure 4.9: Box plots of pipe diameters associated with machines 1, 2, and 3.

Listing 4.22: Box plots of data

```
1  using CSV, StatsPlots; pyplot()
2
3  data1 = CSV.read("../data/machine1.csv", header=false)[:,1]
4  data2 = CSV.read("../data/machine2.csv", header=false)[:,1]
5  data3 = CSV.read("../data/machine3.csv", header=false)[:,1]
6
7  boxplot([data1,data2,data3], c=[:blue :red :green], label="",
8          xticks=([1:1:3;],["1", "2", "3"]), xlabel="Machine type",
9          ylabel="Pipe Diameter (mm)")
```

In lines 3-5 the data files for each of the machines are loaded and the data stored as separate arrays. In lines 7-9 the boxplot is created via the `boxplot()` function from the `StatsPlots` package.

Violin Plot

The *violin plot* is another plot that can be used to compare multiple sample populations. It is similar to the box plot, however the shape of each sample is represented by a mirrored kernel density estimate of the data. Listing 4.23 creates an example of this plot as shown in Figure 4.10. Note this example uses the `iris` dataset from the `RDatasets` package. This dataset is further explored in the next Section.

Listing 4.23: Violin plot

```
1  using RDatasets, StatsPlots
2
3  iris = dataset("datasets", "iris")
4  @df iris violin(:Species, :SepalLength,
5          fill=:blue, xlabel="Species", ylabel="Sepal Length", legend=false)
```

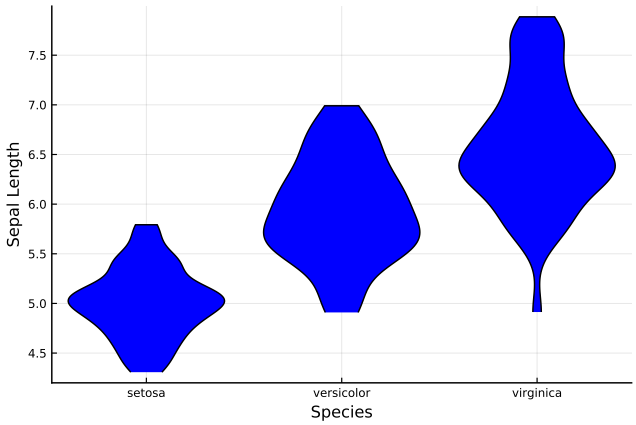

Figure 4.10: An example of a violin plot.

In line 3 the `iris` dataset from the `RDatasets` package is loaded as a `DataFrame` via the `dataset` function. The first argument, `"datasets"`, is the package in `RDatasets` which contains the `"iris"` dataset, which is the second argument. In line 4 the `@df` macro is used to plot the data from the dataframe directly, with the first argument `:Species` the horizontal axis, and the second argument `:SepalLength` the vertical axis.

4.5 Plots for Multivariate and High Dimensional Data

We now consider vectors of observations, $(x_{11}, \ldots, x_{1p}), \ldots, (x_{n1}, \ldots, x_{np})$, where n is the number of observations and p is the number of variables, or features. In cases where p is large the data is called *high dimensional*. In such cases, analysis of the data can be both challenging and rewarding. Such analysis is the focus of Chapters 8 and 9 where we focus on linear regression and machine learning. Analysis of multivariate data often goes hand in hand with visualization. Here the natural constraint is the fact that images (or plots) are limited to lie on the two dimensional plane, while in practice p is often much greater than 2 (denoted $p \gg 2$).

We have already explored several basic plots associated with multivariate data. These include the surface plot and heat map first introduced in Figure 1.8, the contour plot introduced in Figure 3.26, and the scatter plot first introduced in Figure 1.12. We augment these by introducing the *scatter plot matrix*, *heat map with marginals* plot, and *Andrews plot*. Also related are plots generated by PCA as that presented in Figure **??** in Chapter 9.

Note that with the exception of a basic animation example presented in Listing 1.11 in Chapter 1, we don't cover advanced animation methods, sound generation, interactive graphics or 3D printing. Nevertheless, the reader should keep in mind that when properly used, all these forms of media allow one to better visualize high dimensional data. This is still an emerging field and is bound to take on a more prominent role in the coming years. To this end, you may be interested in exploring a growing and diverse set of Julia packages including `PlotLy.jl`, `VegaLite.jl`, and others.

Scatter Plot Matrix

The basic *scatter plot* is very common in data visualization. In its simplest form, when considering pairs of observations $(x_1, y_1), \ldots, (x_n, y_n)$ it is a plot of these coordinates on the *Cartesian plain*. If in addition, the observations are labelled where each pair (x_i, y_i) has a label from a small finite set, then each point can be colored or marked with a symbol, matching the label. See for example Figure 3.25 from Chapter 3 as one of many examples of this type of plot.

When moving from pairs to higher dimensions, each observation is represented as the vector or tuple $(x_{i1}, \ldots, x_{ip})$. If $p = 3$ one may still try to illustrate a *point cloud*, however for higher dimensions this isn't possible. In this case, one of the most popular plots for visualizing relationships is the *scatter plot matrix*. It consists of taking each possible pair of variables and plotting a scatter plot for that pair. This allows one to understand relationships between pairs of variables. With p variables there are p^2 total plots, where p of the plots are redundant because they plot a variable against itself (on the diagonal), and the other $p^2 - p$ plots each contain a duplicate of the plots (with the axis reversed). Hence for example if $p = 4$ there are $(p^2 - p)/2 = 6$ important plots in the scatter plot matrix even though the 4×4 matrix has 16 plots in total.

As an example, Listing 4.24 creates Figure 4.11 where we consider the `iris` data set that consists of four measurements for each flower: 'sepal width', 'sepal length', 'petal width', and 'petal length'. Hence each flower can be considered as a tuple $(x_{i1}, x_{i2}, x_{i3}, x_{i4})$. As with any scatter plot, data in scatter plot matrices can also be colored or labeled. In this example, there are 3 species, 'setosa', 'veriscolor', and 'virginica', and each tuple is associated with a species. The listing output also summarizes basic information about the `iris` dataset. Inspection of Figure 4.11 can yield insight and conjectures about the population of flowers.

Listing 4.24: Scatterplot matrix

```
using RDatasets, Plots, Measures; pyplot()

data = dataset("datasets", "iris")
println("Number of rows: ", nrow(data))

insertSpace(name) = begin
    i = findlast(isuppercase,name)
    name[1:i-1]*" "*name[i:end]
end

featureNames = insertSpace.(string.(names(data)))[1:4]
println("Names of features:\n\t", featureNames)

speciesNames = unique(data.Species)
speciesFreqs = [sn => sum(data.Species .== sn) for sn in speciesNames]
println("Frequency per species:\n\t", speciesFreqs)

default(msw = 0, ms = 3)

scatters = [
    scatter(data[:,i], data[:,j], c=[:blue :red :green], group=data.Species,
        xlabel=featureNames[i], ylabel=featureNames[j], legend = i==1 && j==1)
    for i in 1:4, j in 1:4 ]

plot(scatters..., size=(1200,800), margin = 4mm)
```

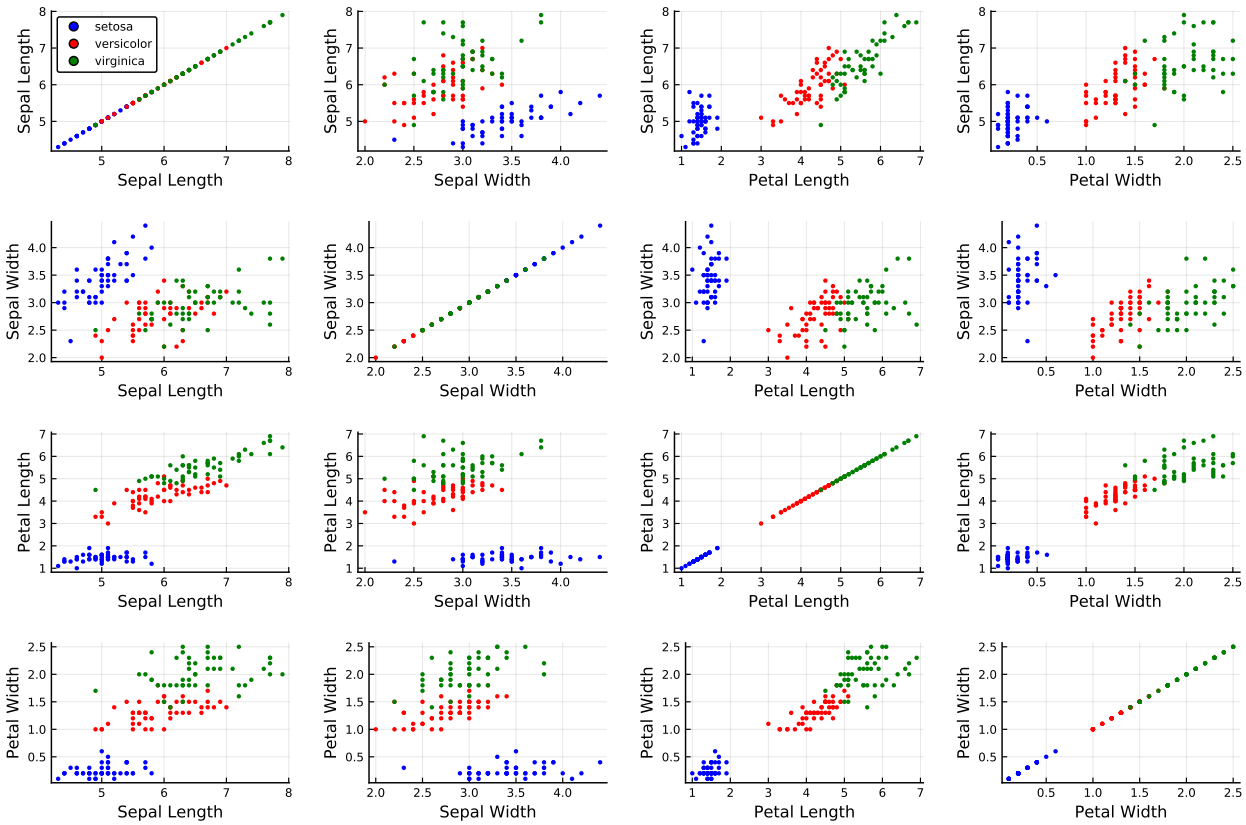

Figure 4.11: A scatter plot matrix of the iris dataset with observations grouped by species. Blue is Setosa, red is versicolor, green is virginica.

```
Number of rows: 150
Names of features:
        ["Sepal Length", "Sepal Width", "Petal Length", "Petal Width", " Species"]
Frequency per species:
        Pair{String,Int64}["setosa"=>50, "versicolor"=>50, "virginica"=>50]
```

In line 3 we create the data frame and in line 4 we print the number of rows in it. In lines 6-9 we define a function that takes a string, `name`, that is assumed to be of the form `"SepalWidth"` as an example. Such are the names of columns in the `iris` dataset. The function then inserts white space prior to the last capital letter so as to convert the string to `"Sepal Width"`. Notice the use of string concatenation using `*` in line 8. We then use this function in line 11 to create `featureNames`, an array of strings that is later used to label the variables. Note the use of `names()` in line 11, yielding an array of symbols. Lines 14-16 deal with the species and their frequency. The names of species are obtained in line 14 and their frequency is obtained in line 15. This is simply for purposes of summarizing these results in the output generated in line 16. In line 18 we use the `default()` function from `Plots` to set parameters used by all scatter plots. In line 20 we create a matrix of scatter plots. Note the use of `group=` in line 21 based on species. Also note the condition in line 22 for presenting a legend only in the top left plot. The plots are then presented in a figure in line 25.

Heat Map with Marginals

The *heat map*, first seen in 1.8, consists of a grid of shaded cells. Another name for it is a *matrix plot*. The colors of the cells indicate the magnitude, where typically, the 'warmer' the color, the higher the value. This is in a sense nothing but a monochrome image.

In cases of pairs of observations $(x_1, y_1), \dots (x_n, y_n)$, the bivariate data can be constructed into a *bivariate histogram* in a manner similar to the (univariate) histogram implemented in Listing 4.14. In the bivariate case, we partition the Cartesian plain (or the subset containing the data) $\mathbb{R}^2$, into a grid of bins $\mathcal{B}_{ij}$ for $i = 1, \dots, L_1$ and $j = 1, \dots, L_2$. Then we count the frequency of observations per bin via,

$$f_{ij} = \frac{1}{n} \sum_{k=1}^{n} \mathbf{1}\{x_k \in \mathcal{B}_{ij}\}, \qquad \text{for} \quad i = 1, \dots, L_1, \qquad j = 1, \dots, L_2. \tag{4.10}$$

Compare this with (4.6) dealing with the univariate case. Now the $L_1 \times L_2$ matrix composed of f_{ij} can be plotted as a heat map to yield a bivariate histogram.

The `marginalhist()` function from `StatsPlots` implements this and goes even further to create and present *marginal histograms*. These are two separate univariate histograms, one for $x_1, \dots, x_n$ and the other for $y_1, \dots, y_n$. Then, as shown in Figure 4.12, these histograms are presented on the margins of the heat maps, estimating the marginal distributions. See also Section 3.7.

In Listing 4.25 we create Figure 4.12 that presents two variants of a heat map with marginals. The left plot is for the Brisbane and Gold Coast temperature data, also used in Listings 4.12, Listing 3.34, as well as others. The right plot is for synthetic data based on a bivariate normal distribution fitted to that data, with the actual parameters fit in Listing 4.12. Note that this data is also plotted as a time-series in Figure 4.6. Hence in interpreting it via a histogram, one needs to exercise caution due to the cyclic nature of the data.

Listing 4.25: Heatmap and marginal histograms

```
using StatsPlots, Distributions, CSV, DataFrames, Measures; pyplot()

realData = CSV.read("../data/temperatures.csv")

N = 10^5
include("../data/mvParams.jl")
biNorm = MvNormal(meanVect,covMat)
syntheticData = DataFrame(rand(MvNormal(meanVect,covMat),N)')
rename!(syntheticData,[:x1=>:Brisbane, :x2 => :GoldCoast])

default(c=cgrad([:blue, :red]),
    xlabel="Brisbane Temperature",
    ylabel="Gold Coast Temperature")

p1 = marginalhist(realData.Brisbane, realData.GoldCoast, bins=10:45)
p2 = marginalhist(syntheticData.Brisbane, syntheticData.GoldCoast, bins=10:.5:45)

plot(p1,p2, size = (1000,500), margin = 10mm)
```

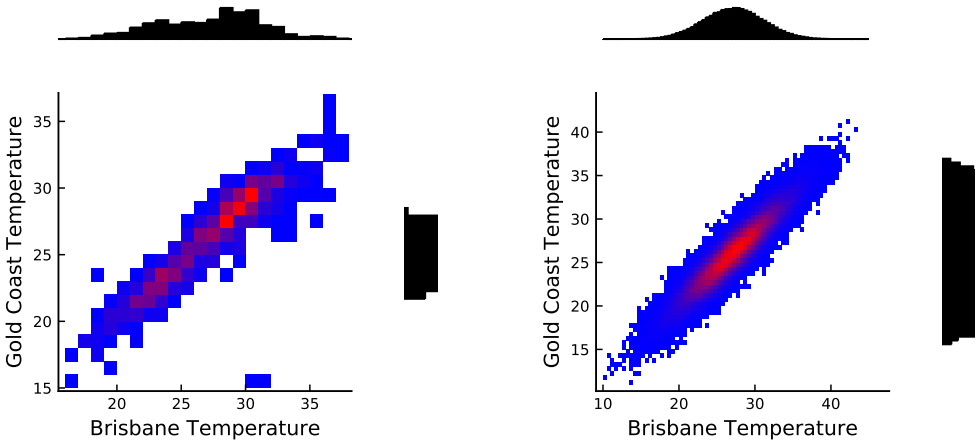

Figure 4.12: Heat map with marginals comparing Brisbane and Gold Coast temperatures. Left: actual data. Right: synthetic multivariate normal data.

In line 3 we create `realData` based on the Brisbane and Gold Coast temperature file. Lines 5-9 create the `syntheticData` data frame with `N` observations based on the bivariate normal distribution `biNorm` using the parameters in `mvParams.jl` similarly to Listing 3.34. The actual creation of the `DataFrame` object in line 8 creates default column names, `x1` and `x2`. We then rename these in line 9. The remainder of the code creates the two heat maps with marginals plots using `marginalhist()` in lines 15-16. Observe that for the synthetic data we are able to use a much larger number of bins. Note the use of the `cgrad()` function in line 11, setting the color gradient as part of the default parameters.

Andrews Plot

We now introduce a completely different way to visualize high-dimensional data. The idea is to represent a data vector $(x_{i1}, \ldots, x_{ip})$ via a real valued function. For any individual vector, such a transformation cannot be generally useful, however when comparing groups of vectors, it may yield a way to visualize structural differences in the data.

The specific transformation rule that we present here creates a plot known as *Andrews plot*. Here for the i'th data vector $(x_{i1}, \ldots, x_{ip})$ we create the function $f_i(\cdot)$ defined on $[-\pi, \pi]$ via,

$$f_i(t) = \frac{x_{i1}}{\sqrt{2}} + x_{i2}\sin(t) + x_{i3}\cos(t) + x_{i4}\sin(2t) + x_{i5}\cos(2t) + x_{i6}\sin(3t) + x_{i7}\sin(3t) + \cdots,$$

with the last term involving a sin() if p is even and a cos() if p is odd. The for $i = 1, \ldots, n$, the functions $f_1(\cdot), \ldots, f_n(\cdot)$ are plotted. In cases where each i has an associated label from a small finite set, different colors or line patterns can be used. An example of this plot is shown in Figure 4.13, where the results hint at similarities within species and differences between species.

In Listing 4.26 below, we present a standard example of Andrews plot based on the `iris` dataset. The resulting Figure 4.13 indicates that differences exist between each of the species.

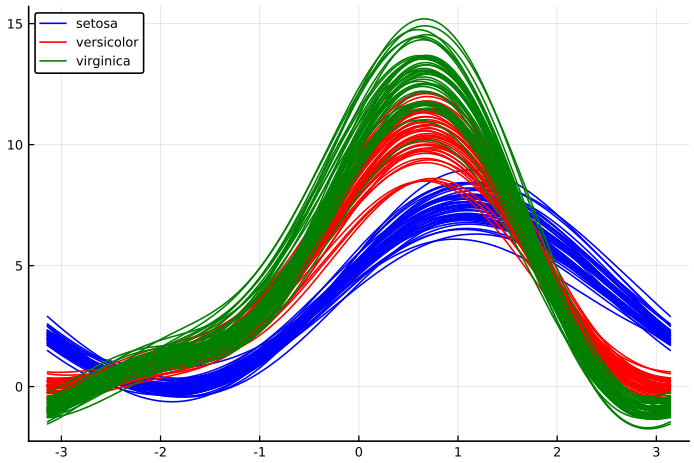

Figure 4.13: Andrews plot, showing underlying structure in the Iris dataset.

Listing 4.26: Andrews plot

```
1  using RDatasets, StatsPlots; pyplot()
2
3  iris = dataset("datasets", "iris")
4  @df iris andrewsplot(:Species, cols(1:4),
5          line=(fill=[:blue :red :green]), legend=:topleft)
```

In line 4 the `andrewsplot()` function from `StatsPlots` is used to plot the data. Note the `@df` macro is used in a similar format to that of Listing 4.23. The first argument, `:Species`, determines how the data should be grouped, while the second argument determines what variables should be included in the calculation, in this case columns 1 to 4.

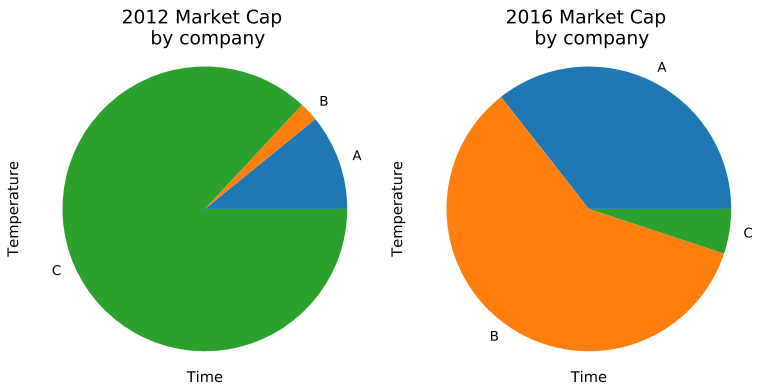

Figure 4.14: Two pie charts.

4.6 Plots for the Board Room

In this section we introduce more simple plots, such as those that one may typically see in business summaries, or news reports. We show how to create *pie charts*, *bar charts*, and *stack plots*. Although the plots covered here are not as technical as those covered previously, they are still useful as they can quickly convey information in a very clear manner. The examples in this section rely on data for three fictitious companies, stored in `companyData.csv`.

Pie Chart

We first look at the *pie chart*, which is a simple plot that conveys relative proportions. In Listing 4.27 we construct two pie charts which show the relative market capitalization of each company A, B, and C for the years 2012 and 2016. The results are shown in Figure 4.14.

Listing 4.27: A pie chart

```
1  using CSV, CategoricalArrays, Plots; pyplot()
2
3  df = CSV.read("../data/companyData.csv")
4  companies = levels(df.Type)
5
6  year2012 = df[df.Year .== 2012, :MarketCap]
7  year2016 = df[df.Year .== 2016, :MarketCap]
8
9  p1 = pie(companies, year2012, title="2012 Market Cap \n by company")
10 p2 = pie(companies, year2016, title="2016 Market Cap \n by company")
11 plot(p1, p2, size=(800, 400))
```

In line 4 `levels()` from the `CategoricalArrays` package is used to extract the name of each company as a level, and store them in the array `companies`. In lines 6-7 the market capitalization for each company is stored as arrays `year2012` and `year2016` for the years 2012 and 2016 respectively. In lines 9-10 the `pie()` function is used to create the pie charts.

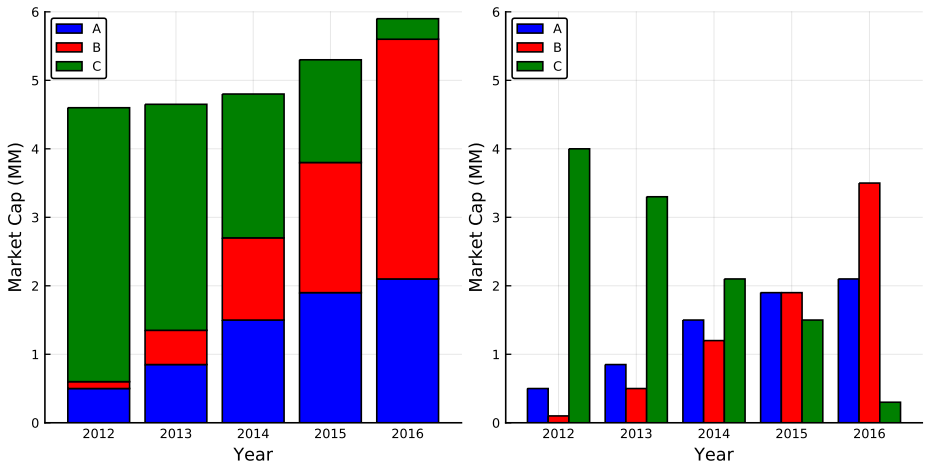

Figure 4.15: A stacked bar plot (left), and non-stacked bar plot (right).

Bar Plot

The *bar plot*, or *bar chart*, is another useful plot which conveys proportions through the use of vertical bars. This plot was first seen in Figure 3.13, and we present another example of this plot here. Listing 4.28 summarizes the data from `companyData.csv`, and presents the total market capitalization of each company for each year through a *stacked bar plot* and a *grouped bar plot*. The results are shown in Figure 4.15.

Listing 4.28: Two different bar plots

```
1  using CSV, CategoricalArrays, StatsPlots; pyplot()
2
3  df = CSV.read("../data/companyData.csv")
4  years = levels(df.Year)
5  data  = reshape(df.MarketCap, 5, 3)
6
7  p1 = groupedbar(years, data, bar_position=:stack)
8  p2 = groupedbar(years, data, bar_position=:dodge)
9  plot(p1, p2, bar_width=0.7, fill=[:blue :red :green], label=["A" "B" "C"],
10         ylims=(0,6), xlabel="Year", ylabel="Market Cap (MM)",
11         legend=:topleft, size=(800,400))
```

In line 5 `reshape()` is used to reshape the market capitalization data from a single column to a 5×3 array, with the rows representing years and columns companies. In lines 7-11 the `groupedbar()` function from `StatsPlots` is used to create the bar plots. By setting `bar_position=:stack`, a stackplot is created, while `bar_position=:dodge` creates a grouped bar plot instead.

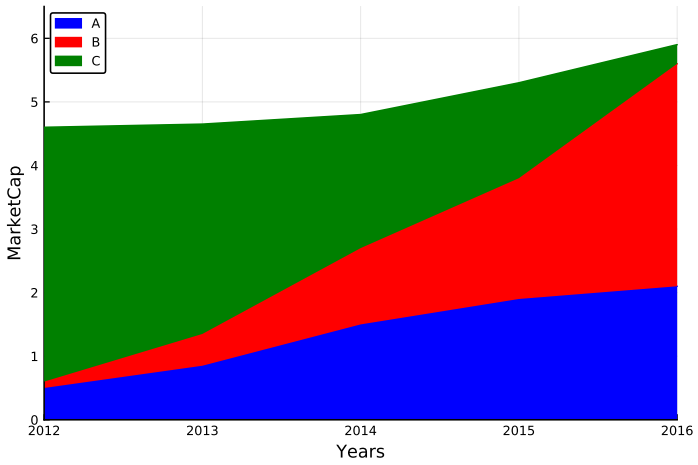

Figure 4.16: A stack plot showing the change in market capitalization of several companies over time.

Stack Plot

The *stack plot* is a commonly used plot which shows how constituent amounts of a metric change over time. In Listing 4.29 we present an example, where we consider the changing total market capitalization of the three companies A, B, and C over several years.

Listing 4.29: A stack plot

```
1  using CSV, CategoricalArrays, Plots; pyplot()
2
3  df = CSV.read("../data/companyData.csv")
4  mktCap = reshape(df.MarketCap, 5, 3)
5  years  = levels(df.Year)
6
7  areaplot(years, mktCap,
8          c=[:blue :red :green], labels=["A" "B" "C"],
9          xlims=(minimum(years),maximum(years)), ylims=(0,6.5),
10         legend=:topleft, xlabel="Years", ylabel="MarketCap")
```

In line 4 the data in the `MarketCap` column is reshaped into a 5×3 array via the `reshape()` function. In line 5 `levels()` is used to store the unique years of the dataset in the array `years` in ascending order. In lines 7-10 `areaplot()` is used to create the plot, with the horizontal values given as the first argument, and the data to be plotted as the second argument, with rows treated as individual years.

4.7 Working with Files and Remote Servers

The ability to work with files is an important skill that the modern data scientist is expected to have. Often one will be required to perform various operations with files programatically, such as create new files, open files, interact with their content, and save information to existing files.

Julia provides various methods to interact with and work with files via *input/output streams* (I/O). In this section we provide two simple examples which involve working with files programatically. In addition, at the end of this section, we provide a brief pseudo-code example of how one might connect to a remote server, query a database, and save the results to a locally stored file.

Searching a File for Keywords

For a first example we show how one might programatically search a file for a specific keyword or content, and then save that content to a separate file. In Listing 4.30 we create a function which searches a text document for a given keyword, and then saves every line of text containing this keyword to a new text file, along with the associated line number.

Listing 4.30: Filtering an input file

```
1  function lineSearch(inputFilename, outputFilename, keyword)
2      infile  = open(inputFilename, "r")
3      outfile = open(outputFilename,"w")
4
5      for (index, line) in enumerate(split(read(infile, String), "\n"))
6          if occursin(keyword, line)
7              println(outfile, "$index: $line")
8          end
9      end
10     close(infile)
11     close(outfile)
12 end
13
14 lineSearch("../data/earth.txt", "../data/waterLines.txt", "water")
```

```
17: 71% of Earth's surface is covered with water, mostly by oceans. The
19: have many lakes, rivers and other sources of water that contribute to the
```

In lines 1-12 the function `lineSearch()` is defined, which searches an input file, `inputFilename`, for a `keyword`, and saves the lines and line numbers where it appears to an output file, `outputFilename`. Line 2 uses `open()` with 'r' to open the file to be searched in read mode. It creates an `IOStream` object, which can be used as arguments to other functions. We define this as the variable `infile`. Line 3 uses `open()` with 'w' to create and open a file in write mode, with the given file name `outputFilename`. This file is created on disk ready to have information written to it. Lines 5-9 contain a for loop, which is used to search through the input file for the given `keyword`. Line 5 reads the file as a `String` via `read()`, and `split()` is used along with '\n' to convert the single string into an array of strings, where the content of each line is stored in a separate index of the array. Line 6 uses `occursin()` to check if the given `line` contains our given `keyword`. If it does, then we proceed to line 7, where `println()` is used to write both the `index` and the `line` content to the `outfile`. Lines 10 and 11 close both our input file and output file. In line 14 `lineSearch` is used to search the file '`earth.txt`', for the keyword '`water`', with the line numbers and text saved to the file `waterLines.txt`'.

Searching for Files in a Directory

In Listing 4.31 we present our next example, where we create a function which searches a directory for all filenames which contain a particular string. It then saves a list of these files to a file, `fileList`. Note that this function does not behave recursively and only searches the directory given.

Listing 4.31: Searching files in a directory

```
1  function directorySearch(directory, searchString)
2      outfile  = open("fileList.txt","w")
3      fileList = filter(x->occursin(searchString, x), readdir(directory))
4
5      for file in fileList
6          println(outfile, file)
7      end
8      close(outfile)
9  end
10
11 directorySearch(pwd(),".jl")
```

In lines 1-9 we define the function `directorySearch`. As arguments, it takes a `directory` to search through, and a `searchString`. Line 2 uses `open` with 'w' to create our output file `fileList.txt`, which we will write to. In line 3 we create a string array of all filenames in our specified `directory` that contain our `searchString`. This string array is defined as the variable `fileList`. The `readdir()` function is used to list all files in the given `directory`, and `filter()` is used, along with `occursin()` to check each element contains the `searchString`. Lines 5-7 loop through each element in `fileList` and print them to theoutput file `outfile`. Line 8 closes the `IOStream` `outfile`. Line 11 provides an example of the use of our `directorySearch` function, where we use it to obtain a shortlist of all files whose extensions contain "`.jl`" within our current working directory, i.e. `pwd()`.

Connecting to a Remote Server

One may not always work with data stored locally on their machine or network. For example, sometimes a dataset is too large to be stored on a workstation, and therefore must be stored remotely in a datacentre, or on a *remote server*. In this scenario one must first connect to the server before working with the data. A typical workflow involves connecting to the remote database, submitting a query, and then saving the result locally. There are different types of databases, including: Oracle, MySQL, PostgreSQL, MongoDB, and many others. There are several Julia packages for connecting to remote servers including `LibPQ.jl`, which is a wrapper for the PostgreSQL libpq C library, `SQLite.jl`, as well as `ODBC.jl` and several others. Once a connection is established, one will typically submit a so-called *SQL* query to the server. SQL stands for strucutred query language, and is a common syntax used to query remote databases in order to extract a subsect of data from the database.

In this section we do not expand on the details of databases, nor the syntax of SQL queries. Instead, in Listing 4.32 we present a simple pseudocode example of how a user may connect to a

remote PostgreSQL database, sumbit a SQL query, and then save the results.

Listing 4.32: Pseudocode for a remote database query

```
1  using LibPQ, DataFrames, CSV
2
3  host     = "remoteHost"
4  dbname   = "db1"
5  user     = "username"
6  password = "userPwd"
7  port     = "1111"
8
9  conStr= "host=" *host *
10         " port=" *port *
11         " dbname=" *dbname *
12         " user=" *user *
13         " password=" *password
14 conn = LibPQ.Connection(conStr)
15
16 df = DataFrame(execute(conn, "SELECT * FROM S1.T1"))
17 close(conn)
18
19 CSV.write("example.csv", df);
```

In line 1 the `LibPQ` package is included. It is a wrapper for the libpq postgreSQL library, and contains methods to remotely connect to postgreSQL servers and submit queries. In lines 3-7 the details of the connection are specified and stored as strings, they include the: host name, database name, username, password, and specific port to connect on. Lines 9-13 concatenate these details together into the string `conStr`. In line 14 a connection to the remote server is established via the `Connection()` function from the `LibPQ` package. The details in the string `conStr` are used to establish the connection. Note that if the password is not given in the connection string, then the server will prompt for a password. In line 16 a SQL query is submitted to the server via the `execute()` function. It takes two arguments, the first is the connection to the server, and the second is the SQL query. The query submitted here is simple: `SELECT *` is used to select all columns `FROM` the `T1` table, from the `S1` schema, from database `db1`. The results are stored as the `DataFrame df`. The connection to the server is closed in line 17 via `close()`. In line 19 the data in `df` is written to the CSV file `example.csv`, in the current working directory.

Chapter 5

Statistical Inference Concepts - DRAFT

This chapter introduces statistical inference concepts, with the goal of establishing a theoretical footing of key concepts that follow in later chapters. The approach is that of classical statistics as opposed to machine learning, covered in Chapter 9. The action of *statistical inference* involves using mathematical techniques to make conclusions about unknown *population* parameters based on collected data. The field of statistical inference employs a variety of stochastic models to analyze and put forward efficient methods for carrying out such analyses.

In broad generality, analysis and methods of statistical inference can be categorized as either *frequentist* (also known as classical) or *Bayesian*. The former is based on the assumption that population parameters of some underlying distribution, or probability law, exist and are fixed, but are yet unknown. The process of statistical inference then deals with making conclusions about these parameters based on sampled data. In the latter Bayesian case, it is only assumed that there is a *prior distribution* of the parameters. In this case, the key process deals with analyzing a *posterior distribution* (of the parameters) - an outcome of the inference process. In this book we focus almost solely on the classical frequentist approach with the exception of Section 5.7 where we explore Bayesian statistics briefly.

In general, a statistical inference process involves *data*, a *model*, and *analysis*. The data is assumed to be comprised of random samples from the model. The goal of the analysis is then to make informed statements about population parameters of the model based on the data. Such statements typically take one of the following forms:

Point estimation - Determination of a single value (or vector of values) representing a best estimate of the parameter/parameters. In this case, the notion of "best" can be defined in different ways.

Confidence intervals - Determination of a range of values where the parameter lies. Under the model and the statistical process used, it is guaranteed that the parameter lies within this range with a pre-specified probability.

Hypothesis tests - The process of determining if the parameter lies in a given region, in the complement of that region, or fails to take on a specific value. Such tests often represent a scientific hypothesis in a very natural way.

Most of the point estimation, confidence intervals and hypothesis tests that we introduce and carry out in this book are elementary. Chapter 6 is devoted to covering elementary confidence intervals in detail, and Chapter 7 is devoted to covering elementary hypothesis tests in detail. We now begin to explore key ideas and concepts of statistical inference.

This chapter is structured as follows: In Section 5.1 we present the concept of a random sample together with the distribution of statistics, such as the distribution of the sample mean and the sample variance. In Section 5.2 we focus on random samples of normal random variables. In this common case, certain statistics have well known distributions that play a central role in statistics. In Section 5.3 we explore the central limit theorem, providing justification for the ubiquity of the normal distribution. In Section 5.4 we explore basics of point estimation. In Section 5.5 we explore the concept of a confidence interval. In Section 5.6 we explore concepts of hypothesis testing. Finally, in Section 5.7 we explore the basics of Bayesian statistics.

5.1 A Random Sample

When carrying out (frequentist) statistical inference, we assume there is some underlying distribution $F(x\,;\,\theta)$ from which we are sampling, where θ is the scalar or vector-valued unknown parameter we wish to know. Importantly, we assume that each observation is statistically independent and identically distributed as the rest. That is, from a probabilistic perspective, the observations are taken as *independent and identically distributed (i.i.d.)* random variables. In mathematical statistics language, this is called a *random sample*. We denote the random variables of the observations by $X_1, \ldots, X_n$ and their respective values by $x_1, \ldots, x_n$.

Typically, we compute *statistics* from the random sample. For example, two common standard statistics include the *sample mean* and *sample variance*, introduced in Section 4.2 in the context of data summary. However, we can model these *statistics* as random variables,

$$\overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i, \qquad \text{and} \qquad S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \overline{X})^2. \tag{5.1}$$

Note that for S^2, the denominator is $n-1$ (as opposed to n as one might expect). This makes S^2 an *unbiased estimator*. We discuss this property further in Section 5.4.

In general, the phrase *statistic* implies a quantity calculated based on the sample. When working with data, the sample mean and sample variance are nothing but numbers computed from our sample observations. However, in the statistical inference paradigm, we associate random variables to these values, since they themselves are functions of the random sample. We look at properties of such statistics, and see how they play a role in estimating the unknown underlying distribution parameter θ.

To illustrate the fact that $\overline{X}$ and S^2 are random variables, assume we have sampled data from an exponential distribution with $\lambda = 4.5^{-1}$ (a mean of 4.5 and a variance of 20.25). If we collect $n = 10$ observations, then the sample mean and sample variance are random variables. In Listing 5.1, we investigate their distribution through Monte Carlo simulation and create Figure 5.1. The point to see is that $\overline{X}$ and S^2 are themselves random variables with underlying distributions.

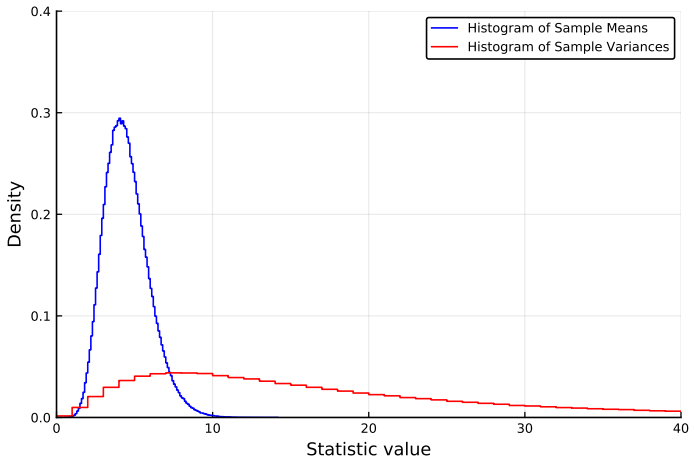

Figure 5.1: Histograms of the sample mean and sample variance of an exponential distribution.

Listing 5.1: Distributions of the sample mean and sample variance

```
using Random, Distributions, Plots; pyplot()
Random.seed!(0)

lambda  = 1/4.5
expDist = Exponential(1/lambda)
n, N    = 10, 10^6

means     = Array{Float64}(undef, N)
variances = Array{Float64}(undef, N)

for i in 1:N
    data = rand(expDist,n)
    means[i] = mean(data)
    variances[i] = var(data)
end

println("Actual mean: ",mean(expDist),
                "\nMean of sample means: ",mean(means))
println("Actual variance: ",var(expDist),
        "\nMean of sample variances: ",mean(variances))

stephist(means, bins=200, c=:blue, normed=true,
        label="Histogram of Sample Means")
stephist!(variances, bins=600, c=:red, normed=true,
        label="Histogram of Sample Variances", xlims=(0,40), ylims=(0,0.4),
        xlabel = "Statistic value", ylabel = "Density")
```

```
Actual mean: 4.5
Mean of sample means: 4.500154606762812
Actual variance: 20.25
Mean of sample variances: 20.237117004185237
```

In lines 8-9 we initialize the empty arrays `means` and `variances` respectively. In lines 11-15 we create `N` random samples, each of length `n`. For each sample we calculate the sample mean and sample variance. In lines 17-20, we calculate the `mean()` of both arrays `means` and `variances`. It can be seen that the estimated expected value of our simulated data are good approximations to the mean and variance parameters of the underlying exponential distribution. That is, for an exponential distribution with rate λ the mean is λ^{-1} and the variance is λ^{-2}. In lines 22-26, we generate histograms of the sample means and sample variances, using `200` and `600` bins respectively.

5.2 Sampling from a Normal Population

It is often assumed that the distribution $F(x\,;\,\theta)$ is a normal distribution, and hence $\theta = (\mu, \sigma^2)$. This assumption is called the *normality assumption*, and is sometimes justified due to the central limit theorem, which we cover in Section 5.3. Under the normality assumption, the distribution of the random variables $\overline{X}$ and S^2 as well as transformations of them are well known. The following three distributional relationships play a key role:

$$\begin{aligned} \overline{X} &\sim \text{Normal}(\mu, \sigma^2/n), \\ (n-1)S^2/\sigma^2 &\sim \chi^2_{n-1}, \\ T := \frac{\overline{X} - \mu}{S/\sqrt{n}} &\sim t_{n-1}. \end{aligned} \tag{5.2}$$

Here '$\sim$' denotes '*distributed as*', and implies that the statistics on the left hand side of the '$\sim$' symbols are distributed according to the distributions on the right hand side. The notation χ^2_{n-1} and t_{n-1} denotes a *chi-squared* and *student T-distribution* respectively, each with $n-1$ degrees of freedom. The chi-squared distribution is a gamma distribution (see Section 3.6) with parameters $\lambda = 1/2$ and $\alpha = n/2$. The student T-distribution is discussed later in this section.

Importantly, these distributional properties of the statistics from a normal sample theoretically support the statistical procedures that are presented in Chapters 6 and 7.

We now look at an example in Listing 5.2, where we sample data from a normal distribution and compute the statistics, $\overline{X}$, T and S^2. As seen in Figure 5.2, the distribution of sample means, sample variances and T-statistics (T) indeed follow the distributions given by (5.2).

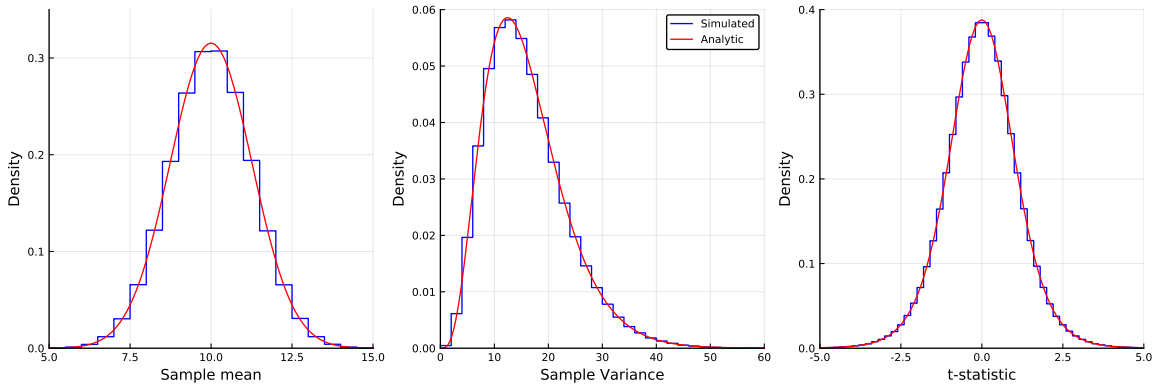

Figure 5.2: Histograms of the simulated sample means, sample variances, and T-statistics, against their analytic counterparts.

Listing 5.2: Friends of the normal distribution

```
using Distributions, Plots; pyplot()

mu, sigma = 10, 4
n, N = 10, 10^6

sMeans = Array{Float64}(undef, N)
sVars  = Array{Float64}(undef, N)
tStats = Array{Float64}(undef, N)

for i in 1:N
    data       = rand(Normal(mu,sigma),n)
    sampleMean = mean(data)
    sampleVars = var(data)
    sMeans[i]  = sampleMean
    sVars[i]   = sampleVars
    tStats[i]  = (sampleMean - mu)/(sqrt(sampleVars/n))
end

xRangeMean = 5:0.1:15
xRangeVar = 0:0.1:60
xRangeTStat = -5:0.1:5

p1 = stephist(sMeans, bins=50, c=:blue, normed=true, legend=false)
p1 = plot!(xRangeMean, pdf.(Normal(mu,sigma/sqrt(n)), xRangeMean),
    c=:red, xlims=(5,15), ylims=(0,0.35), xlabel="Sample mean",ylabel="Density")

p2 = stephist(sVars, bins=50, c=:blue, normed=true, label="Simulated")
p2 = plot!(xRangeVar, (n-1)/sigma^2*pdf.(Chisq(n-1), xRangeVar*(n-1)/sigma^2),
    c=:red, label="Analytic", xlims=(0,60), ylims=(0,0.06),
    xlabel="Sample Variance",ylabel="Density")

p3 = stephist(tStats, bins=100, c=:blue, normed=true, legend=false)
p3 = plot!(xRangeTStat, pdf.(TDist(n-1), xRangeTStat),
    c=:red, xlims=(-5,5), ylims=(0,0.4), xlabel="t-statistic",ylabel="Density")

plot(p1, p2, p3, layout = (1,3), size=(1200, 400))
```

In line 3 we specify the parameters of the underlying normal distribution from which we sample our data. In line 4 we specify the number of samples in each group `n`, and the total number of Monte Carlo repetitions `N`. In lines 6-8 we initialize three arrays which will be used to store our sample means, variances, and T-statistics. In lines 10–17, we conduct our numerical simulation by taking `n` sample observations from the underlying normal distribution, and calculating the sample mean, sample variance, and T-statistic. This process is repeated `N` times, and the values are stored in the arrays `sMeans`, `sVars`, and `tStats` respectively. The remainder of the code creates the histograms of the sample means, sample variances, and T-statistics alongside the analytic PDF's given by (5.2). Observe the PDF of the sample mean in line 24. Observe the PDF of a scaled chi-squared distribution through the use of the `pdf()` and `Chisq()` functions in line 28. Note that the values on the x-axis and the density are both normalized by `(n-1)/sigma^2` to reflect the fact we are interested in the PDF of a scaled chi-squared distribution. Finally, observe the PDF of the T-statistic (T), which is described by a T-distribution, is plotted via the use of the `TDist()` function in line 33.

Independence of the Sample Mean and Sample Variance

We now look at a key property of the sample mean and sample variance. Consider a random sample, $X_1, \ldots, X_n$. In general, one would not expect the sample mean, $\overline{X}$ and the sample variance S^2 to be independent random variables - since both of these statistics rely on the same underlying values. For example, consider a random sample where $n = 2$, and let each X_i be Bernoulli distributed, with parameter p. The joint distribution of $\overline{X}$ and S^2 can then be computed as follows.

If both X_i's are 0, which happens with probability $(1-p)^2$, then,

$$\overline{X} = 0 \qquad \text{and} \qquad S^2 = 0.$$

If both X_i's are 1, which happens with probability p^2, then,

$$\overline{X} = 1 \qquad \text{and} \qquad S^2 = 0.$$

If one of the X_i's is 0, and the other is 1, which happens with probability $2p(1-p)$, then,

$$\overline{X} = \frac{1}{2} \qquad \text{and} \quad S^2 = 1 - 2\left(\frac{1}{2}\right)^2 = \frac{1}{2}.$$

.

Hence, as shown in Figure 5.3 the joint PMF of $\overline{X}$ and S^2 is,

$$\mathbb{P}(\overline{X} = \overline{x}, S^2 = s^2) = \begin{cases} (1-p)^2, & \text{for } \overline{x} = 0 \text{ and } s^2 = 0, \\ 2p(1-p), & \text{for } \overline{x} = 1/2 \text{ and } s^2 = 1/2, \\ p^2, & \text{for } \overline{x} = 1 \text{ and } s^2 = 1. \end{cases} \tag{5.3}$$

Furthermore, the (marginal) PMF of $\overline{X}$ is,

$$\mathbb{P}_{\overline{X}}(0) = (1-p)^2, \qquad \mathbb{P}_{\overline{X}}\left(\frac{1}{2}\right) = 2p(1-p), \qquad \mathbb{P}_{\overline{X}}(1) = p^2.$$

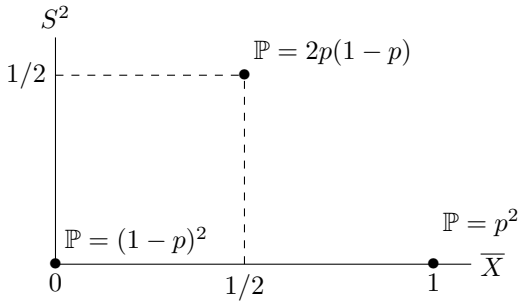

Figure 5.3: PMF of the sample mean and sample variance.

And the (marginal) PMF of S^2 is,

$$\mathbb{P}_{S^2}(0) = (1-p)^2 + p^2, \qquad \mathbb{P}_{S^2}\left(\frac{1}{2}\right) = 2p(1-p), \qquad \mathbb{P}_{S^2}(1) = 0.$$

We now see that $\overline{X}$ and S^2 are not independent because the joint distribution,

$$\tilde{\mathbb{P}}(i,j) = \mathbb{P}_{\overline{X}}(i)\,\mathbb{P}_{S^2}(j) \qquad \text{for} \qquad i,j \in \{0, \frac{1}{2}, 1\},$$

constructed by the product of the marginal distributions does not equal the joint distribution in (5.3).

The example above demonstrates dependence between $\overline{X}$ and S^2. This is in many ways unsurprising. However importantly, in the special case where the samples, $X_1, \ldots, X_n$ are from a normal distribution, independence between $\overline{X}$ and S^2 does hold. In fact, this property characterizes the normal distribution - that is, this property only holds for the normal distribution, see [Luk42].

We now explore this concept further in Listing 5.3. In it we compare a standard normal distribution to what we call a standard uniform distribution - a uniform distribution on $[-\sqrt{3}, \sqrt{3}]$ which exhibits zero mean and unit variance. For both distributions, we consider a random sample of size $n = 3$, and from this we obtain the pair $(\overline{X}, S^2)$. We then plot points of these pairs against points of pairs where $\overline{X}$ and S^2 are each obtained from two separate sample groups.

From Figure 5.4 it can be seen that for the normal distribution, regardless of whether the pair $(\overline{X}, S^2)$ is calculated from the same sample group, or from two different sample groups, the points appear to behave similarly. This is because they have the same joint distribution. However, for the standard uniform distribution, it can be observed that the points behave in a completely different manner. If the sample mean and variance are calculated from the same sample group, then all pairs of $\overline{X}$ and S^2 fall within a specific bounded region. The envelope of this blue region can be clearly observed, and represents the region of all possible combinations of $\overline{X}$ and S^2 when calculated based on the same sample data. On the other hand, if $\overline{X}$ and S^2 are calculated from two separate samples, then we observe a scattering of data, shown by the points in red. This difference in behavior shows that in this case $\overline{X}$ and S^2 are not independent, but rather the outcome of one imposes some restriction on the outcome of the other. By comparison, in the case of the standard normal distribution, regardless of how the pair $(\overline{X}, S^2)$ are calculated, (from the same sample group or from two different groups) the same scattering of points is observed, supporting the fact that $\overline{X}$ and S^2 are independent.

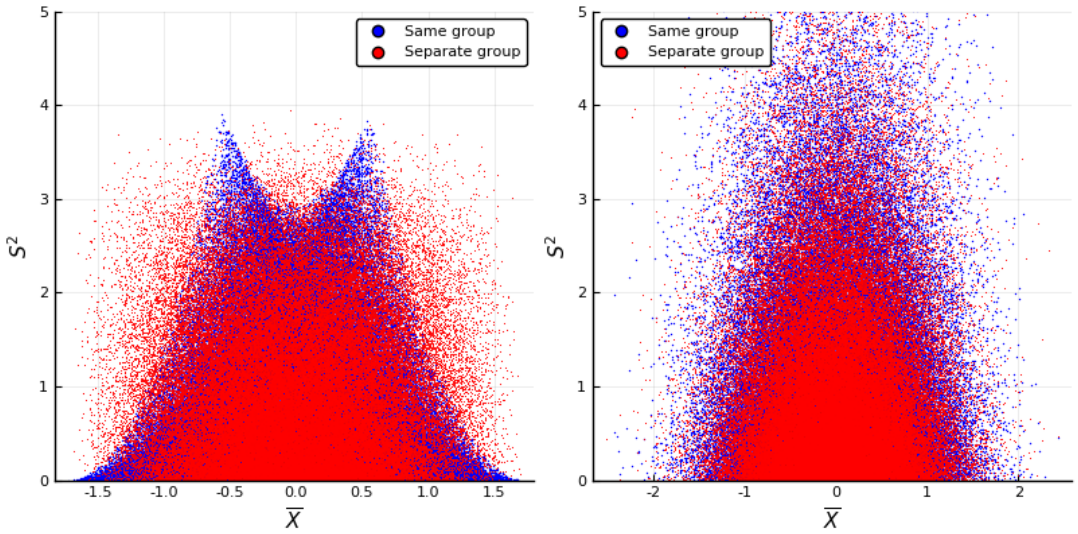

Figure 5.4: Pairs of $\overline{X}$ and S^2 for standard uniform (left) and standard normal (right). Blue points are for statistics calculated from the same sample, and red for statistics calculated from separate samples.

Listing 5.3: Are the sample mean and variance independent?

```
using Distributions, Plots, LaTeXStrings; pyplot()

function statPair(dist,n)
    sample = rand(dist,n)
    [mean(sample),var(sample)]
end

stdUni = Uniform(-sqrt(3),sqrt(3))
n, N = 3, 10^5

dataUni     = [statPair(stdUni,n) for _ in 1:N]
dataUniInd  = [[mean(rand(stdUni,n)),var(rand(stdUni,n))] for _ in 1:N]
dataNorm    = [statPair(Normal(),n) for _ in 1:N]
dataNormInd = [[mean(rand(Normal(),n)),var(rand(Normal(),n))] for _ in 1:N]

p1 = scatter(first.(dataUni), last.(dataUni),
        c=:blue, ms=1, msw=0, label="Same group")
p1 = scatter!(first.(dataUniInd), last.(dataUniInd),
        c=:red, ms=0.8, msw=0, label="Separate group",
        xlabel=L"\overline{X}", ylabel=L"S^2")

p2 = scatter(first.(dataNorm), last.(dataNorm),
        c=:blue, ms=1, msw=0, label="Same group")
p2 = scatter!(first.(dataNormInd), last.(dataNormInd),
        c=:red, ms=0.8, msw=0, label="Separate group",
        xlabel=L"\overline{X}", ylabel=L"$S^2$")

plot(p1, p2, ylims=(0,5), size=(800, 400))
```

In lines 3-6 the function `statPair()` is defined. It takes a distribution and integer `n` as input, generates a random sample of size `n`, and then returns the sample mean and sample variance of this random sample as an array. In line 8 we define the standard uniform distribution, which has a mean of zero and a standard deviation of 1. In line 9 we set the number of observations for each sample `n`, along with the total number of sample groups `N`. In line 11, the function `statPair()` is used along with a comprehension to calculate `N` pairs of sample means and variances from `N` sample groups. Note that the observations are all sampled from the standard uniform distribution `stdUni`, and that the output is an array of arrays. In line 12 a similar approach to line 11 is used. However, in this case, rather than calculating the sample mean and variance from the same sample group each time, they are calculated from two separate sample groups `N` times. As before, the data is sampled from the standard uniform distribution `stdUni`. Lines 13 and 14 are identical to lines 11-12, however in this case observations are sampled from a standard normal distribution `Normal()`.

More on the T-Distribution

Having explored the fact that $\overline{X}$ and S^2 are independent in the case of a normal sample, we now elaborate on the *Student T-distribution* and focus on the distribution of the *T-statistic*, that appeared earlier in (5.2). This random variable is given by:

$$T = \frac{\overline{X} - \mu}{S/\sqrt{n}}.$$

Denoting the mean and variance of the normally distributed observations by μ and σ^2 respectively, we can represent the T-statistic as,

$$T = \frac{\sqrt{n}(\overline{X} - \mu)/\sigma}{\sqrt{(n-1)S^2/\sigma^2(n-1)}} = \frac{Z}{\sqrt{\frac{\chi^2_{n-1}}{n-1}}}. \tag{5.4}$$

Here the numerator Z is a standard normal random variable and in the denominator the random variable, $\chi^2_{n-1} = (n-1)S^2/\sigma^2$ is chi-squared distributed with $n-1$ degrees of freedom, as claimed in (5.2). Furthermore, the numerator and denominator random variables are independent because they are based on the sample mean and sample variance.

One can show that a ratio of a standard normal random variable and the square root of a scaled independent chi-squared random variable (scaled by its degrees of freedom parameter) is distributed according to a T-distribution with the same number of degrees of freedom as the chi-squared random variable. Hence $T \sim t(n-1)$. This means a "T-distribution with $n-1$ degrees of freedom". The T-distribution is a symmetric distribution with a "bell-curved" shape similar to that of the normal distribution, with "heavier tails" for non-large n. A t-distribution with k degrees of freedom can be shown to have a density function,

$$f(x) = \frac{\Gamma\left(\frac{k+1}{2}\right)}{\sqrt{k\pi}\,\Gamma\left(\frac{k}{2}\right)}\left(1 + \frac{x^2}{k}\right)^{-\frac{k+1}{2}}.$$

Note the presence of the gamma function, $\Gamma(\cdot)$, which is defined in Section 3.6.

To gain further insight from the representation (5.4), note that $\mathbb{E}[\chi^2] = (n-1)$ and $\text{Var}(\chi^2) = 2(n-1)$. Thus the variance of $\chi^2/(n-1)$ is $2/(n-1)$, and hence one may expect that as $n \to \infty$, the random variable $\chi^2/(n-1)$ gets more and more concentrated around 1, with the same holding for $\sqrt{\chi^2/(n-1)}$. Hence for large n one may expect the distribution of T to be similar to the distribution of Z, which is indeed the case. This plays a role in the confidence intervals and hypothesis tests in the chapters that follow.

In practice, when carrying out elementary statistical inference using the T-distribution (as presented in the following chapters), the most commonly used attribute is the quantile, covered in Section 3.3. It is typically denoted by $t_{k,\alpha}$ where the *degrees of freedom* (DOF), k, define the specific T-distribution. Such quantiles are often tabulated in standard statistical tables.

In Listing 5.4 below we first illustrate the validity of the representation (5.4) by generating T-distributed random variables by using a standard normal and a chi-squared random variable. We then plot the PDFs of several T-distributions, illustrating that as the degrees of freedom increase, the PDF converges to the standard normal PDF. See Figure 5.5.

Listing 5.4: Student's T-distribution

```
1  using Distributions, Random, Plots; pyplot()
2  Random.seed!(0)
3
4  n, N, alpha = 3, 10^7, 0.1
5
6  myT(nObs) = rand(Normal())/sqrt(rand(Chisq(nObs-1))/(nObs-1))
7  mcQuantile = quantile([myT(n) for _ in 1:N],alpha)
8  analyticQuantile = quantile(TDist(n-1),alpha)
9
10 println("Quantile from Monte Carlo: ", mcQuantile)
11 println("Analytic qunatile: ", analyticQuantile)
12
13 xGrid = -5:0.1:5
14 plot(xGrid, pdf.(Normal(), xGrid), c=:black, label="Normal Distribution")
15 scatter!(xGrid, pdf.(TDist(1) ,xGrid),
16         c=:blue, msw=0, label="DOF = 1")
17 scatter!(xGrid, pdf.(TDist(3), xGrid),
18         c=:red, msw=0, label="DOF = 3")
19 scatter!(xGrid, pdf.(TDist(100),xGrid),
20         c=:green, msw=0, label="DOF = 100",
21         xlims=(-4,4), ylims=(0,0.5), xlabel="X", ylabel="Density")
```

```
Quantile from Monte Carlo: -1.8848554309670498
Analytic qunatile: -1.8856180831641265
```

In line 6 we specify the function `myT()` which generates a t-distributed random variable by using a standard normal and a chi-squared random variable, just as in (5.4). In line 7 we use `N` replications of `myT()` to estimate the `alpha` quantile. Then in line 8 we compute the quantile analytically for a corresponding t-distribution represented by `TDist(n-1)`. The estimated quantile and computed quantile are then printed in lines 10-11. The remainder of the code plots three t-distributions, generating Figure 5.5.

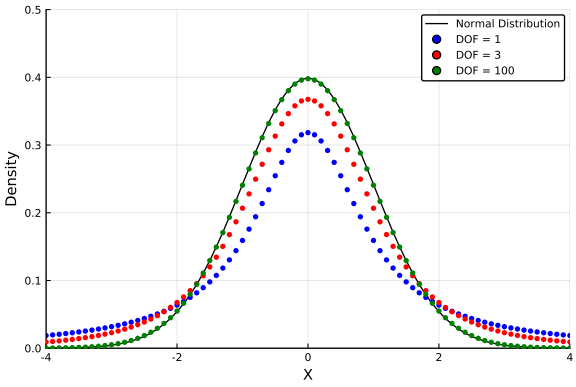

Figure 5.5: As the number of degrees of freedom (DOF) increases, the T-distribution approaches that of the normal distribution.

Two Samples and the F-Distribution

Many statistical procedures involve the ratio of sample variances, or similar quantities, for two or more samples. For example, if $X_1, \ldots, X_{n_1}$ is one sample and $Y_1, \ldots, Y_{n_2}$ is another sample, and both samples are distributed normally with the same parameters, one can look at the ratio of the two sample variances,

$$F_{\text{statistic}} = \frac{S_X^2}{S_Y^2}.$$

It turns out that such a statistic is distributed according to what is called the *F-distribution*, with density given by,

$$f(x) = K(a,b)\,\frac{x^{a/2-1}}{(b+ax)^{(a+b)/2}} \qquad \text{with} \qquad K(a,b) = \frac{\Gamma\Big(\dfrac{a+b}{2}\Big)a^{a/2}b^{b/2}}{\Gamma\Big(\dfrac{a}{2}\Big)\Gamma\Big(\dfrac{b}{2}\Big)}.$$

Here the parameters a and b are the *numerator degrees of freedom* and *denominator degrees of freedom* respectively. In the case of $F_{\text{statistic}}$ we set $a = n_1 - 1$ and $b = n_2 - 1$.

In agreement with (5.2), an alternative view is that the random variable F is obtained by the ratio of two independent chi-squared random variables, normalized by their degrees of freedom. The F-distribution plays a key role in the popular Analysis of Variance (ANOVA) procedures, further explored in Section 7.3.

We now briefly explore the F-distribution in Listing 5.5 by simulating two sample sets of data with n_1 and n_2 observations respectively from a normal distribution. The ratio of the sample variances from the two distributions is then compared to the PDF of an F-distribution with parameters $n_1 - 1$ and $n_2 - 1$. The listing generates Figure 5.6.

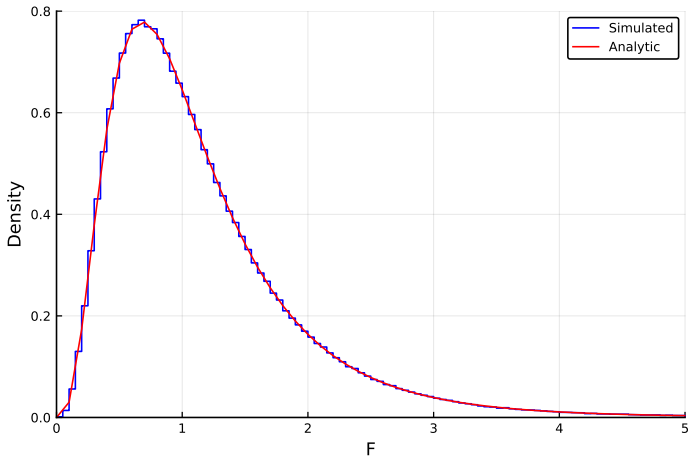

Figure 5.6: Histogram of the ratio of two sample variances against the PDF of an F-distribution.

Listing 5.5: Ratio of variances and the F-distribution

```
1  using Distributions, Plots; pyplot()
2
3  n1, n2 = 10, 15
4  N = 10^6
5  mu, sigma = 10, 4
6  normDist = Normal(mu,sigma)
7
8  fValues = Array{Float64}(undef, N)
9
10 for i in 1:N
11     data1 = rand(normDist,n1)
12     data2 = rand(normDist,n2)
13     fValues[i] = var(data1)/var(data2)
14 end
15
16 fRange = 0:0.1:5
17 stephist(fValues, bins=400, c=:blue, label="Simulated", normed=true)
18 plot!(fRange, pdf.(FDist(n1-1, n2-1), fRange),
19         c=:red, label="Analytic", xlims=(0,5), ylims=(0,0.8),
20         xlabel = "F", ylabel = "Density")
```

In lines 3-4 we define the total number of observations for our two sample groups, `n1` and `n2`, as well as the total number of F-statistics we will generate, `N`. In lines 10-14 we simulate two separate sample groups, `data1` and `data2`, by randomly sampling from the same underlying normal distribution. A single F-statistic is then calculated from the ratio of the sample variances of these two groups. The remainder of the code creates the figure where in line 18 the constructor `FDist()` is used to create an F-distribution with the parameters `n1-1` and `n2-2`.

5.3 The Central Limit Theorem

In the previous section we assumed sampling from a normal population, and this assumption gave rise to a variety of properties of statistics associated with the sampling. However, why would such an assumption hold? A key lies in one of the most fundamental results of probability and statistics, *the Central Limit Theorem* (CLT).

While the CLT has several versions and many generalizations, they all have one thing in common: summations of a large number of random quantities, each with finite variance, yields a sum that is approximately normally distributed. This is the main reason that the normal distribution is ubiquitous in nature and present throughout the universe.

We now develop this more formally. Consider an i.i.d. sequence $X_1, X_2, \ldots$ where all X_i are distributed according to some distribution $F(x_i \,;\, \theta)$ with mean μ and finite variance σ^2. Consider now the random variable,

$$Y_n := \sum_{i=1}^{n} X_i.$$

It is clear that $\mathbb{E}[Y_n] = n\mu$ and $\mathrm{Var}(Y_n) = n\sigma^2$. Hence we may consider a random variable,

$$\tilde{Y}_n := \frac{Y_n - n\mu}{\sqrt{n}\sigma}.$$

Observe that $\tilde{Y}_n$ is zero mean and unit variance. The CLT states that as $n \to \infty$, the distribution of $\tilde{Y}_n$ converges to a standard normal distribution. That is, for every $x \in \mathbb{R}$,

$$\lim_{n\to\infty} \mathbb{P}(\tilde{Y}_n \leq x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-\frac{u^2}{2}}\, du.$$

Alternatively, this may be viewed as indicating that for non-small n,

$$Y_n \underset{\text{approx}}{\sim} N\big(n\mu, n\sigma^2\big),$$

where N is a the normal distribution with mean $n\mu$ and variance $n\sigma^2$.

In addition, by dividing the numerator and denominator of $\tilde{Y}_n$ by n, we see an immediate consequence of the CLT. That is, for non-small n, the sample mean of n observations denoted by $\overline{X}_n$ satisfies,

$$\overline{X}_n \underset{\text{approx}}{\sim} N\left(\mu, \left(\frac{\sigma}{\sqrt{n}}\right)^2\right).$$

Hence the CLT states that sample means from i.i.d. samples with finite variances are asymptotically distributed according to a normal distribution as the sample size grows. This ubiquity of the normal distribution justifies the normality assumption employed when using many of the statistical procedures that we cover in Chapters 6 and 7.

To illustrate the CLT, consider three different distributions below, noting that each has a mean and variance both equal 1:

1. A uniform distribution, on $[1 - \sqrt{3},\ 1 + \sqrt{3}]$.

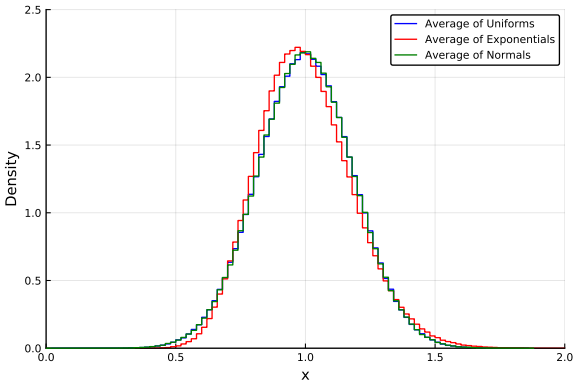

Figure 5.7: Histograms of sample means for different underlying distributions.

2. An exponential distribution with $\lambda = 1$.

3. A normal distribution with both a mean and variance of 1.

In Listing 5.6, we illustrate the central limit theorem, by generating a histogram of `N` sample means for each of the three different distributions mentioned above. Although each of the underlying distributions is very different, i.e. uniform, exponential and normal, the sampling distribution of the sample means all approach that of the normal distribution centered about 1 with standard deviation $1/\sqrt{n}$. Notice that in the case of the exponential distribution, $n = 30$ isn't "enough" to get a "perfect fit" to a normal distribution.

Listing 5.6: The central limit theorem

```
1  using Distributions, Plots; pyplot()
2
3  n, N = 30, 10^6
4
5  dist1 = Uniform(1-sqrt(3),1+sqrt(3))
6  dist2 = Exponential(1)
7  dist3 = Normal(1,1)
8
9  data1 = [mean(rand(dist1,n)) for _ in 1:N]
10 data2 = [mean(rand(dist2,n)) for _ in 1:N]
11 data3 = [mean(rand(dist3,n)) for _ in 1:N]
12
13 stephist([data1 data2 data3], bins=100,
14     c=[:blue :red :green], xlabel = "x", ylabel = "Density",
15     label=["Average of Uniforms" "Average of Exponentials" "Average of Normals"],
16     normed=true, xlims=(0,2), ylims=(0,2.5))
```

In lines 5-7 we define three different distribution type objects: a continuous uniform distribution over the domain $[1-\sqrt{3},\ 1+\sqrt{3}]$, an exponential distribution with a mean of 1, and a normal distribution with mean and standard deviation both 1. In lines 9-11, we generate `N` sample means, each consisting of `n` observations, for each distribution defined above. In lines 13-16 we plot three separate histograms based on the sample mean vectors previously generated. It can be observed that for large `N`, these histograms approach that of a normal distribution, and in addition, the mean of the data approaches the mean of the underlying distribution from which the samples were taken.

5.4 Point Estimation

Given a random sample, $X_1, \ldots, X_n$, a common task of statistical inference is to estimate a parameter θ, or a function of it, say $h(\theta)$. The process of designing an estimator, analyzing its performance, and carrying out the estimation is called *point estimation*.

Although we can never know the underlying parameter θ, or $h(\theta)$ exactly, we can arrive at an estimate for it via an *estimator* $\hat{\theta} = f(X_1, \ldots, X_n)$. Here the design of the estimator is embodied by $f(\cdot)$, a function that specifies how to construct the estimate from the sample.

An important question to ask is how close is $\hat{\theta}$ to the actual unknown quantity θ or $h(\theta)$. In this section we first describe several ways of quantifying and categorizing this "closeness", and then present two common methods for designing estimators; the *method of moments* and *maximum likelihood estimation* (MLE).

The design of (point) estimators is a central part of statistics. However in elementary statistics courses for science students, engineers, or social studies researchers, point estimation is often not explicitly mentioned. The reason for this is that one can estimate the mean and variance via, $\overline{X}$ and S^2 respectively, see (5.1). That is, in the case of $h(\cdot)$ being either the mean or the variance of the distribution, the estimator given by the sample mean or sample variance respectively is a natural candidate and performs exceptionally well. However, in other cases, choosing an estimation procedure is less straight forward.

Consider for example the case of a uniform distribution on the range $[0, \theta]$, and say we are interested in estimating θ based on a random sample, $X_1, \ldots, X_n$. In this case one could construct and estimator in many different ways. For example, here are a few alternative estimators:

$$\begin{aligned}
\hat{\theta}_1 &= f_1(X_1, \ldots, X_n) := \max\{X_i\}, \\
\hat{\theta}_2 &= f_2(X_1, \ldots, X_n) := 2\,\overline{X}, \\
\hat{\theta}_3 &= f_3(X_1, \ldots, X_n) := 2\,\text{median}(X_1, \ldots, X_n), \\
\hat{\theta}_4 &= f_4(X_1, \ldots, X_n) := \sqrt{12 S^2}.
\end{aligned} \tag{5.5}$$

Each of these makes some sense in their own right; $\hat{\theta}_1$ is based on the fact that θ is an upper bound of the observations, $\hat{\theta}_2$ and $\hat{\theta}_3$ utilize the fact that the sample mean and sample median are both expected to fall on $\theta/2$, and finally $\hat{\theta}_4$ utilizes the fact that the variance of the distribution is given by $S^2 = \theta^2/12$. Given that there are various possible estimators, we require a methodology for comparing them and perhaps developing others, with the aim of choosing a suitable one. In the remainder of this section we describe some methods for analyzing the performance of such estimators and others.

Describing the Performance and Behavior of Estimators

When analyzing the performance of an estimator $\hat{\theta}$, it is important to understand that it is a random variable. One common measure of its performance is the *Mean Squared Error* (MSE),

$$MSE_\theta(\hat{\theta}) := \mathbb{E}\big[(\hat{\theta} - \theta)^2\big] = \text{Var}(\hat{\theta}) + (\mathbb{E}[\hat{\theta}] - \theta)^2 := \text{variance} + \text{bias}^2. \tag{5.6}$$

The second equality arises naturally from adding and subtracting $\mathbb{E}[\hat{\theta}]$, expanding and collecting terms. In this representation, we see that the MSE can be decomposed into the variance of the estimator, and it's bias squared. Low variance is clearly a desirable performance measure. The same applies to the *bias*, which is a measure of the expected difference between the estimator and the true parameter value. Note that in machine learning the "Bias variance tradeoff" is often considered as a tradeoff between model complexity and model generalizability. The idea is similar to the decomposition in (5.6), however it is conceptually different because the setting is different. More details are in Chapter 9.

One question that arises with regards to estimation is: are there cases where estimators are *unbiased* - that is, they have a bias of 0, or alternatively $\mathbb{E}[\hat{\theta}] = \theta$? The answer is yes. We show this now using the sample mean as a simple example.

Consider $X_1, \ldots, X_n$ distributed according to any distribution with a finite mean μ. In this case, say we are interested in estimating μ (note that $\mu = h(\theta)$ for some function h). It is easy to see that the sample mean $\overline{X}$ is itself a random variable with mean μ, and is hence unbiased. Furthermore, the variance of this estimator is σ^2/n, where σ^2 is the original variance of X_i. Since the estimator is unbiased, the MSE equals the variance, i.e. σ^2/n. In fact, it can be shown that the sample mean is the estimator of θ with minimal mean square error over all other estimators.

Now consider a case where the population mean μ is known, but the population variance σ^2 is unknown, and that we wish to estimate it. As a sensible estimator consider,

$$\hat{\sigma^2} := \frac{1}{n}\sum_{i=1}^{n}(X_i - \mu)^2. \tag{5.7}$$

Computing the mean of $\hat{\sigma^2}$ yields:

$$\mathbb{E}[\hat{\sigma^2}] = \frac{1}{n}\mathbb{E}\left[\sum_{i=1}^{n}(X_i - \mu)^2\right] = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}\Big[(X_i - \mu)^2\Big] = \frac{1}{n}n\sigma^2 = \sigma^2.$$

Hence $\hat{\sigma^2}$ is an unbiased estimator for σ^2. However, say we are now also interested in estimating the (population) standard deviation, σ. In this case it is natural to use the estimator,

$$\hat{\sigma} := \sqrt{\hat{\sigma^2}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(X_i - \mu)^2}.$$

Interestingly, while this is a perfectly sensible estimator, it is not unbiased. We illustrate this via simulation in Listing 5.7. In it we consider a uniform distribution over $[0, 1]$, where the population mean, variance and standard deviation are 0.5, 1/12 and $\sqrt{1/12}$ respectively. We then estimate the bias of $\hat{\sigma^2}$ and $\hat{\sigma}$ via Monte Carlo simulation. The output shows that $\hat{\sigma}$ is not unbiased. However, as the numerical results illustrate, it is *asymptotically unbiased*. That is, the bias tends to 0 as the sample size n grows.

Listing 5.7: A biased estimator

```
1  using Random, Statistics
2  Random.seed!(0)
3
4  trueVar, trueStd = 1/12, sqrt(1/12)
5
6  function estVar(n)
7      sample = rand(n)
8      sum((sample .- 0.5).^2)/n
9  end
10
11 N = 10^7
12 for n in 5:5:30
13     biasVar = mean([estVar(n) for _ in 1:N]) - trueVar
14     biasStd = mean([sqrt(estVar(n)) for _ in 1:N]) - trueStd
15     println("n = ",n, " Var bias: ", round(biasVar, digits=5),
16                      "\t Std bias: ", round(biasStd, digits=5))
17 end
```

```
n = 5 Var bias: 1.0e-5          Std bias: -0.00642
n = 10 Var bias: 1.0e-5         Std bias: -0.00303
n = 15 Var bias: 0.0            Std bias: -0.00199
n = 20 Var bias: -1.0e-5        Std bias: -0.00148
n = 25 Var bias: -1.0e-5        Std bias: -0.00117
n = 30 Var bias: 0.0            Std bias: -0.00098
```

In lines 6-8 the function `estVar()` is defined, which implements (5.7). In lines 12-17, we loop over sample sizes $n = 5, 10, 15, \ldots, 30$, and for each we repeat `N` sampling experiments, for which we estimate the biases for $\hat{\sigma^2}$ and $\hat{\sigma}$ respectively. The biases are then estimated and the values stored in `biasVar` and `biasStd`.

Having explored an estimator for σ^2 with μ known, as well as briefly touching an estimator for σ for the same case, we now ask the question: What would be a sensible estimator for σ^2 for the more realistic case where μ is not known? A natural first suggestion would be to replace μ in (5.7) with $\overline{X}$ to obtain,

$$\tilde{S}^2 := \frac{1}{n}\sum_{i=1}^{n}(X_i - \overline{X})^2.$$

With a few lines of computations involving expectations, one can verify that,

$$\mathbb{E}[\tilde{S}^2] = \frac{n-1}{n}\sigma^2.$$

Hence it is biased, albeit asymptotically unbiased. This is the reason that the preferred estimator, S^2 is actually,

$$S^2 = \frac{n}{n-1}\tilde{S}^2.$$

as in (5.1). This yields an unbiased estimator.

There are other important qualitative properties of estimators that one may explore. One such property is *consistency*. Roughly, we say that an estimator is consistent if it converges to the true value as the number of observations grows to infinity. More can be found in mathematical

statistics references such as [DS11] and [CB01]. The remainder of this section presents two common methodologies for estimating parameters; method of moments and maximum likelihood estimation. A comparison of these two methodologies is presented.

Method of Moments

The *method of moments* is a methodological way to obtain parameter estimates for a distribution. The key idea is based on moment estimators for the k'th moment, $\mathbb{E}[X_i^k]$,

$$\hat{m}_k = \frac{1}{n}\sum_{i=1}^{n} X_i^k. \tag{5.8}$$

As a simple example, consider a uniform distribution on $[0, \theta]$. An estimator for the first moment ($k = 1$) is then, $\hat{m}_1 = \overline{X}$. Now we denote by X a typical random variable from this sample. For such a distribution $\mathbb{E}[X^1] = \theta/2$. We can then equate the moment estimator with the first moment expression to arrive at the equation,

$$\frac{\theta}{2} = \hat{m}_1.$$

Notice that this equation involves the unknown parameter θ and the moment estimator obtained from the data. Then trivially solving for θ yields the estimator,

$$\hat{\theta} = 2\hat{m}_1,$$

which is exactly $\hat{\theta}_2$ from (5.5).

In cases where there are multiple unknown parameters, say K, we use the first K moment estimates to formulate a system of K equations and K unknowns. This system of equations can be written as,

$$\mathbb{E}[X^k \ ; \ \theta_1, \ldots, \theta_K] = \hat{m}_k \qquad \text{for} \qquad k = 1, \ldots, K. \tag{5.9}$$

For many textbook examples (such as the uniform distribution case described above), we are able to solve this system of equations analytically, yielding a solution,

$$\hat{\theta}_k = g_k(\hat{m}_1, \ldots, \hat{m}_K) \qquad \text{for} \qquad k = 1, \ldots, K. \tag{5.10}$$

Here the functions $g_k(\cdot)$ describe the solution of the system of equations. However, it is often not possible to obtain explicit expressions for $g_k(\cdot)$. In these cases numerical techniques are typically used to solve the corresponding system of equations.

As an example, consider the triangular distribution with density,

$$f(x) = \begin{cases} 2\dfrac{x-a}{(b-a)(c-a)}, & x \in [a, c), \\ 2\dfrac{b-x}{(b-a)(b-c)}, & x \in [c, b]. \end{cases}$$

This distribution has support $[a, b]$, and a maximum at c with $a \leq c \leq b$ and $a < b$. Note that the `Julia` triangular distribution function uses this same parameterization: `TriangularDist(a,b,c)`.

Now straightforward (yet tedious) computation yields the first three moments, $\mathbb{E}[X^1]$, $\mathbb{E}[X^2]$, $\mathbb{E}[X^3]$, as well as the system of equations for the method of moments:

$$\begin{aligned}\hat{m}_1 &= \frac{1}{3}(a+b+c),\\ \hat{m}_2 &= \frac{1}{6}(a^2+b^2+c^2+ab+ac+bc),\\ \hat{m}_3 &= \frac{1}{10}(a^3+b^3+c^3+a^2b+a^2c+b^2a+b^2c+c^2a+c^2b+abc).\end{aligned} \tag{5.11}$$

Generally, this system of equations is not analytically solvable. Hence, the method of moments estimator is given by a numerical solution to (5.11). In Listing 5.8, given a series of observations, we numerically solve this system of equations through the use of the `NLsolve` package, and arrive at estimates for the values of a, b and c. Observe that the equations (5.11) are symmetric in-terms of a, b and c in the sense that permuting these values does not change the equations. Hence, when using a numerical solver, there is a possibility that it will return an arbitrary permutation of the solutions. We remedy this by sorting the solutions and picking estimators for a, b and c according to the sorted order.

Listing 5.8: Point estimation via the method of moments using a numerical solver

```
using Random, Distributions, NLsolve
Random.seed!(0)

a, b, c = 3, 5, 4
dist = TriangularDist(a,b,c)
n = 2000
samples = rand(dist,n)

m_k(k,data) = 1/n*sum(data.^k)
mHats = [m_k(i,samples) for i in 1:3]

function equations(F, x)
    F[1] = 1/3*( x[1] + x[2] + x[3] ) - mHats[1]
    F[2] = 1/6*( x[1]^2 + x[2]^2 + x[3]^2 + x[1]*x[2] + x[1]*x[3] +
                 x[2]*x[3] ) - mHats[2]
    F[3] = 1/10*( x[1]^3 + x[2]^3 + x[3]^3 + x[1]^2*x[2] + x[1]^2*x[3] +
                 x[2]^2*x[1] + x[2]^2*x[3] + x[3]^2*x[1] + x[3]^2*x[2] +
                 x[1]*x[2]*x[3] ) - mHats[3]
end

nlOutput = nlsolve(equations, [ 0.1; 0.1; 0.1])
sol = sort(nlOutput.zero)
aHat, bHat, cHat = sol[1], sol[3], sol[2]
println("Found estimates for (a,b,c) = ", (aHat, bHat, cHat) , "\n" )
println(nlOutput)
```

```
Found estimates for (a,b,c) = (3.002706152232, 5.003033254712, 3.999191608726)

Results of Nonlinear Solver Algorithm
 * Algorithm: Trust-region with dogleg and autoscaling
 * Starting Point: [0.1, 0.1, 0.1]
 * Zero: [5.00303, 3.99919, 3.00271]
 * Inf-norm of residuals: 0.000000
 * Iterations: 14
 * Convergence: true
```

```
  * |x - x'| < 0.0e+00: false
  * |f(x)| < 1.0e-08: true
* Function Calls (f): 15
* Jacobian Calls (df/dx): 13
```

In line 1 the we specify using the `NLsolve` package. This package contains numerical methods for solving non-linear systems of equations. In lines 4-7, we specify the parameters of the triangular distribution and the distribution itself `dist`. We also specify the total number of samples `n`, and generate our sample set of observations `samples`. In line 9, the function `m_k()` is defined, which implements (5.8), and in line 10, this function is used to estimate the first three moments, given our observations `samples`. In line 12-19, we set up the system of simultaneous equations within the function `equations()`. This specific format is used as it is a requirement of the `nlsovle()` function which is used later. The `equations()` function takes two arrays as input, `F` and `x`. The elements of `F` represent the left hand side of the series of equations (which are later solved for zero), and the elements of `x` represent the corresponding constants of the equations. Note that in setting up the equations from (5.11), the moment estimators are moved to the right hand side, so that the zeros can be found. In line 21, the `nlsolve()` function from the `NLsolve` package is used to solve the zeros of the function `equations()`, given starting coefficient estimates of `[0.1; 0.1; 0.1]`. In this example, since the Jacobian was not specified, it is computed by finite differences. In lines 22-23 we sort the solution and set the estimates of the parameters based on the sorted order. In line 24, the zeros of our function are printed as output through the use of `.zero`, which is used to return just the zero field of the `nlsolve()` output. In line 25, the complete output from the function `nlsolve()` is printed as output.

Maximum Likelihood Estimation (MLE)

Maximum likelihood estimation is another commonly used technique for creating point estimators. In fact, in the study of mathematical statistics, it is probably the most popular method used. The key principle is to consider the *likelihood* of the parameter θ having a specific value given observations $x_1, \ldots, x_n$. That is, what is the most likely parameter value based on the observations. This is done via the likelihood function, which is presented below for the i.i.d. case of continuous probability distributions,

$$L(\theta \ ; \ x_1, \ldots, x_n) = f_{X_1,\ldots,X_n}(x_1, \ldots, x_n \ ; \ \theta) = \prod_{i=1}^{n} f(x_i \ ; \ \theta). \tag{5.12}$$

In the second equality, the joint probability density of $X_1, \ldots, X_n$ is represented as the product of the individual probability densities, since the observations are assumed i.i.d.

A key observation is that the likelihood, $L(\cdot)$, in (5.12) is a function of the parameter θ, influenced by the sample, $x_1, \ldots, x_n$. Now given the likelihood, the *maximum likelihood estimator* is a value θ that maximizes $L(\theta \ ; \ x_1, \ldots, x_n)$. The rational behind using this as an estimator is that it chooses the parameter value θ that is most plausible, given the observed sample.

As an example, consider the continuous uniform distribution on $[0, \theta]$. In this case, it is useful to consider the PDF for an individual observation as,

$$f(x \ ; \ \theta) = \frac{1}{\theta}\mathbf{1}\{x \in [0, \theta]\} \qquad \text{for} \qquad x \in \mathbb{R}.$$

Here the *indicator function* $\mathbf{1}\{\cdot\}$ explicitly constrains the support of the random variable to $[0, \theta]$. Now using (5.12), it follows that,

$$L(\theta \ ; \ x_1, \ldots, x_n) = \frac{1}{\theta^n} \prod_{i=1}^{n} \mathbf{1}\{x_i \in [0, \theta]\} = \frac{1}{\theta^n} \mathbf{1}\{0 \leq \min_i x_i\} \mathbf{1}\{\max_i x_i \leq \theta\}.$$

From this we see that for any sample $x_1, \ldots, x_n$ with non-negative values, this function (of θ) is maximized at $\hat{\theta} = \max_i x_i$. Hence as you can see the MLE for this case is exactly $\hat{\theta}_1$ from (5.5).

Many textbooks present constructed examples of MLEs, where the likelihood is a differentiable function of θ. In such cases, these MLEs can be solved explicitly, by carrying out the optimization of the likelihood function analytically (for example, see [CB01] and [DS11]). However, this is not always possible, and often numerical optimization of the likelihood function is carried out instead.

As an example, consider the case where we have n random samples from what we know to be a gamma distribution, with PDF,

$$f(x) = \frac{\lambda^\alpha}{\Gamma(\alpha)} x^{\alpha - 1} e^{-\lambda x}.$$

and parameters, $\lambda > 0$ and $\alpha > 0$. In such a case where λ and α are both unknown, there is not an explicit solution to the MLE optimization problem, and hence we resort to numerical methods instead. In Listing 5.9, we use MLE to construct a plot of the likelihood function. That is, given synthetic data, we calculate the likelihood function for various combinations of α and λ. Note that directly after this example, we present an elegant approach for this numerical problem.

Listing 5.9: The likelihood function for a gamma distributions parameters

```
using Random, Distributions, Plots, LaTeXStrings; pyplot()
Random.seed!(0)

actualAlpha, actualLambda = 2,3
gammaDist = Gamma(actualAlpha,1/actualLambda)
n = 10^2
sample = rand(gammaDist, n)

alphaGrid = 1:0.02:3
lambdaGrid = 2:0.02:5

likelihood = [prod([pdf.(Gamma(a,1/l),v) for v in sample])
                        for l in lambdaGrid, a in alphaGrid]

surface(alphaGrid, lambdaGrid, likelihood, lw=0.1,
        c=cgrad([:blue, :red]), legend=:none, camera = (135,20),
        xlabel=L"\alpha", ylabel=L"\lambda", zlabel="Likelihood")
```

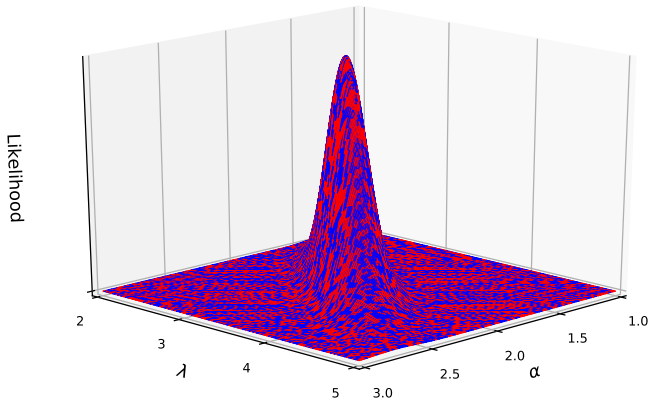

Figure 5.8: Likelihood function on different combinations of α and λ for a gamma distribution.

In lines 4-5 we specify the parameters α and λ, as well as the underlying distribution, `gammaDist`. Note that the gamma distribution in Julia, `Gamma()`, uses a different parameterization to what is outlined in Chapter 3 (i.e. `Gamma()` uses α, and $1/\lambda$). In lines 6-7, we generate `n` sample observations, `sample`. In lines 9-10 we specify the grid of values over which we will calculate the likelihood function, based on various combinations of α and λ. In lines 12-13 we first evaluate the likelihood function, (5.12), through the use of the `prod()` function on an array of all PDF values, evaluated for each sample observation, `v`. Through the use of a two-way comprehension, this process is repeated for all possible combinations of `a` and `l` in `alphaGrid` and `lambdaGrid` respectively. This results in a 2-dimensional array of evaluated likelihood functions for various combinations of α and λ, denoted `likelihood`. Lines 15-17 create the plot.

The likelihood function plotted in Figure 5.8 embodies the data. An MLE is then the maximizer of the likelihood. We now investigate this optimization problem further, and in the process present further insight. First observe that any maximizer, $\hat{\theta}$, of $L(\theta \ ; \ x_1, \ldots, x_n)$ will also maximize its logarithm. Practically, both from an analytic and numerical persepective, considering this *log-likelihood function* is often more attractive:

$$\ell(\theta \ ; \ x_1, \ldots, x_n) := \log L(\theta \ ; \ x_1, \ldots, x_n) = \sum_{i=1}^{n} \log\big(f(x_i \ ; \ \theta)\big).$$

Hence, given a sample from a gamma distribution as before, the log-likelihood function is,

$$\ell(\theta \ ; \ x_1, \ldots, x_n) = n\alpha \log(\lambda) - n \log(\Gamma(\alpha)) + (\alpha - 1) \sum_{i=1}^{n} \log(x_i) - \lambda \sum_{i=1}^{n} x_i.$$

We may then divide by n (without compromising the optimizer) to obtain the following function that needs to be maximized:

$$\tilde{\ell}(\theta \ ; \overline{x}, \ \overline{x}_\ell) = \alpha \log(\lambda) - \log(\Gamma(\alpha)) + (\alpha - 1)\overline{x}_\ell - \lambda \overline{x},$$

where, $\overline{x}$ is the sample mean and,

$$\overline{x}_\ell := \frac{1}{n} \sum_{i=1}^{n} \log(x_i).$$

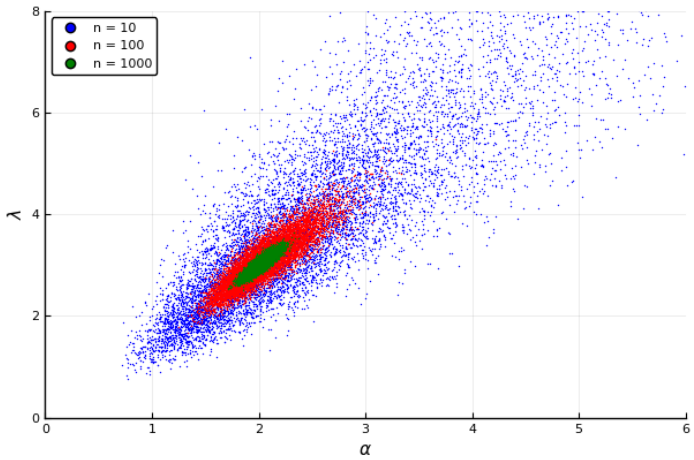

Figure 5.9: Repetitions of MLE for a gamma$(2, 3)$ distribution with $n = 10, 100, 1000$. For $n = 100$ and $n = 1000$, asymptotic normality is visible.

Further simplification is possible by removing the stand-alone $-\overline{x}_\ell$ term, as it does not affect the optimal value. Hence our optimization problem is then,

$$\max_{\lambda>0,\ \alpha>0} \quad \alpha\big(\log(\lambda) + \overline{x}_\ell\big) - \log(\Gamma(\alpha)) - \lambda\overline{x}. \tag{5.13}$$

As is typical in such cases, the function actually depends on the sample only through the two *sufficient statistics* $\overline{x}$ and $\overline{x}_\ell$. Now in optimizing (5.13), we aren't able to obtain an explicit expression for the maximizer. However, taking α as fixed, we may consider the derivative with respect to λ, and equate this to 0:

$$\frac{\alpha}{\lambda} - \overline{x} = 0.$$

Hence, for any optimal α^*, we have that $\lambda^* = \alpha^*/\overline{x}$. This allows us to substitute λ^* for λ in (5.13) to obtain:

$$\max_{\alpha>0} \quad \alpha(\log(\alpha) - \log(\overline{x}) + \overline{x}_\ell) - \log(\Gamma(\alpha)) - \alpha. \tag{5.14}$$

Now by taking the derivative of (5.14) with respect to α, and equating this to 0, we obtain,

$$\log(\alpha) + 1 - \log(\overline{x}) + \overline{x}_\ell - \psi(\alpha) - 1 = 0,$$

where $\psi(z) := \frac{d}{dz}\log(\Gamma(z))$ is the well known *digamma function*. Hence we find that α^* must satisfy:

$$\log(\alpha) - \psi(\alpha) - \log(\overline{x}) + \overline{x}_\ell = 0. \tag{5.15}$$

In addition, since $\lambda^* = \alpha^*/\overline{x}$, our optimal MLE solution is given by (α^*, λ^*). In order to find this value, (5.15) must be solved numerically.

In Listing 5.10 we do just this. In fact, we repeat the act of numerically solving (5.15) many times, and in the process illustrate the distribution of the MLE in terms of λ and α. Note that there are many more properties of the MLE that we do not discuss here, including the asymptotic distribution of the MLE, which happens to be a multivariate normal. However, through this example, we provide an intuitive illustration of the distribution of the MLE, which is bivariate in this case, and can be observed in Figure 5.9.

Listing 5.10: MLE for the gamma distribution

```
1  using SpecialFunctions, Distributions, Roots, Plots, LaTeXStrings; pyplot()
2
3  eq(alpha, xb, xbl) = log(alpha) - digamma(alpha) - log(xb) + xbl
4
5  actualAlpha, actualLambda = 2, 3
6  gammaDist = Gamma(actualAlpha,1/actualLambda)
7
8  function mle(sample)
9      alpha  = find_zero( (a)->eq(a,mean(sample),mean(log.(sample))), 1)
10     lambda = alpha/mean(sample)
11     return [alpha,lambda]
12 end
13
14 N = 10^4
15
16 mles10   = [mle(rand(gammaDist,10)) for _ in 1:N]
17 mles100  = [mle(rand(gammaDist,100)) for _ in 1:N]
18 mles1000 = [mle(rand(gammaDist,1000)) for _ in 1:N]
19
20 scatter(first.(mles10), last.(mles10),
21         c=:blue, ms=1, msw=0, label="n = 10")
22 scatter!(first.(mles100), last.(mles100),
23         c=:red, ms=1, msw=0, label="n = 100")
24 scatter!(first.(mles1000), last.(mles1000),
25         c=:green, ms=1, msw=0, label="n = 1000",
26         xlims=(0,6), ylims=(0,8), xlabel=L"\alpha", ylabel=L"\lambda")
```

In line 1, we specify usage of the `SpecialFunctions` and `Roots` packages, as they contain the `digamma()` and `find_zero()` functions respectively. In line 3, the `eq()` function implements equation (5.15). Note it takes three arguments, an alpha value `alpha`, a sample mean `xb`, and the mean of the log of each observation `xbl`, which is calculated element wise via `log.()`. This allows us to apply `eq()` on vectors. In lines 5-6 we specify the actual parameters of the underlying gamma distribution, as well as the distribution itself. In lines 8-12 the function `mle()` is defined, which in line 9 takes an array of sample observations, and solves the value of `alpha` which satisfies the zero of `eq()`. This is done through the use of the `find_zero()` function, and the anonymous function `(a)->eq(a,mean(sample),mean(log.(sample)))`. Note the trailing `1` in line 9, which is used as the initial value of the iterative solver. In line 10, the corresponding `lambda` value is calculated, and both alpha and lambda are returned as an array of values. In line 16, 10 random samples are made from our gamma distribution, and then the function `mle()` is used to solve for the corresponding values of alpha and lambda and an array of arrays. This experiment is repeated through a comprehension `N` times total, and the resulting array of arrays stored as `mles10`. Lines 17-18 repeat the same procedure as that in line 16, however in these two cases, the experiments are conducted for 100 and 1000 random samples respectively. In lines 20-22, a scatterplot of the resulting pairs of $\hat{\alpha}$ and $\hat{\lambda}$ are plotted, for the cases of the sample size being equal to 10, 100 and 1000. Note the use of the `first()` and `last()` functions, which are used to return the values of alpha and lambda respectively. Note that the bivariate distribution of alpha and lambda can be observed. In addition, for a larger number of observations, it can be seen that the data is centered about the true underlying parameters alpha and lambda values of 2 and 3. This agrees with the fact that the MLE is asymptotically unbiased.

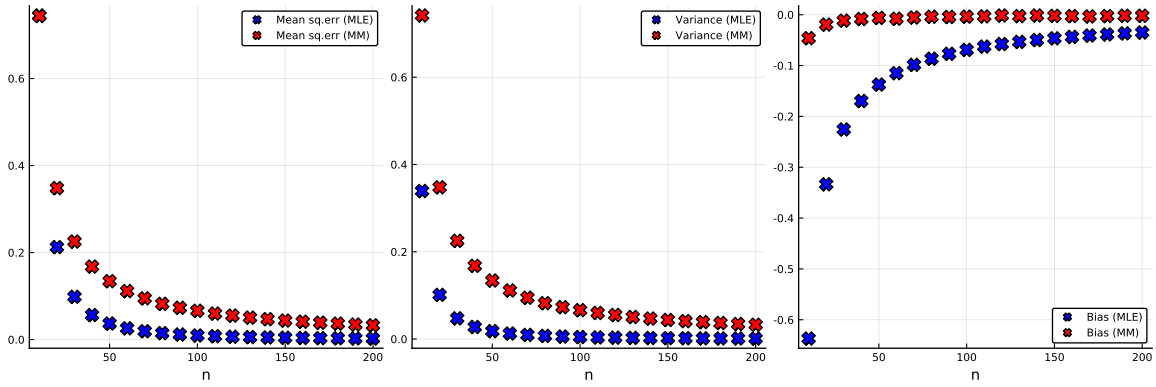

Figure 5.10: Comparing the method of moments and MLE in terms of MSE, variance, and bias.

Comparing the Method of Moments and MLE

We now carry out an illustrative comparison between a method of moments estimator and an MLE estimator on a specific example. Consider a random sample $x_1, \ldots, x_n$ from a uniform distribution on the interval (a, b). The MLE for the parameter $\theta = (a, b)$, can be shown to be,

$$\hat{a} = \min\{x_1, \ldots, x_n\}, \qquad \hat{b} = \max\{x_1, \ldots, x_n\}. \tag{5.16}$$

For the method of moments estimator, since $X \sim \text{uniform}(a, b)$, it follows that,

$$\mathbb{E}[X] = \frac{a+b}{2}, \qquad \text{Var}(X) = \frac{(b-a)^2}{12}.$$

Hence, by solving for a and b, and replacing $\mathbb{E}[X]$ and $\text{Var}(X)$ with $\overline{x}$ and s^2 respectively, we obtain,

$$\hat{a} = \overline{x} - \sqrt{3}\, s, \qquad \hat{b} = \overline{x} + \sqrt{3}\, s. \tag{5.17}$$

Observe that here we are actually using the second central moment (variance) as opposed to the second moment to construct the estimator. This is a slight variation on the method of moments method described above and yields a nicer expression.

Now we can compare how the estimators (5.16) and (5.17) perform based on MSE, specifically the variance and bias. In Listing 5.11 we use Monte Carlo simulation to compare the estimates of $\hat{b}$ using both the method of moments and MLE, for different cases of n. The code creates Figure 5.10 analyzing MSE, bias and variance. As can be seen the MSE of maximum likelihood is lower than the MSE of the method of moments and this is due to the variance of maximum likelihood being lower. However, maximum likelihood exhibits more significant bias than the method of moments. Nevertheless, observe that after squaring, the bias contribution to the MSE is not as significant as the variance. The reader should keep in mind that these conclusions about MSE/variance/bias are specific to this example. However, there is more supporting theory for the usefullnes of maximum likelihood estimation as $n \to \infty$. See for example [CB01].

Listing 5.11: MSE, bias and variance of estimators

```
1  using Distributions, Plots; pyplot()
2
3  N = 10^5
4  nMin, nStep, nMax = 10, 10, 200
5  nn = Int(nMax/nStep)
6  sampleSizes = nMin:nStep:nMax
7  trueB = 5
8  trueDist = Uniform(-2, trueB)
9
10 MLEest(data) = maximum(data)
11 MMest(data)  = mean(data) + sqrt(3)*std(data)
12
13 res = Dict{Symbol,Array{Float64}}(
14     ((sym) -> sym => Array{Float64}(undef,nn)).(
15         [:MSeMLE,:MSeMM, :VarMLE,:VarMM,:BiasMLE,:BiasMM]))
16
17 for (i, n) in enumerate(sampleSizes)
18     mleEst, mmEst = Array{Float64}(undef, N), Array{Float64}(undef, N)
19     for j in 1:N
20         sample    = rand(trueDist,n)
21         mleEst[j] = MLEest(sample)
22         mmEst[j]  = MMest(sample)
23     end
24     meanMLE, meanMM = mean(mleEst), mean(mmEst)
25     varMLE, varMM = var(mleEst), var(mmEst)
26
27     res[:MSeMLE][i] = varMLE + (meanMLE - trueB)^2
28     res[:MSeMM][i] = varMM + (meanMM - trueB)^2
29     res[:VarMLE][i] = varMLE
30     res[:VarMM][i] = varMM
31     res[:BiasMLE][i] = meanMLE - trueB
32     res[:BiasMM][i] = meanMM - trueB
33 end
34
35 p1 = scatter(sampleSizes, [res[:MSeMLE] res[:MSeMM]], c=[:blue :red],
36     label=["Mean sq.err (MLE)" "Mean sq.err (MM)"])
37 p2 = scatter(sampleSizes, [res[:VarMLE] res[:VarMM]], c=[:blue :red],
38     label=["Variance (MLE)" "Variance (MM)"])
39 p3 = scatter(sampleSizes, [res[:BiasMLE] res[:BiasMM]], c=[:blue :red],
40     label=["Bias (MLE)" "Bias (MM)"])
41
42 plot(p1, p2, p3, ms=10, shape=:xcross, xlabel="n",
43     layout=(1,3), size=(1200, 400))
```

In line 4 the minimum, maximum and step size for sample size observations are specified. These are used to define the number of sample size groups `nn`. In lines 7 and 8 the true parameter, `trueB` is specified. Lines 10 and 11 specify the two estimators in the functions `MLEest()` and `MMest()`. Line 13-15 create a dictionary mapping symbols (type `Symbol`) to arrays (type `Array{Float64}`). The dictionary is initialized with symbol keys `:MSeMLE,...,:BiasMM`, and with values that are empty arrays. The main simulation loop is in lines 17-33 where we use `enumerate` to loop over tuples `(i,n)`, with `i` the index of the iteration and `n` a value from the range `sampleSizes`. In each iteration we initialize empty arrays for parameter estimates in line 18. We then repeat the experiment `N` times in the loop of lines 19-23. Lines 24-32 record performance measures in the dictionary `res`.

5.5 Confidence Interval as a Concept

Now that we have dealt with the concept of a point estimator, we consider how confident we are about our estimate. The previous section included analysis of such confidence in terms of the mean squared error and its variance and bias components. However, given a single sample, $X_1, \ldots, X_n$, how does one obtain an indication about the accuracy of the estimate? Here the concept of a *confidence interval* comes as an aid.

Consider the case where we are trying to estimate the parameter θ. A confidence interval is then an interval $[L, U]$ obtained from our sample data, such that,

$$\mathbb{P}(L \leq \theta \leq U) = 1 - \alpha, \tag{5.18}$$

where $1 - \alpha$ is called the *confidence level*. Knowing this range $[L, U]$ in addition to θ is useful, as it indicates some level of certainty in regards to the unknown value. Much of elementary classical statistics involves explicit formulas for L and U, based on the sample $X_1, \ldots, X_n$. Most of Chapter 6 is dedicated to this, however in this section we simply introduce the concept through an elementary non-standard example.

Consider a case of a single observation ($n = 1$) taken from a symmetric triangular distribution, with a spread of 2 and an unknown center (mean) μ. In this case, we would set,

$$L = X + q_{\alpha/2}, \qquad U = X + q_{1-\alpha/2},$$

where q_u is the u'th quantile of a triangular distribution centered at 0, and having a spread of 2. Setting L and U in this manner ensures that (5.18) holds. Note that this is not the only possible construction of a confidence interval, however it makes sense due to the symmetry of the problem. For such a triangular distribution, calcuating quantiles using integration or areas of triangles, it holds that $q_{\alpha/2} = -1 + \sqrt{\alpha}$ and $q_{1-\alpha/2} = 1 - \sqrt{\alpha}$.

Now, given observations, (a single observation in this case), we can compute L and U. A demonstration of this is performed in Listing 5.12 below.

Listing 5.12: A confidence interval for a symmetric triangular distribution

```
using Random, Distributions
Random.seed!(0)

alpha = 0.05
L(obs) = obs - (1-sqrt(alpha))
U(obs) = obs + (1-sqrt(alpha))

mu = 5.57
observation = rand(TriangularDist(mu-1,mu+1,mu))
println("Lower bound L: ", L(observation))
println("Upper bound U: ", U(observation))
```

```
Lower bound L: 5.1997170907797585
Upper bound U: 6.7525034952798
```

In lines 5-6, the functions `L()` and `U()` implement the formulas above. In this simple example, the actual (unknown) parameter value μ is set in line 8. Then the sample, a single observation in this case, is obtained in line 9. The virtue of the example is in presenting the 95% confidence interval, as output by lines 10 and 11. Based on the output (after rounding), we know that with probability 0.95, the unknown parameter lies in the range $[5.2, 6.75]$.

Let us now further explore the meaning of a confidence interval by considering (5.18). The key point is that there is a $1-\alpha$ chance that the actual parameter θ lies in the interval $[L, U]$. This means that if the sampling experiment is repeated say N times, then on average, $N \times (1-\alpha)\%$ of the time the actual parameter θ is covered by the interval.

In Listing 5.13 we present an example where we repeat the previous sampling process $N = 100$ times. Each time we take a single sample (a single observation in this case) and construct the corresponding confidence interval. We observe that about $\alpha \times 100$ times the confidence interval, $[L, U]$, does not include the parameter in question, μ. The results are presented in Figure 5.11.

Listing 5.13: Repetitions of a confidence interval

```
1  using Random, Distributions, StatsPlots; pyplot()
2  Random.seed!(2)
3
4  alpha  = 0.05
5  L(obs) = obs - (1-sqrt(alpha))
6  U(obs) = obs + (1-sqrt(alpha))
7
8  mu = 5.57
9  triDist = TriangularDist(mu-1,mu+1,mu)
10
11 N = 100
12 hitBounds, missBounds = zeros(N, 2), zeros(N,2)
13 for i in 1:N
14     observation = rand(triDist)
15     LL, UU = L(observation), U(observation)
16     if LL <= mu && mu <= UU
17         hitBounds[i,:] = [LL  UU-LL]
18     else
19         missBounds[i,:] = [LL  UU-LL]
20     end
21 end
22
23 groupedbar(hitBounds, bar_position=:stack,
24     c=:blue, la=0, fa=[0 1], label="", ylims=(3,8))
25 groupedbar!(missBounds, bar_position=:stack,
26     c=:red, la=0, fa=[0 1], label="", ylims=(3,8))
27 plot!([0,N+1],[mu,mu],
28     c=:black, xlims=(0,N+1),
29     ylims=(3,8), label="Parameter value", ylabel="Value Estimate")
```

At the heart of this example we repeat the experiment `N = 100` times and create the matrices `hitBounds` and `missBounds`. These are plotted via the `groupedbar()` function from package `StatsPlots`. The main loop in lines 13-21 records a confidence interval as a "hit" in line 17 or alternatively as a "miss" in line 19.

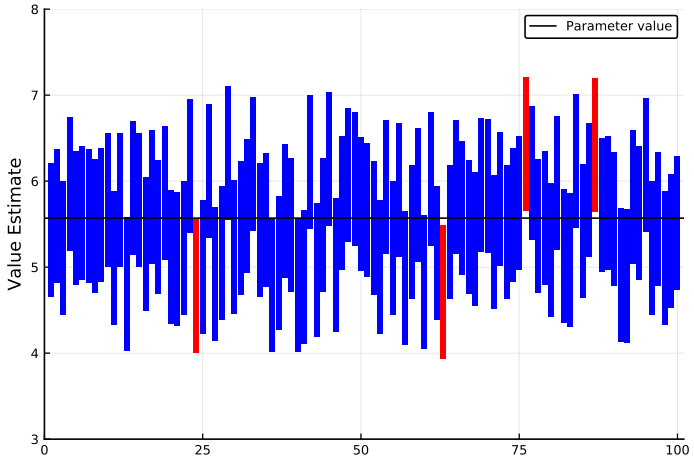

Figure 5.11: 100 confidence intervals. The blue confidence interval bars contain the unknown parameter, while the red ones do not.

5.6 Hypothesis Tests Concepts

Having explored point estimation and confidence intervals, we now consider ideas associated with *hypothesis testing*. The approach involves partitioning the parameter space Θ into Θ_0 and Θ_1, and then, based on the sample, concluding whether one of two hypotheses, H_0 or H_1, holds. Here,

$$H_0 : \theta \in \Theta_0, \qquad H_1 : \theta \in \Theta_1. \tag{5.19}$$

The hypothesis H_0 is called the *null hypothesis* and H_1 the *alternative hypothesis*. The former is the default hypothesis, and in carrying out hypothesis testing our general aim (or hope) is to reject this hypothesis. This is because in typical situations we are wishing to demonstrate that the alternative hypothesis holds, as opposed to some well established status quo captured by the null hypothesis.

		Decision	
		Do not reject H_0	Reject H_0
Reality	H_0 is true	Correct (1-α) "true negative"	Type I error (α) "false positive"
	H_0 is false	Type II error (β) "false negative'	Correct ($1-\beta$) "true positive"

Table 5.1: Type I and Type II errors with their probabilities α and β respectively.

Since our decision is based on a random sample, there is always a chance of making a mistakenly false conclusion. As summarized in Table 5.1, the two types of errors that can be made are a *type I error*: Rejecting H_0 falsely, sometimes called a "false positive", or a *type II error*: Failing to correctly reject H_0, sometimes called a "false negative". The probability α quantifies the likelihood of making a type I error, while the probability of making a type II error is denoted by β. Note that $1-\beta$ is known as the *power* of the hypothesis test, and this concept of power is covered in more detail in

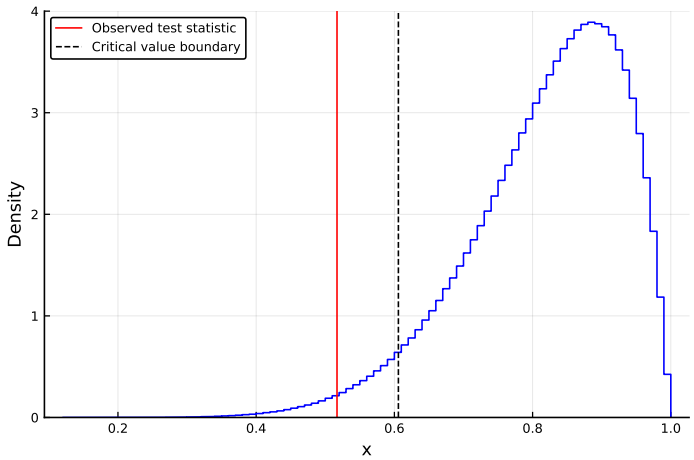

Figure 5.12: The distribution of the test statistic, X^*, under H_0. With $\alpha = 0.05$ the rejection region is to the left of the black dashed line. In a specific sample, the test statistic is on the red line and we reject H_0.

Section 7.5. Note that in carrying out an hypothesis test, α is typically specified, while power is not directly controlled, but rather is influenced by the sample size and other factors.

An important point in terminology is that we don't use the phrase "accept" for the null hypothesis, rather we "fail to reject it" (if we stick with H_0) or "reject it" (if we choose H_1). This is because when we fail to reject H_0, we typically don't know the actual value of β, hence we aren't able to put a level of certainty on H_0 being the case. However if we do reject H_0, then by the design of hypothesis tests we can say that our error probability is bounded by α.

We now present some elementary examples which illustrate the basic concepts involved. Standard hypothesis tests are discussed in depth in Chapter 7.

The Test Statistic, Rejection Region and p-Values

In general, the key objects in hypothesis testing are the *test statistic*, the *rejection region* and *p-values*. Once the scientific question is formulated as an hypothesis by partitioning the parameter space according to (5.19), the next step is to calculate the test statistic. For this we define the test statistic, denoted X^*, as a function of the data. An example can be the sample mean, the sample variance, or other statistics. Importantly, with probabilistic assumptions on the sample data, the test statistic is a random variable itself.

Since the test statistic follows some distribution under H_0, the next step is to consider how likely it is to observe the specific value calculated from our sample data. To this end, in setting up the hypothesis test we typically choose a significance level α, at 0.05, 0.01, or a similar value. It quantifies our level of tolerance for enduring a type I error. For example setting $\alpha = 0.01$ implies we wish to design a test where the probability of type I error is at most 0.01 if H_0 holds. Clearly a low α is desirable, however there are tradeoffs involved since seeking a very low α will imply a high β (low power).

With the test statistic and α at hand, we are able to determine the rejection region which we denote by $\mathcal{R}$. It is a subset of the real line where $\mathbb{P}(X^* \in \mathcal{R}) \leq \alpha$, under H_0. The idea is then to calculate the test statistic X^* and reject H_0 if $X^* \in \mathcal{R}$, and otherwise not to reject H_0. Typically $\mathcal{R}$ is selected at one or both extremes of the support, depending on the distribution of the test statistic and the hypothesis (5.19).

To illustrate these concepts, we now present a simple yet non-standard example. Consider that we have a series of sample observations distributed as continuous uniform between 0 and some unknown upper bound, m. Say that we set,

$$H_0: \quad m = 1, \qquad H_1: \quad m < 1.$$

With observations $X_1, \ldots, X_n$, one possible test statistic is the sample range:

$$X^* = \max(X_1, \ldots, X_n) - \min(X_1, \ldots, X_n).$$

As is always the case, the test statistic is a random variable. Under H_0 we expect the distribution of X^* to have support $[0, 1]$ with the most likely value being close to 1. This is because low values of X^* are less plausible under H_0, since we can expect the minimum to be near 0 and the maximum to be near 1. The explicit form of the distribution of X^* can be analytically obtained however for simplicity we use a Monte Carlo simulation to estimate it and and present the density in Figure 5.12 for $n = 10$ observations.

For this case, it is sensible to reject H_0 if X^* is small. Hence, denoting quantiles of this distribution by $q_0(u)$ we set the rejection region as $\mathcal{R} = [0, q_0(\alpha)]$. Using Monte Carlo, we also compute the rejection region and present it in the figure where the *critical value* is the upper boundary, $q_0(\alpha)$, of the rejection region. Note that computing the rejection region does not require any sample data as it is based on model assumptions and not the sample. Still, it is computed via Monte Carlo in this specific example. The *decision rule* for this hypothesis test is simple: Compare the observed value of the test statistic, x^*, to the critical value $q_0(\alpha)$ and reject H_0 if $x^* \leq q_0(\alpha)$, otherwise do not reject.

An alternative view of hypothesis tests is to consider the p-value. Here we collect the data and compute the observed value of the test statistic x^*. The p-value is then the maximal α under which the test would be rejected with the observed test statistic. In other words we find p which solves $x^* = q_0(p)$. This is computed via $F_0(x^*)$ where $F_0(\cdot)$ is the CDF of X^*.

Using the p-value approach, reporting a low p-value (e.g. $p = 0.0024$) implies that we are very confident in rejecting H_0, while a high p-value (e.g. $= 0.24$) implies we are not. The p-value approach can be used to decide whether H_0 should be rejected or not with a specified α. For this, simply compare p and α, and reject H_0 if $p \leq \alpha$.

Listing 5.14 creates Figure 5.12 and illustrates the operation of the hypothesis test. In this case, we illustrate a scenario where the unknown parameter m is `muActual = 0.75`. With the specific seed selected, it turns out that $x^* = 0.517$. This corresponds to a p-value of 0.0141, which is rejected for $\alpha = 0.05$, however would not be rejected if $\alpha = 0.01$. Keep in mind that the `N` repetitions in this example are simply to obtain the distribution of X^* under H_0 and the critical value, `0.6058`. In the many standard hypothesis tests presented in Chapter 7, the distribution of the test statistic is analytically available, so such Monte Carlo based computation is not needed.

Notice that with specific sample (depends on the seed), lines 18-19 of the code are not executed. However, if you were to change the seed in line 2, this would simulate a scenario with different data points, and it is possible to not reject H_0 even though H_1 holds (`muActual` < 1).

Listing 5.14: The distribution of a test statistic under H_0

```
1  using Distributions, Random, Statistics, Plots; pyplot()
2  Random.seed!(2)
3
4  n, N, alpha = 10, 10^7, 0.05
5  mActual = 0.75
6  dist0, dist1 = Uniform(0,1), Uniform(0,mActual)
7
8  ts(sample) = maximum(sample) - minimum(sample)
9
10 empiricalDistUnderH0 = [ts(rand(dist0,n)) for _ in 1:N]
11 rejectionValue = quantile(empiricalDistUnderH0,alpha)
12
13 sample = rand(dist1,n)
14 testStat = ts(sample)
15 pValue = sum(empiricalDistUnderH0 .<= testStat)/N
16
17 if testStat > rejectionValue
18     print("Didn't reject: ", round(testStat,digits=4))
19     print(" > ", round(rejectionValue,digits=4))
20 else
21     print("Reject: ", round(testStat,digits=4))
22     print(" <= ", round(rejectionValue,digits=4))
23 end
24 println("\np-value = $(round(pValue,digits=4))")
25
26 stephist(empiricalDistUnderH0, bins=100, c=:blue, normed=true, label="")
27 plot!([testStat, testStat], [0,4], c=:red, label="Observed test statistic")
28 plot!([rejectionValue, rejectionValue], [0,4], c=:black, ls=:dash,
29         label="Critical value boundary", legend=:topleft, ylims=(0,4),
30         xlabel = "x", ylabel = "Density")
```

```
Reject: 0.517 <= 0.6058
p-value = 0.0141
```

In line 8 we define the function `ts()` which calculates the test statistic from a sample. We use it in lines 10-11 to obtain `N` (many) samples under H_0 and calculate the `rejectionValue`, $q_0(\alpha)$. The actual testing procedure begins in line 13 when we collect our sample, simulating a point in H_1 (since `mActual = 0.75`). The test statistic is calculated in line 14 and the p-value in line 15. The decision rule is then executed in lines 17-23 and the p-value is also presented. The remainder of the code creates Figure 5.12.

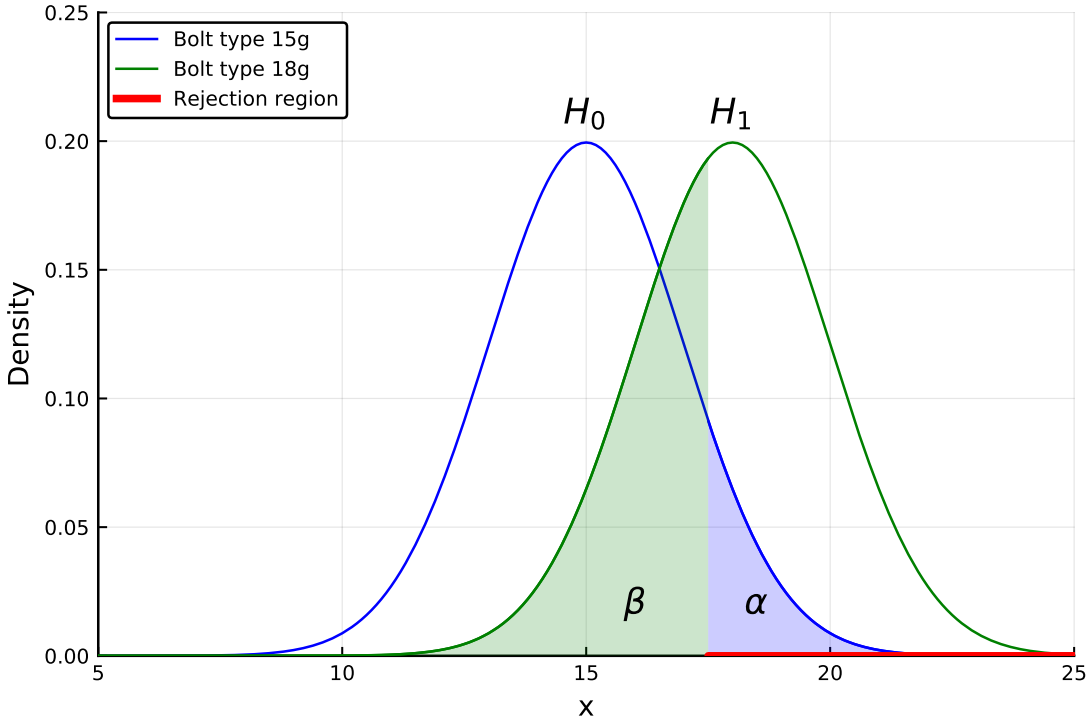

Figure 5.13: Type I (blue) and Type II (green) errors. The rejection region based from $\tau = 17.5$ to the right is colored with red on the horizontal axis.

Simple Hypothesis Tests

When the alternative parameter spaces Θ_0 and Θ_1 are only comprised of a single point each, the hypothesis test is called a *simple hypothesis test*. Such a test is often not of great practical use, but we introduce it here for pedagogical purposes. Specifically, by analyzing such tests we can understand how type I and type II errors interplay.

As an introductory example, consider a container that contains two identical types of pipes, except that one type weighs 15 grams on average and the other 18 grams on average. The standard deviation of the weights of both pipe types is 2 grams. Imagine now that we sample a single pipe, and wish to determine its type. Denote the weight of this pipe by the random variable X. For this example we devise the following statistical hypothesis test: $\Theta_0 = \{15\}$ and $\Theta_1 = \{18\}$. Now, given a threshold τ, we reject H_0 if $X > \tau$, otherwise we retain H_0.

In this circumstance, we can explicitly analyze the probabilities of both the type I and type II errors, α and β respectivily. Listing 5.15 below generates Figure 5.13, which illustrates this graphically for $\tau = 17.5$. You may try to modify the value of `tau` in the code to see how the probabilities for type I and type II errors vary.

Listing 5.15: A simple hypothesis test

```
1  using Distributions, StatsBase, Plots, LaTeXStrings; pyplot()
2
3  mu0, mu1, sd, tau  = 15, 18, 2, 17.5
4  dist0, dist1 = Normal(mu0,sd), Normal(mu1,sd)
5  grid = 5:0.1:25
6  h0grid, h1grid = tau:0.1:25, 5:0.1:tau
7
8  println("Probability of Type I error: ", ccdf(dist0,tau))
9  println("Probability of Type II error: ", cdf(dist1,tau))
10
11 plot(grid, pdf.(dist0,grid),
12         c=:blue, label="Bolt type 15g")
13 plot!(h0grid, pdf.(dist0, h0grid),
14         c=:blue, fa=0.2, fillrange=[0 1], label="")
15 plot!(grid, pdf.(dist1,grid),
16         c=:green, label="Bolt type 18g")
17 plot!(h1grid, pdf.(dist1, h1grid),
18         c=:green, fa=0.2, fillrange=[0 1], label="")
19 plot!([tau, 25],[0,0],
20         c=:red, lw=3, label="Rejection region",
21         xlims=(5, 25), ylims=(0,0.25) , legend=:topleft,
22     xlabel="x", ylabel="Density")
23 annotate!([(16, 0.02, text(L"\beta")),(18.5, 0.02, text(L"\alpha")),
24             (15, 0.21, text(L"H_0")),(18, 0.21, text(L"H_1"))])
```

```
Probability of Type I error: 0.10564977366685525
Probability of Type II error: 0.4012936743170763
```

In line 3 we set the parameters of the example. In line 4 we define the distributions under H_0 and H_1. Line 6 sets girds of values that are used for plotting type I and type II error ranges. In lines 8-9 we compute α and β using `ccdf()` and `cdf()` on `dist0` and `dist1` respectively. The remainder of the code creates the figure using the `pdf()` function. Notice the calls to `plot!()` in lines 13-14 and 17-18 using the `fillrange` argument.

The Receiver Operating Curve

In the previous example, $\tau = 17.5$ was arbitrarily chosen. Clearly if τ was increased the probability of making a Type I error, α, would decrease, while the probability of making a type II error, β, would increase. Conversely if we decreased τ the reverse would occur. We now introduce the *Receiver Operating Curve* (ROC), also sometime called the *receiver operating characteristic curve*. It is a tool that helps to visualize the tradeoff between type I and type II errors. It allows one to visualize the error tradeoffs for all possible τ values simultaneously, for a particular alternative hypothesis H_1.

We look at three different scenarios for μ_1 : 16, 18, and 20. Clearly, the bigger the difference between μ_0 and μ_1, the easier it should be to make a decision without errors. In Listing 5.16 we consider each scenario and shift τ, and in the process plot the analytic coordinates of $\big(\alpha(\tau), 1-\beta(\tau)\big)$. This is the ROC. It is a *parametric plot* of the probability of a type I error and power. The results are in Figure 5.14. ROCs are also a way of comparing different sets of hypotheses simultaneously.

By plotting several different ROCs on the same figure, we can compare the likelihood of making errors for various scenarios of different μ_1's.

To better understand how the ROCs are generated, consider also Figure 5.13 and imagine the effect of sliding τ. In this figure, the shaded blue area represents α, while 1 minus the green area represents power. If one considers $\tau = 25$, then both α and power are almost zero, and this corresponds (approximately) to $(0, 0)$ in Figure 5.14. Now, as the τ threshold is slowly decreased, it can be seen that the power increases at a much faster rate than α, and this behavior is observed in the ROC. In addition, as the difference in means between the null and alternative hypotheses are greater, the ROC curves are shown to be pushed "further out" from the diagonal dashed line, reflecting the fact that such alternative sets of hypotheses are easier to detect.

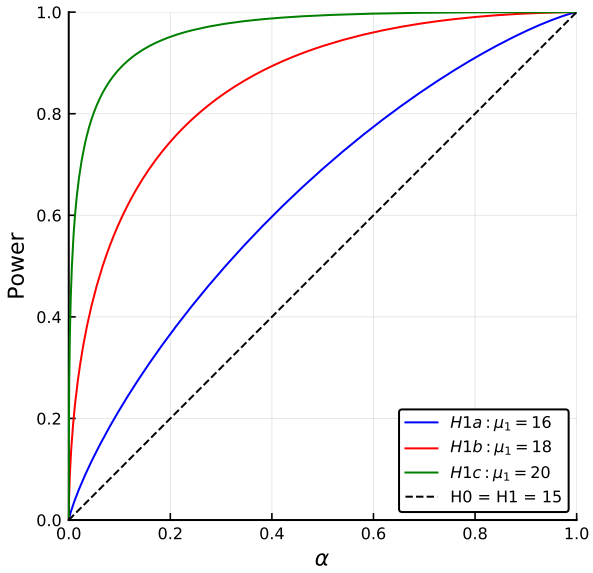

Figure 5.14: Three ROCs for various points within H_1.

Listing 5.16: Comparing receiver operating curves

```
using Distributions, StatsBase, Plots, LaTeXStrings; pyplot()

mu0, mu1a, mu1b, mu1c, sd = 15, 16, 18, 20, 2
tauGrid = 5:0.1:25

dist0 = Normal(mu0,sd)
dist1a, dist1b, dist1c  = Normal(mu1a,sd), Normal(mu1b,sd), Normal(mu1c,sd)

falsePositive = ccdf.(dist0,tauGrid)
truePositiveA, truePositiveB, truePositiveC =
    ccdf.(dist1a,tauGrid), ccdf.(dist1b,tauGrid), ccdf.(dist1c,tauGrid)

plot(falsePositive, [truePositiveA truePositiveB truePositiveC],
    c=[:blue :red :green],
    label=[L"H1a: \mu_1 = 16" L"H1b: \mu_1 = 18" L"H1c: \mu_1 = 20"])
plot!([0,1], [0,1], c=:black, ls=:dash, label="H0 = H1 = 15",
    xlims=(0,1), ylims=(0,1), xlabel=L"\alpha", ylabel="Power",
    ratio=:equal, legend=:bottomright)
```

The range `tauGrid` presents the range of possible values for τ that are used. The distribution `dist0` is for H_0 and the distributions `dist1a`, `dist1b` and `dist1c` are for three variants of H_1. The plots are then plots of `falsePositives` vs. `truePositiveA`, `truePositiveB` or `truePositiveC`. The essence of the plotting code is to use `falsePositives` for arguments of the horizontal coordinate and `truePositiveA`, `truePositiveB` or `truePositiveC` as arguments of the vertical coordinate. This creates a parametric plot. Lines 16-18 plot a diagonal dashed line. This line represents the extreme case of the distributions of H_0 and H_1 directly overlapping. In this case, the probability of a Type I error is the same as the power.

A Randomized Hypothesis Test

We now investigate the concept of a *randomization test*, which is a type of *non-parametric test*, i.e. a statistical test which does not require that we know what type of distribution the data comes from. A virtue of non-parametric tests is that they do not impose a specific model. Consider the following example, where a farmer wants to test whether a new fertilizer is effective at increasing the yield of her tomato plants. As an experiment, she took 20 plants, kept 10 as controls and treated the remaining 10 with fertilizer. After two months, she harvested the plants, and recorded the yield of each plant (in kg) as shown in Table 5.2.

Control	4.17	5.58	5.18	6.11	4.5	4.61	5.17	4.53	5.33	5.14
Fertilizer	6.31	5.12	5.54	5.5	5.37	5.29	4.92	6.15	5.8	5.26

Table 5.2: Yield in kg for 10 plants with, and 10 plants without fertilizer (control).

It can be observed that the group of plants treated with fertilizer have an average yield 0.494 kg greater than that of the control group. One could argue that this difference is due to the effects of the fertilizer. We now investigate if this is a reasonable assumption. Let us assume for a moment that the fertilizer had no effect on plant yield (H_0), and that the result was simply due to random chance. In such a scenario, we actually have 20 observations from the same group, and regardless of how we arrange our observations we would expect to observe similar results.

Hence we can investigate the likelihood of this outcome occurring by random chance, by considering all possible *combinations* of 10 samples from our group of 20 observations, and counting how many of these combinations result in a difference in sample means greater than or equal to 0.494 kg. The proportion of times this occurs is analogous to the likelihood that the difference we observe in our sample means was purely due to random chance. It is in a sense the p-value.

Before proceeding we calculate the number of ways one can sample $r = 10$ unique items from $n = 20$ total, which is given by,

$$\binom{20}{10} = 184,756.$$

Hence the number of possible combinations in our example is computationally manageable. Note that in a different situation where n and r would be bigger, e.g. $n = 40$ and $r = 20$, the number of combinations would be too big for an exhaustive search (about 137 billion). In such a case, a viable alternative is to randomly sample combinations for estimating the p-value.

In Listing 5.17 we use Julia's `Combinatorics` package to enumerate the difference in sample means for every possible combination. From the output we observe that only 2.39% of all possible combinations result in a sample mean greater than or equal to our treated group, i.e. a difference greater than or equal to 0.494 kg. Therefore there is significant statistical evidence that the fertilizer increases the yield of the tomato plants, since under H_0, there is only a 2.39% chance of obtaining this value or greater by random chance.

Listing 5.17: A randomized hypothesis test

```
1  using Combinatorics, Statistics, DataFrames, CSV
2
3  data = CSV.read("../data/fertilizer.csv")
4  control = data.Control
5  fertilizer = data.FertilizerX
6
7  subGroups = collect(combinations([control;fertilizer],10))
8
9  meanFert = mean(fertilizer)
10 pVal = sum([mean(i) >= meanFert for i in subGroups])/length(subGroups)
11 println("p-value = ", pVal)
```

```
p-value = 0.023972157873086666
```

We use the `Combinatorics` package for the `combinations()` function. In line 3-5 we import our data, and store the data for the control and fertilized groups in the arrays `control` and `fertilizer`. In line 7 all observations are concatenated into one array via the use of `[ ; ]`. Following this, the `combinations()` function is used to generate an iterator object for all combinations of 10 elements from our 20 observations. The `collect()` function then converts this iterator into an array of all possible combinations of 10 objects, sampled from 20 total. This array of all combinations is stored as `subGroups`.In line 9, the mean of the fertilizer group is calculated and assigned to the variable `meanFert`. In line 10 the mean of each combination in the array `x` is calculated and compared against `meanFert`. The proportion of means which are greater than or equal to `meanFert` is then calculated through the use of a comprehension, and the functions `sum()` and `length()`.

5.7 A Taste of Bayesian Statistics

In this section we briefly explore the Bayesian approach to statistical inference as an alternative to the frequentist view of statistics which was introduced in Sections 5.4, 5.5 and 5.6, and used throughout the remainder of the book. In the Bayesian paradigm, the (scalar or vector) parameter, θ is not assumed to exist as some fixed unknown quantity, but instead is assumed to follow a distribution. That is, the parameter itself is a random variable, and the act of *Bayesian inference* is the process of obtaining more information about the distribution of θ. Such a setup is useful in many practical situations since it allows one to incorporate prior beliefs about the parameter, before experience from new observations is taken into consideration. It also allows one to carry out repeated inference in a very natural manner by allowing inference in future periods to rely on past experience or past data.

The key objects at play are the *prior distribution* of the parameter and the *posterior distribution*

of the parameter. The former is postulated beforehand, or exists as a consequence of previous inference, while the latter captures the distribution of the parameter after observations are taken into account. The relationship between the prior and the posterior is then,

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{evidence}} \qquad \text{or} \qquad f(\theta \mid x) = \frac{f(x \mid \theta) \times f(\theta)}{\int f(x \mid \theta) f(\theta)\, d\theta}\ . \tag{5.20}$$

This is nothing but Bayes' rule applied to densities. Here the prior distribution (density) is $f(\theta)$ and the posterior distribution (density) is $f(\theta \mid x)$. Observe that the denominator, known as *evidence* or *marginal likelihood*, is constant with respect to the parameter θ. This allows the equation to be written as,

$$f(\theta \mid x) \propto f(x \mid \theta) \times f(\theta), \tag{5.21}$$

where the symbol "$\propto$" denotes "proportional to". Hence the posterior distribution can be easily obtained up to the normalizing constant (the evidence) by multiplying the prior with the likelihood, $f(x \mid \theta)$.

In general, carrying out *Bayesian inference* involves the following steps:

1. Assume some distributional model for the parameters θ.

2. Use previous inference experience, elicit an expert, or make an educated guess to determine a prior distribution for the parameter, $f(\theta)$. The prior distribution might be parameterized by its own parameters, called *hyperparameters*.

3. Collect data x, and an expression or a computational mechanism for the likelihood $f(x \mid \theta)$ based on the distributional model chosen.

4. Use the relationship (5.20) to obtain the posterior distribution of the parameters, $f(\theta \mid x)$. In most cases, the evidence (denominator of (5.20)) is not easily computable. Hence the posterior distribution is only available up to a normalizing constant. In some special cases the form of the posterior distribution is the same as the prior distribution. In such cases, *conjugacy* holds, the prior is called a *conjugate prior*, and the hyperparameters are updated from prior to posterior.

5. The posterior distribution can then be used to make conclusions about the model. For example, if a single specific parameter value is needed to make the model concrete, a *Bayes estimate* based on the posterior distribution, such as for example the *posterior mean*, may be computed:

$$\hat{\theta} = \int \theta f(\theta \mid x)\, d\,\theta. \tag{5.22}$$

   Further analyses such as obtaining *credible intervals*, similar to confidence intervals, may also be carried out. See a brief discussion in Section 6.7.

6. The model with $\hat{\theta}$ can then be used for making conclusions. Alternatively, a whole class of models based on the posterior distribution $f(\theta \mid x)$ can be used. This often goes hand in hand with simulation as one is able to generate Monte Carlo samples from the posterior distribution.

Bayesian inference has gained significant popularity over the past few decades and has evolved together with the whole field of *computational statistics*. Unless conjugacy holds, there is typically not an explicit expression for the evidence (the integral in (5.20)) and hence a computational challenge is to make use of the posterior available only up to a normalizing constant. We now elaborate on the details through variants of a very simple example in order to understand the main concepts. For a general treatment of Bayesian inference we recommend [Rob07].

A Simple Poisson Example

Consider an example where an insurance company models the number of weekly fires in a city using a Poisson distribution with parameter λ. Here λ is also the expected number of fires per week. Assume that the following data is collected over a period of 16 weeks,

$$x = (x_1, \dots, x_{16}) = (2, 1, 0, 0, 1, 0, 2, 2, 5, 2, 4, 0, 3, 2, 5, 0).$$

Each data point indicates the number of fires per week. In this case the MLE is $\hat{\lambda} = 1.8125$ simply obtained by the sample mean. Hence in a frequentist approach, after 16 weeks the distribution of the number of fires per week is modeled by a Poisson distribution with $\lambda = 1.8125$. One can then obtain estimates for say, the probability of having more than 5 fires in a given week as follows:

$$\mathbb{P}(\text{fires per week} > 5) = 1 - \sum_{k=0}^{5} e^{-\lambda} \frac{\lambda^k}{k!} \approx 0.0107. \tag{5.23}$$

However, the drawback of such an approach in estimating λ is that it didn't make use of previous information. By comparison, in a Bayesian approach the estimate would allow one to incorporate information from previous years, or alternatively from adjacent geographical areas. Say that for example, further knowledge comes to light that the number of fires per week ranges between 0 and 10 and that the typical number is 2 fires per week. In this case one can assign a prior distribution to λ that captures this belief. Here is where some critics claim that such use of Bayesian statistics turns into somewhat of a "voodoo science" since we have an infinite number of options to choose for the prior. Still, it is often useful.

Anyways, you also have an infinite number of choices for the model no matter what approach you use! Truth is you are always using some prior info.

One could easily argue that the MLE assumes that 100000 fires per week is just as likely as 1, which is not only imposing prior assumptions, but prior assumptions which are wrong and possibly allow for higher values than are plausible for small datasets.

Anyways, priors are just a fancy form of regularization really. Bayesian approaches tend to be most beneficial in small data regimes.

In our example, assume that we decide to use a triangular distribution as shown in blue in Figure 5.15. Such a triangular distribution captures prior beliefs about the parameter λ well, because it has a defined range and a defined mode.

With the prior assigned and the data collected, we can use the machinery of Bayesian inference of (5.20). In this specific case the prior distribution of the parameter λ is the triangular distribution

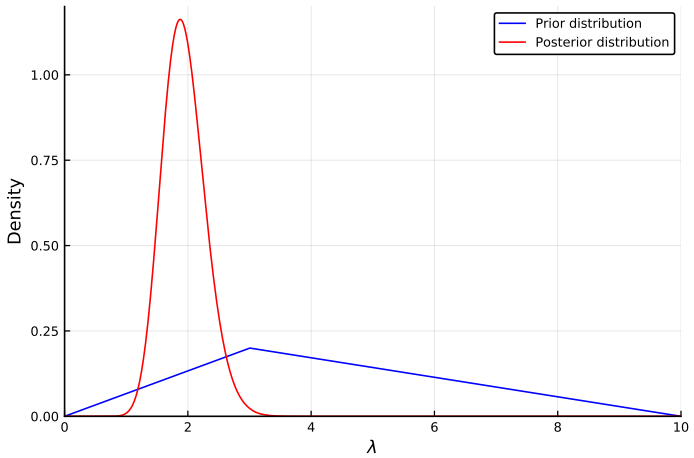

Figure 5.15: The prior distribution in blue and the posterior in red for Bayesian estimation of a Poisson distribution.

with the PDF,

$$f(\lambda) = \begin{cases} \frac{1}{15}\lambda, & \lambda \in [0, 3], \\ \frac{1}{35}(10 - \lambda), & \lambda \in (3, 10]. \end{cases}$$

With the 16 observations, $x_1, \ldots, x_{16}$, the likelihood is,

$$f(\lambda \mid x) = \prod_{k=1}^{16} e^{-\lambda} \frac{\lambda^{x_k}}{x_k!}.$$

Hence the posterior is proportional to $f(\lambda \mid x) f(\lambda)$. However, normalization of this function in of λ, requires dividing it by the evidence, given by,

$$\int_0^{10} f(x \mid \lambda) f(\lambda)\, d\lambda.$$

Typically this integral isn't easy to evaluate analytically, hence numerical methods are often used. For illustration purposes, we carry out this numerical integration as part of Listing 5.18 where we also plot the resulting posterior distribution (red curve in Figure 5.15). To appreciate potential problems with such a numerical solution, imagine cases where the parameter θ is not just the scalar λ but rather consists of multiple dimensions. The integral of the evidence cannot be efficiently computed in such cases.

In Listing 5.18, once the prior distribution is obtained, we compute its mean to obtain a Bayes estimate for λ. The value obtained differs from the MLE obtained above and hence probability estimates using the model, such as (5.23) would also vary. Importantly, by employing the Bayesian perspective, we were able to incorporate prior knowledge into the inference procedure.

Listing 5.18: Bayesian inference with a triangular prior

```
1  using Distributions, Plots, LaTeXStrings; pyplot()
2
3  prior(lam) = pdf(TriangularDist(0, 10, 3), lam)
4  data = [2,1,0,0,1,0,2,2,5,2,4,0,3,2,5,0]
5
6  like(lam) = *([pdf(Poisson(lam),x) for x in data]...)
7  posteriorUpToK(lam) = like(lam)*prior(lam)
8
9  delta = 10^-4.
10 lamRange = 0:delta:10
11 K = sum([posteriorUpToK(lam)*delta for lam in lamRange])
12 posterior(lam) = posteriorUpToK(lam)/K
13
14 bayesEstimate = sum([lam*posterior(lam)*delta for lam in lamRange])
15 println("Bayes estimate: ",bayesEstimate)
16
17 plot(lamRange, prior.(lamRange),
18         c=:blue, label="Prior distribution")
19 plot!(lamRange, posterior.(lamRange),
20         c=:red, label="Posterior distribution",
21         xlims=(0, 10), ylims=(0, 1.2),
22         xlabel=L"\lambda",ylabel="Density")
```

```
Bayes estimate: 1.9371887551439297
```

In line 3 we define the prior. In line 4 we set the data values. In line 6 the likelihood function is defined. Notice that the $\star$ operator is used as a function, and that the splat operator, `...` is applied inside the brackets. Equation (5.21) is implemented in Line 7, while lines 9-11 are used to numerically compute the evidence. The actual posterior is defined in line 12. In line 14 a Bayes estimate from the prior is calculated, according to (5.22) and printed in line 15. The remainder of the code creates Figure 5.15.

Conjugate Priors

Following on from the previous example, a natural question arises: why use the specific form of the prior distribution that we used? After all, the results would vary if we were to choose a different prior. While in generality Bayesian statistics doesn't supply a complete answer, there are cases where certain families of prior distributions work very well with certain (other) families of statistical models.

For example, in our case of a Poisson probability distribution model, it turns out that assuming a gamma prior distribution works nicely. This is because the resulting posterior distribution is also guaranteed to be gamma. In such a case, the gamma distribution is said to be a *conjugate prior* to the Poisson distribution. The parameters of the prior/posterior distribution are called *hyperparameters*, and by exhibiting a conjugate prior distribution relationship, the hyperparameters typically have a simple update law from prior to posterior. This relieves a huge computational burden.

To see this in the case of gamma-Poisson, assume the hyperparameters of the prior to have α

(shape parameter) and β (rate parameter). Now using the Poisson likelihood and the gamma PDF we obtain:

$$\begin{aligned} \text{posterior} &\propto \left(\prod_{k=1}^{n} e^{-\lambda}\frac{\lambda^{x_k}}{x_k!}\right)\frac{\beta^{\alpha}}{\Gamma(\alpha)}\lambda^{\alpha-1}e^{-\beta\lambda} \\ &\propto e^{-n\lambda}\lambda^{\sum_{k=1}^{n} x_k}\lambda^{\alpha-1}e^{-\beta\lambda} \\ &= \lambda^{\alpha+\sum_{k=1}^{n} x_k-1}e^{-\lambda(\beta+n)} \\ &\propto \text{gamma density with shape parameter } \alpha+\sum x_i \text{ and scale parameter } \beta+n. \end{aligned} \tag{5.24}$$

This shows us the gamma-Poisson conjugacy and implies a slick update rule for the hyperparameters: The hyperparameter α is updated to $\alpha+\sum x_i$ and the hyperparameter β is updated to $\beta+n$.

In Listing 5.19 we use a gamma prior with prior parameters of $\alpha=8$ and $\beta=2$. For illustration, we compute the posterior both using the brute force method of the previous listing and using the simple hyperparameter update rule due to conjugacy. The posterior and prior are plotted in Figure 5.16.

Listing 5.19: Bayesian inference with a gamma prior

```
using Distributions, Plots; pyplot()

alpha, beta = 8, 2
prior(lam) = pdf(Gamma(alpha, 1/beta), lam)
data = [2,1,0,0,1,0,2,2,5,2,4,0,3,2,5,0]

like(lam) = *([pdf(Poisson(lam),x) for x in data]...)
posteriorUpToK(lam) = like(lam)*prior(lam)

delta = 10^-4.
lamRange = 0:delta:10
K = sum([posteriorUpToK(lam)*delta for lam in lamRange])
posterior(lam) = posteriorUpToK(lam)/K

bayesEstimate = sum([lam*posterior(lam)*delta for lam in lamRange])

newAlpha, newBeta = alpha + sum(data), beta + length(data)
closedFormBayesEstimate = mean(Gamma(newAlpha, 1/newBeta))

println("Computational Bayes Estimate: ", bayesEstimate)
println("Closed form Bayes Estimate: ", closedFormBayesEstimate)

plot(lamRange, prior.(lamRange),
        c=:blue, label="Prior distribution")
plot!(lamRange, posterior.(lamRange),
        c=:red, label="Posterior distribution",
        xlims=(0, 10), ylims=(0, 1.2),
        xlabel=L"\lambda",ylabel="Density")
```

```
Computational Bayes Estimate: 2.055555555555556
Closed form Bayes Estimate: 2.0555555555555554
```

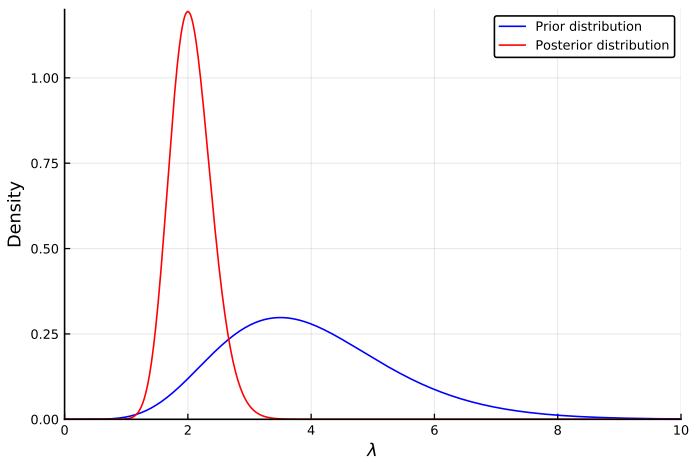

Figure 5.16: The prior and posterior for Bayesian estimation of a Poisson distribution using gamma conjugacy.

In lines 3 the prior hyperparameters are defined and in line 4 the prior distribution is defined. In lines 7-13 the posterior is calculated in the brute force same manner as listing 5.18. Similarly in line 15 we compute the Bayes estimate in the same manner. Line 17 is where the simplicity of conjugacy comes about, the hyperparameters are updated according to the conjugacy rule. Then in line 18 `closedFormBayesEstimate` is computed just using the formula for the mean of a gamma distribution (using `mean()` from `Distributions`). The bayes estimates are printed in lines 20-21 and the remaining code lines create Figure 5.16.

Markov Chain Monte Carlo

In many applicative cases of Bayesian statistics, convenient situations of conjugate priors are not available, yet computation of posterior distributions and Bayes estimates are needed. In cases where the dimension of the parameter space is high, carrying out straightforward integration as done in Listing 5.18 is not possible. However, there are other ways of carrying out Bayesian inference. One such popular way is by using algorithms that fall under the category known as *Markov Chain Monte Carlo*, MCMC, also known as *Monte Carlo Markov Chain* (with a different word order).

The *Metropolis–Hastings* algorithm is one such popular MCMC algorithm. It produces a series of samples $\theta(1), \theta(2), \theta(3), \ldots$, where it is guaranteed that for large t, $\theta(t)$ is distributed according to the posterior distribution. Technically, the random sequence $\{\theta(t)\}_{t=1}^{\infty}$ is a Markov chain (see Chapter 9 for more details about Markov chains) and it is guaranteed that the stationary distribution of this Markov chain is the specified posterior distribution. That is, the posterior distribution is an input parameter to the algorithm.

The major benefit of Metropolis–Hastings and similar MCMC algorithms is that they only uses ratios of the posterior on different parameter values. For example, for parameter values θ_1 and θ_2, the algorithm only uses the posterior distribution via the ratio,

$$L(\theta_1, \theta_2) = \frac{f(\theta_1 \mid x)}{f(\theta_2 \mid x)}.$$

This means that the normalizing constant (evidence) is not needed as it is implicitly cancelled out. Thus using the posterior in the proportional form (5.21) suffices.

Further to the posterior distribution, an additional input parameter to Metropolis–Hastings is the so-called *proposal density*, denoted by $q(\cdot \mid \cdot)$. This is a family of probability distributions where given a certain value of θ_1 taken as a parameter, the new value, say θ_2, is distributed with PDF,

$$q(\theta_2 \mid \theta_1).$$

The idea of Metropolis–Hastings is to walk around the parameter space by randomly generating new values using $q(\cdot \mid \cdot)$. Then some new values are 'accepted' while others are not, all with a manner which ensures the desired limiting behavior. The algorithm specification is to accept with probability,

$$H = \min\Big\{1, \quad L\big(\theta^*, \theta(t)\big)\ \frac{q\big(\theta(t) \mid \theta^*\big)}{q\big(\theta^* \mid \theta(t)\big)}\Big\},$$

where θ^* is the new proposed value, generated via $q\big(\ \cdot \mid \theta(t)\big)$, and $\theta(t)$ is the current value. With each such iteration, the new value is accepted with probability H and otherwise rejected. With certain technical requirements on the posterior and proposal densities, the theory of Markov chains then guarantees that the stationary distribution of the sequence $\{\theta(t)\}$ is the posterior distribution.

Different variants of the Metropolis–Hastings algorithm employ different types of proposal densities. There are also generalizations and extensions that we don't discuss here, such as *Gibbs Sampling* and *Hamiltonian Monte Carlo* for example.

To help illustrate some of these concepts, we now implement a simple version of Metropolis–Hastings where we use the *folded normal distribution* as a proposal density. This distribution is achieved by taking a normal random variable X with mean μ and variance σ^2 and considering $Y = |X|$. In this case, the PDF of Y is,

$$f(y) = \frac{1}{\sigma\sqrt{2\pi}}\Big(e^{-\frac{(y-\mu)^2}{2\sigma^2}} + e^{-\frac{(y+\mu)^2}{2\sigma^2}}\Big). \tag{5.25}$$

Our choice of this specific density is purely for simplicity of implementation, and in addition it suits the case that we demonstrate, where the support of the parameter in question is non-negative.

In Listing 5.20 we implement Metropolis–Hastings for the same data and prior as the previous example, Listing 5.19. In such an example one would not use to use MCMC since conjugacy is much more efficient, however we do so here for purpuses of comparison. Our results show that we obtain the same numerical results as we did using gamma conjugacy. The histogram of the samples is plotted in Figure 5.17.

.

Listing 5.20: Bayesian inference using MCMC

```
using Distributions, Plots; pyplot()

alpha, beta = 8, 2
prior(lam) = pdf(Gamma(alpha, 1/beta), lam)
data = [2,1,0,0,1,0,2,2,5,2,4,0,3,2,5,0]

like(lam) = *([pdf(Poisson(lam),x) for x in data]...)
posteriorUpToK(lam) = like(lam)*prior(lam)

sig = 0.5
foldedNormalPDF(x,mu) = (1/sqrt(2*pi*sig^2))*(exp(-(x-mu)^2/2sig^2)
                                              + exp(-(x+mu)^2/2sig^2))
foldedNormalRV(mu) = abs(rand(Normal(mu,sig)))

function sampler(piProb,qProp,rvProp)
    lam = 1
    warmN, N = 10^5, 10^6
    samples = zeros(N-warmN)

    for t in 1:N
        while true
            lamTry = rvProp(lam)
            L = piProb(lamTry)/piProb(lam)
            H = min(1,L*qProp(lam,lamTry)/qProp(lamTry,lam))
            if rand() < H
                lam = lamTry
                if t > warmN
                    samples[t-warmN] = lam
                end
                break
            end
        end
    end
    return samples
end

mcmcSamples = sampler(posteriorUpToK,foldedNormalPDF,foldedNormalRV)
println("MCMC Bayes Estimate: ",mean(mcmcSamples))

stephist(mcmcSamples, bins=100,
        c=:black, normed=true, label="Histogram of MCMC samples")

lamRange = 0:0.01:10
plot!(lamRange, prior.(lamRange),
        c=:blue, label="Prior distribution")

closedFormPosterior(lam)=pdf(Gamma(alpha + sum(data),1/(beta+length(data))),lam)
plot!(lamRange, closedFormPosterior.(lamRange),
        c=:red, label="Posterior distribution",
        xlims=(0, 10), ylims=(0, 1.2),
    xlabel=L"\lambda",ylabel="Density")
```

```
MCMC Bayes Estimate: 2.065756632471559
```

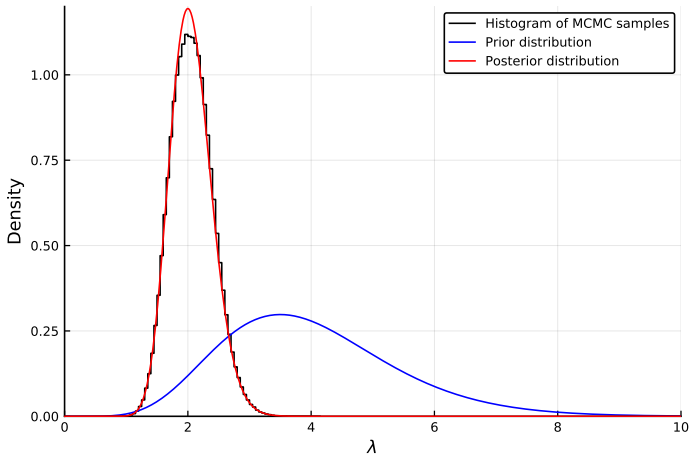

Figure 5.17: The prior and the posterior for Monte Carlo Markov Chain samples generated using Metropolis–Hastings

Lines 3-8 are similar to the previous listings 5.18 and 5.19. In ines 10-13 the proposal density `foldedNormalPDF()` is define d in accordance with (5.25), along with a function for generating a proposal random variable, `foldedNormalRV()`. Lines 15-35 define the function `sampler()`. It operates on a desired (non-normalized) density, `piProb` and runs the Metropolis–Hastings algorithm for sampling from that density. The argument, `qProp`, is the proposal density and the argument `rvProp` is for generating from the proposal. All three arguments are assumed to be functions which `sampler()` invokes. Our implementation uses a *warm up sequence* with a length specified by `warmN` in line 17. The idea here is to let the algorithm run for a while to remove any bias introduced by initial values. Lines 20-33 constitute the main loop over N samples generated by the algorithm. In our implementation, we setup an internal loop (lines 21-32) that iterates until a proposal is accepted (and breaks in line 30). Line 45 prints the Bayes estimate. As can be seen, it agrees with the estimate of Listing 5.19.

Chapter 6

Confidence Intervals - DRAFT

In this chapter we cover a variety of confidence intervals used in standard statistical procedures. As introduced in Section 5.5, a confidence interval with a confidence level $1-\alpha$ is an interval $[L, U]$ resulting from the observations. When considering confidence intervals in the setting of symmetric sampling distributions (as is the case for most of this chapter), a typical formula for $[L, U]$ is of the form,

$$\hat{\theta} \pm K_\alpha \; s_{\text{err}}. \tag{6.1}$$

Here $\hat{\theta}$ is typically the point estimate for the parameter in question, s_{err} is some measure or estimate of the variability (e.g. standard error), and K_α is a constant which depends on the model at hand and on α. Typically by decreasing $\alpha \to 0$, we have that K_α increases, implying a wider confidence interval. For the examples in this chapter, common values for K_α are in the range of $[1.5, 3.5]$ for values of α in the range of $[0.01, 0.1]$. Most of the confidence intervals presented in this chapter follow the form of (6.1), with the specific form of K_α often depending on conditions such as:

Sample size: Is the sample size small or large.

Variance: Is the variance σ^2 known or unknown.

Distribution: Is the population assumed normally distributed or not.

In exploring confidence intervals with Julia, we compute the confidence intervals using standard statistical formulas and then illustrate how they can be obtained using the `HypothesisTests` package. This package includes various functions that generate objects resulting from specific statistical procedures. We can either look at the output of these objects, or query them using other functions, specifically the `confint()` function. This package is also used extensively in Chapter 7.

The individual sections of this chapter focus on specific confidence intervals and general concepts. In Section 6.1 we cover confidence intervals for the mean of a single population. In Section 6.2 we present comparisons of means of two populations. In Section 6.3 we cover confidence intervals for proportions. In Section 6.4 we gain a better understanding of model assumptions via the example of a confidence interval for the variance. In Section 6.5 we present the bootstrap method, a general methodology for creating confidence intervals. In Section 6.6 we present *prediction intervals*, a concept dealing with prediction of future observations based on previous ones. We close with Section 6.7 which deals with credible intervals from Bayesian statistics.

6.1 Single Sample Confidence Intervals for the Mean

Let us first consider the case where we wish to estimate the population mean μ using a random sample, $X_1, \ldots, X_n$. As covered previously, a point estimate for the mean is the sample mean $\overline{X}$. A typical formula for the confidence interval of the mean is then,

$$\overline{X} \pm K_\alpha \frac{S}{\sqrt{n}}. \tag{6.2}$$

Here the bounds around the point estimator $\overline{X}$ are defined by the addition and subtraction of a multiple, K_α, of the standard error, $S/\sqrt{n}$, first introduced in Section 4.2. The multiple K_α takes on different forms depending on the specific case at hand.

Population Variance Known

If we assume that the population variance, σ^2, is known and the data is normally distributed, then the sample mean $\overline{X}$ is normally distributed with mean μ and variance σ^2/n. This yields,

$$\mathbb{P}\Big(\mu - z_{1-\frac{\alpha}{2}} \frac{\sigma}{\sqrt{n}} \leq \overline{X} \leq \mu + z_{1-\frac{\alpha}{2}} \frac{\sigma}{\sqrt{n}}\Big) = 1 - \alpha, \tag{6.3}$$

where $z_{1-\frac{\alpha}{2}}$ is the $1 - \frac{\alpha}{2}$ quantile of the standard normal distribution. In Julia this is computed via `quantile(Normal(),1-alpha/2)`. If we denote the actual sample mean estimate obtained from data by $\overline{x}$, then by rearranging the inequalities inside the probability statement above, we obtain the following confidence interval formula,

$$\overline{x} \pm z_{1-\frac{\alpha}{2}} \frac{\sigma}{\sqrt{n}}. \tag{6.4}$$

In practice σ^2 is rarely known, hence it is tempting to replace σ by s (sample standard deviation), in the formula above. Such a replacement is generally fine for large samples. However, in the case of small samples, one should confidence intervals assuming population variance unknown, covered at the end of this section.

The validity of the normality assumption should also be considered. In cases where the data is not normally distributed, the probability statement (6.3) only approximately holds. However as $n \to \infty$, it quickly becomes precise due to the central limit theorem. Hence the confidence interval (6.4) may be used for non-small samples.

In Julia, computation of confidence intervals is done using functions from the `HypothesisTests` package (even when we don't carry out an hypothesis test). The code in Listing 6.1 illustrates computation of the confidence interval (6.4) using both the package and by evaluating the formula directly. It can be observed that both the direct computation and the use of the `confint()` function yield the same result.

Listing 6.1: CI for single sample population, variance assumed known

```
1  using CSV, Distributions, HypothesisTests
2
3  data = CSV.read("../data/machine1.csv", header=false)[:,1]
4  xBar, n = mean(data), length(data)
5  sig = 1.2
6  alpha = 0.1
7  z = quantile(Normal(),1-alpha/2)
8
9  println("Calculating formula: ", (xBar - z*sig/sqrt(n), xBar + z*sig/sqrt(n)))
10 println("Using confint() function: ", confint(OneSampleZTest(xBar,sig,n),alpha))
```

```
Calculating formula:      (52.51484557853184, 53.397566664027984)
Using confint() function: (52.51484557853184, 53.397566664027984)
```

Line 3 loads the data. Note the use of the `header=false` argument, and also the trailing `[:,1]` which is used to select all rows of the data. In line 4 we calculate the sample mean, and the number of observations. In line 5, we stipulate the standard deviation as 1.2, as this scenario is one in which the population standard deviation, or population variance, is assumed known. In line 7 we calculate the value of z for $1-\alpha/2$. This quantity does not depend on the sample but is a fixed number. As is well known from statistical tables it equals approximately 1.65 when $\alpha = 10\%$. In line 9 the formula for the confidence interval (6.4) is evaluated directly. In line 10 the function `OneSampleZTest()` is first used to conduct a one sample z-test given the parameters `xBar`, `sig`, and `n`. The `confint()` function is then applied to this output, for the specified value of `alpha`. It can be observed that the two methods are in agreement. Note that hypothesis tests are covered further in Chapter 7.

Population Variance Unknown

A celebrated procedure in elementary statistics is the confidence interval based on the T-distribution. Here we relax the assumptions of the previous confidence interval by allowing σ^2 to be an unknown quantity. In this case, if we replace σ by the sample standard deviation s, then the probability statement (6.3) no longer holds. However, by using the T-distribution (see Section 5.2) we are able to correct the confidence interval to,

$$\overline{x} \pm t_{1-\frac{\alpha}{2},n-1}\ \frac{s}{\sqrt{n}}. \tag{6.5}$$

Here, $t_{1-\frac{\alpha}{2},n-1}$ is the $1-\frac{\alpha}{2}$ quantile of a T-distribution with $n-1$ degrees of freedom. This can be calculated in Julia via `quantile(TDist(n-1),1-alpha/2)`.

For small samples, the replacement of $z_{1-\frac{\alpha}{2}}$ from (6.4) by $t_{1-\frac{\alpha}{2},n-1}$ in (6.5) significantly affects the width of the confidence interval, as for the same value of α, the T case is wider. However, as $n \to \infty$, we have, $t_{1-\frac{\alpha}{2},n-1} \to z_{1-\frac{\alpha}{2}}$, as illustrated in Figure 5.5. Hence for non-small samples, the confidence interval (6.5) is very close to the confidence interval (6.4) with s replacing σ. Note that the T-confidence interval hinges on the normality assumption of the data. In fact for small samples, cases that deviate from normality imply imprecision of the confidence intervals. However for larger samples, these confidence intervals serve as a good approximation. Still in these larger sample cases, one might as well use $z_{1-\frac{\alpha}{2}}$ instead of $t_{1-\frac{\alpha}{2},n-1}$.

The code in Listing 6.2 calculates the confidence interval (6.5), where it is assumed that the population variance is unknown.

Listing 6.2: CI for single sample population with variance assumed unknown

```
1  using CSV, Distributions, HypothesisTests
2
3  data = CSV.read("../data/machine1.csv", header=false)[:,1]
4  xBar, n = mean(data), length(data)
5  s = std(data)
6  alpha = 0.1
7  t = quantile(TDist(n-1),1-alpha/2)
8
9  println("Calculating formula: ", (xBar - t*s/sqrt(n), xBar + t*s/sqrt(n)))
10 println("Using confint() function: ", confint(OneSampleTTest(xBar,s,n),alpha))
```

```
Calculating formula:      (52.49989385779555, 53.412518384764276)
Using confint() function: (52.49989385779555, 53.412518384764276)
```

This example is very similar to Listing 6.1, however there are several differences. In line 5, since the population variance is assumed unknown, the population standard deviation `sig` of Listing 6.1 is replaced with the sample standard deviation `s`. In line 7 the quantile `t` is calculated on a T-distribution, `TDist(n-1)`, with $n-1$ degrees of freedom. Previously, the quantile `z` was calculated on a standard normal distribution `Normal()`. Lines 9 and 10 are very similar to those in the previous listing, but `z` and `sig` are replaced with `t` and `s` respectively. It can be seen that the outputs of lines 9 and 10 are in agreement, and that the confidence interval is wider than that calculated in the previous Listing 6.1.

6.2 Two Sample Confidence Intervals for the Difference in Means

We now consider cases in which there are two populations involved. As an example, consider two separate machines, 1 and 2, which are designed to make pipes of the same diameter. In this case, due to small differences and tolerances in the manufacturing process, the distribution of pipe diameters from each machine will differ. In such cases where two populations are involved, it is often of interest to estimate the difference between the population means, $\mu_1 - \mu_2$.

In order to do this we first collect two random samples, $x_{11}, \ldots, x_{n_1 1}$ and $x_{12}, \ldots, x_{n_2 2}$. For each sample $i = 1, 2$ we have the sample mean $\overline{x}_i$, and sample standard deviation s_i. In addition, the difference in sample means, $\overline{x}_1 - \overline{x}_2$ serves as a point estimate for the difference in population means, $\mu_1 - \mu_2$.

A confidence interval for $\mu_1 - \mu_2$ around the point estimate $\overline{x}_1 - \overline{x}_2$ is then constructed via the same process seen previously,

$$\overline{x}_1 - \overline{x}_2 \pm K_\alpha s_{\text{err}}. \tag{6.6}$$

We now elaborate on the values of K_α and s_{err} based on model assumptions.

Population Variances Known

In the (unrealistic) case that the population variances are known, we may explicitly compute,

$$\text{Var}(\overline{X}_1 - \overline{X}_2) = \frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}.$$

Hence the standard error is given by,

$$s_{\text{err}} = \sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}. \tag{6.7}$$

When combined with the assumption that the data is normally distributed, we can derive the following confidence interval,

$$\overline{x}_1 - \overline{x}_2 \pm z_{1-\frac{\alpha}{2}} \sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}. \tag{6.8}$$

While this case is not often applicable in practice, it is useful to cover for pedagogical reasons. Due to the fact that the population variances are almost always unknown, the `HypothesisTests` package in Julia does not have a function for this case. However, for completeness, we evaluate equation (6.8) manually in Listing 6.3.

Listing 6.3: CI for difference in population means with variances known

```
1  using CSV, Distributions, HypothesisTests
2
3  data1 = CSV.read("../data/machine1.csv", header=false)[:,1]
4  data2 = CSV.read("../data/machine2.csv", header=false)[:,1]
5  xBar1, xBar2 = mean(data1), mean(data2)
6  n1, n2 = length(data1), length(data2)
7  sig1, sig2 = 1.2, 1.6
8  alpha = 0.05
9  z = quantile(Normal(),1-alpha/2)
10
11 println("Calculating formula: ", (xBar1 - xBar2 - z*sqrt(sig1^2/n1+sig2^2/n2),
12                                   xBar1 - xBar2 + z*sqrt(sig1^2/n1+sig2^2/n2)))
```

```
Calculating formula:      (1.1016568035908845, 2.9159620096069574)
```

This listing is similar to those previously covered in this chapter. The sample means and number of observations are calculated in lines 5-6. In line 7, we stipulate the standard deviations of both populations 1 and 2, as 1.2 and 1.6 respectively (since in this scenario the population variances are assumed known). In lines 11-12 the confidence interval (6.8) is evaluated manually and printed as output.

Population Variances Unknown and Assumed Equal

Typically, when considering cases consisting of two populations, the population variances are unknown. In such cases, a common and practical assumption is that the variances are equal, denoted

by σ^2. Based on this assumption, it is sensible to use both sample variances to estimate σ^2. This estimated variance using both samples is known as the *pooled sample variance*, and is given by,

$$S_p^2 = \frac{(n_1-1)S_1^2 + (n_2-1)S_2^2}{n_1+n_2-2}.$$

Upon closer inspection, it can be observed that the above is in fact a weighted average of the sample variances of the individual samples.

In this case, it can be shown that,

$$T = \frac{\overline{X}_1 - \overline{X}_2 - (\mu_1 - \mu_2)}{S_{\text{err}}} \tag{6.9}$$

is distributed according to a T-distribution with n_1+n_2-2 degrees of freedom, where the standard error is,

$$S_{\text{err}} = S_p\sqrt{\frac{1}{n_1}+\frac{1}{n_2}}.$$

Hence we arrive at the following confidence interval,

$$\overline{x}_1 - \overline{x}_2 \pm t_{1-\frac{\alpha}{2},n-2}\; s_p\; \sqrt{\frac{1}{n_1}+\frac{1}{n_2}}, \tag{6.10}$$

where s_p is the square root of the observed pooled sample variance and $t_{1-\frac{\alpha}{2},n-2}$ is a quantile of a T-distribution with $n-2$ degrees of freedom.

The code in Listing 6.4 calculates the confidence interval (6.10), where it is assumed that the population variance is unknown. This is compared with the result from the `HypothesisTests` package using `EqualVarianceTTest()`. It can be observed that the results are in agreement.

Listing 6.4: CI for difference in means, variance unknown, assumed equal

```
1  using CSV, Distributions, HypothesisTests
2
3  data1 = CSV.read("../data/machine1.csv", header=false)[:,1]
4  data2 = CSV.read("../data/machine2.csv", header=false)[:,1]
5  xBar1, xBar2 = mean(data1), mean(data2)
6  n1, n2 = length(data1), length(data2)
7  alpha = 0.05
8  t = quantile(TDist(n1+n2-2),1-alpha/2)
9
10 s1, s2 = std(data1), std(data2)
11 sP = sqrt(((n1-1)*s1^2 + (n2-1)*s2^2) / (n1+n2-2))
12
13 println("Calculating formula: ", (xBar1 - xBar2 - t*sP* sqrt(1/n1 + 1/n2),
14                                    xBar1 - xBar2 + t*sP* sqrt(1/n1 + 1/n2)))
15 println("Using confint(): ", confint(EqualVarianceTTest(data1,data2),alpha))
```

```
Calculating formula:      (1.1127539574575822, 2.90486485574026)
Using confint() function: (1.1127539574575822, 2.90486485574026)
```

In line 8, a T-distribution with `n1+n2-2` degrees of freedom is used. In line 10 the sample standard deviations are calculated. In line 11, the pooled sample variance `sP` is calculated. In lines 13-14, (6.10) is evaluated manually, while in line 15 the `confint()` function is used.

Population Variances Unknown and not Assumed Equal

In certain cases, it may be appropriate to relax the assumption of equal population variances. In this case, the estimate for S_{err} is given by,

$$S_{\text{err}} = \sqrt{\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}}.$$

This is due to the fact that the variance of the difference of two independent sample means is the sum of the variances of each of the means. Hence in this case the statistic (6.9) is adapted to,

$$T = \frac{\overline{X}_1 - \overline{X}_2 - (\mu_1 - \mu_2)}{\sqrt{\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}}}. \tag{6.11}$$

It turns out that (6.11) is only T-distributed if the variances are equal, otherwise it isn't. Nevertheless, an approximate confidence interval is commonly used by approximating the distribution of (6.11) with a T-distribution. This is called the *Satterthwaite approximation.*

The approximation suggests a T-distribution with a parameter (degrees of freedom) given via,

$$v = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{\left(s_1^2/n_1\right)^2}{n_1 - 1} + \frac{\left(s_2^2/n_2\right)^2}{n_2 - 1}}. \tag{6.12}$$

Now it holds that,

$$T \underset{\text{approx}}{\sim} t(v). \tag{6.13}$$

That is, the random variable T from (6.11) is approximately distributed according to a T-distribution with v degrees of freedom. Note that v does not need to be an integer. We investigate this approximation further in Listing 6.6 later.

Using the Satterthwaite approximation, following steps similar to previous confidence intervals, and given (6.13), we arrive at the following confidence interval formula,

$$\overline{x}_1 - \overline{x}_2 \pm t_{1-\frac{\alpha}{2},v} \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}. \tag{6.14}$$

In Listing 6.5, we calculate the confidence interval (6.14), where it is assumed that the population variances are unknown and not assumed equal. We then compare the result to those resulting from the use of `UnequalVarianceTTest()` from the `HypothesisTests` package. It can be observed that the results are in agreement.

Listing 6.5: CI for difference in means, variance unknown and unequal

```
1  using CSV, Distributions, HypothesisTests
2
3  data1 = CSV.read("../data/machine1.csv", header=false)[:,1]
4  data2 = CSV.read("../data/machine2.csv", header=false)[:,1]
5  xBar1, xBar2 = mean(data1), mean(data2)
6  s1, s2 = std(data1), std(data2)
7  n1, n2 = length(data1), length(data2)
8  alpha = 0.05
9
10 v = (s1^2/n1 + s2^2/n2)^2 / ( (s1^2/n1)^2 / (n1-1) + (s2^2/n2)^2 / (n2-1) )
11
12 t = quantile(TDist(v),1-alpha/2)
13
14 println("Calculating formula: ", (xBar1 - xBar2 - t*sqrt(s1^2/n1 + s2^2/n2),
15                                    xBar1 - xBar2 + t*sqrt(s1^2/n1 + s2^2/n2)))
16 println("Using confint(): ",   confint(UnequalVarianceTTest(data1,data2),alpha))
```

```
Calculating formula: (1.0960161148824918, 2.9216026983153505)
Using confint():     (1.0960161148824918, 2.9216026983153505)
```

The main difference in this code block from the previous code block is the calculation of the degrees of freedom, `v`, which is performed in line 10. In line 12 `v` is then used to derive the T-statistic `t`. In lines 14-15, equation (6.14) is evaluated manually, while in line 16 the `confint()` function is used.

Exploring the Satterthwaite Approximation

We now investigate the approximate distribution stated in (6.13). Observe that both sides of the "distributed as" ($\sim$) symbol are random variables which depend on the same random experiment. Hence the statement can be presented generally, as a case of the following format,

$$X(\omega) \sim F_{h(\omega)}, \tag{6.15}$$

where ω is a point in the sample space (see Chapter 2). Here $X(\omega)$ is a random variable, and F is a distribution that depends on a parameter h, which itself depends on ω. In our case of the Satterthwaite approximation, h is given by (6.12). That is, h can be thought of as v, which itself depends on the specific observations made for our two sample groups (a function of s_1 and s_2).

Now by recalling the inverse probability transform from Section 3.4, we have that (6.15) is equivalent to,

$$F^{-1}_{h(\omega)}\big(X(\omega)\big) \;\sim\; \text{uniform}(0,1). \tag{6.16}$$

Hence in the case of the Satterthwaite approximation, we expect that (6.16) hold approximately. This distributional relationship would not hold with the naive alternative of treating h as simply dependent on the number of observations made (n_1 and n_2). Hence in this naive case, the distribution is not expected to be uniform.

We investigate this in Listing 6.6, where we construct Figure 7.4, a Q-Q plot comparing T-values calculated from the Satterthwaite approximation (6.13) and T-values calculated via the naive equal variance case. See Section 4.4 for a description of Q-Q plots.

The results in Figure 7.4, indicate that the Satterthwaite approximation is a better approximation than simply using the degrees of freedom. It can be observed that the data from the fixed v case deviates further from the 1:1 slope in comparison to the case where v was calculated based on each experiments sample observations, i.e. calculated from equation (6.12). Hence the distribution of T-statistics from Satterthwaite calculated v's yields better results than the constant v case.

Listing 6.6: Analyzing the Satterthwaite approximation

```
using Distributions, Statistics, Plots, Random; pyplot()
Random.seed!(0)

mu1, sig1, n1 = 0, 2, 8
mu2, sig2, n2 = 0, 30, 15
dist1 = Normal(mu1, sig1)
dist2 = Normal(mu2, sig2)

N = 10^6
tdArray = Array{Tuple{Float64,Float64}}(undef,N)

df(s1,s2,n1,n2) =
    (s1^2/n1 + s2^2/n2)^2 / ( (s1^2/n1)^2/(n1-1) + (s2^2/n2)^2/(n2-1) )

for i in 1:N
    x1Data = rand(dist1, n1)
    x2Data = rand(dist2, n2)
    x1Bar,x2Bar = mean(x1Data),mean(x2Data)
    s1,s2 = std(x1Data),std(x2Data)
    tStat = (x1Bar - x2Bar) / sqrt(s1^2/n1 + s2^2/n2)
    tdArray[i] = (tStat , df(s1,s2,n1,n2))
end
sort!(tdArray, by = first)

invVal(v,i) = quantile(TDist(v),i/(N+1))

xCoords  = Array{Float64}(undef,N)
yCoords1 = Array{Float64}(undef,N)
yCoords2 = Array{Float64}(undef,N)

for i in 1:N
    xCoords[i] = first(tdArray[i])
    yCoords1[i] = invVal(last(tdArray[i]), i)
    yCoords2[i] = invVal(n1+n2-2, i)
end

scatter(xCoords, yCoords1, c=:blue, label="Calculated v", msw=0)
scatter!(xCoords, yCoords2, c=:red, label="Fixed v", msw=0)
plot!([-10,10], [-10,10],
         c=:black, lw=0.3, xlims=(-8,8), ylims=(-8,8), ratio=:equal, label="",
         xlabel="Theoretical t-distribution quantiles",
         ylabel="Simulated t-distribution quantiles", legend=:topleft)
```

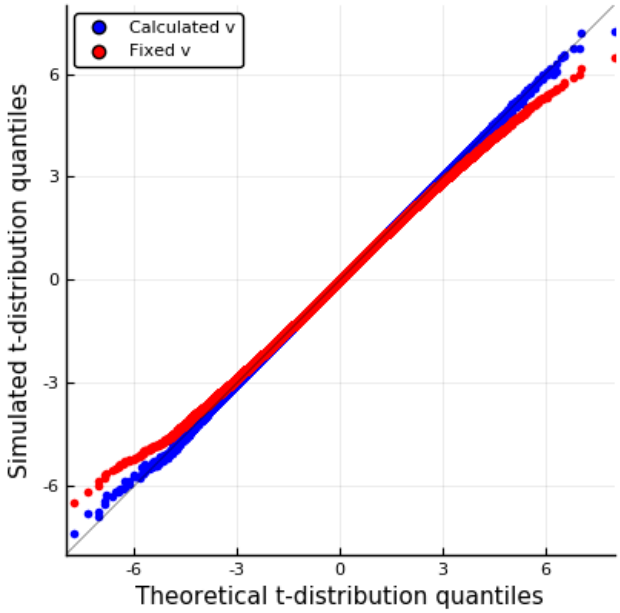

Figure 6.1: Q-Q plots of T-statistics from identical experiments, given Satterthwaite calculated v's, along with T-statistics given constant v.

In lines 4-5 we set the means and standard deviations of the two underlying distributions, and the number of observations that will be made for each group. In line 8 we set the number of times we repeat the experiment, `N`. In line 9 we pre-allocate the array `tdArray`, in which each element is a tuple. The first element of each tuple will be the T-statistic calculated via (6.11), while the second will be the corresponding degrees of freedom calculated via (6.12). In lines 12-13 we define the function `df()`, which implements (6.12). In lines 15-22, we conduct `N` experiments, where for each we calculate the T-statistic, and the degrees of freedom. In line 23 `sort!()` is used to re-order `tdArrray` in ascending order according to the T-statistics via `by = first`. This is done so that we can construct the Q-Q plot. In line 25 the function `invVal()` is defined, which uses the `quantile()` function to perform the inverse probability transform on the degrees of freedom associated with each T-statistic for each experiment. Note that the number of quantiles is one more than the number of experiments, i.e. `N+1`. In lines 31-35 the quantiles of our data are calculated. Here `xCoords` represents the T-statistic quantiles, and `yCoords1` represents the quantiles of a T-distribution with v degrees of freedom, where v is calculated via (6.12). The array `yCoords2` on the other hand represents the quantiles of a T-distribution with v degrees of freedom, where $v = n_1 + n_2 - 2$. Lines 37-42 plot the Q-Q plots creating Figure 6.6.

6.3 Confidence Intervals for Proportions

In certain inference settings the parameter of interest is a *population proportion*. Examples include the proportion of females within an animal population, the proportion of customers that own two or more cars, or the proportion of baby turtles that survive the first day after hatching. In all such cases, one may view the proportion as either a characteristic of the population or alternatively as the probability of some event happening when randomly sampling from the population.

When carrying out inference for a proportion we assume that there exists some unknown population proportion $p \in (0,1)$. We then sample an i.i.d. sample of observations $I_1, \ldots, I_n$, where for the i'th observation, $I_i = 0$ if the event in question does not happen, and $I_i = 1$ if the event occurs. For example, dealing with the proportion of females, we set $I_i = 0$ if the i'th sample is not a female and $I_i = 1$ if the i'th sample is a female.

A natural estimator for the proportion is then the sample mean of $I_1, \ldots, I_n$ which we denote,

$$\hat{p} = \frac{\sum_{i=1}^{n} I_i}{n}. \tag{6.17}$$

In this case since the summands in the numerator are indicator variables, the sum is simply a count of the number of observations for which the event occurred. Hence we also call $\hat{p}$, the *proportion estimator*.

Now observe that each I_i is a Bernoulli random variable with success probability p. Under the i.i.d. assumption this means that the numerator of (6.17) is binomially distributed with parameters n and p (see Section 3.5 to review the binomial distribution). Hence,

$$\mathbb{E}\left[\sum_{i=1}^{n} I_i\right] = np, \qquad \text{and} \qquad \mathrm{Var}\left(\sum_{i=1}^{n} I_i\right) = np(1-p).$$

By combining (6.17) with the above, we have that,

$$\mathbb{E}[\hat{p}] = p, \qquad \text{and} \qquad \mathrm{Var}(\hat{p}) = \frac{p(1-p)}{n}. \tag{6.18}$$

Hence $\hat{p}$ is an unbiased and consistent estimator of p. That is, on average $\hat{p}$ estimates p perfectly and if more observations are collected the variance of the estimator vanishes and $\hat{p} \to p$. Furthermore, we can use the central limit theorem to create a normal approximation for the distribution of $\hat{p}$ and yield an approximate confidence interval. To do so denote,

$$\tilde{Z}_n = \frac{\hat{p} - p}{\sqrt{p(1-p)/n}}.$$

This is a random variable that approximately follows a standard normal distribution. The approximation becomes exact as n grows. The same also holds for a slightly different random variable,

$$\hat{Z}_n = \frac{\hat{p} - p}{\sqrt{\hat{p}(1-\hat{p})/n}}. \tag{6.19}$$

This is because $\hat{p}$ is an unbiased and consistent estimator of p and thus replacing the p's in the denominator of $\tilde{Z}_n$ with $\hat{p}$ to yield $\hat{Z}_n$ does not significantly affect the distribution for large n. We now use the approximate normality of $\hat{Z}_n$ to create a confidence interval for p. First observe that as a consequence of the approximate standard normal distribution,

$$\mathbb{P}(z_{\alpha/2} \leq \hat{Z}_n \leq z_{1-\alpha/2}) \approx 1 - \alpha. \tag{6.20}$$

We now use (6.19) in (6.20), along with the fact that $z_{\alpha/2} = -z_{1-\alpha/2}$ as follows,

$$\begin{aligned}
1-\alpha &\approx \mathbb{P}(-z_{1-\alpha/2} \leq \hat{Z}_n \leq z_{1-\alpha/2}) \\
&= \mathbb{P}(-z_{1-\alpha/2} \leq \frac{\hat{p}-p}{\sqrt{\hat{p}(1-\hat{p})/n}} \leq z_{1-\alpha/2}) \\
&= \mathbb{P}(-z_{1-\alpha/2}\sqrt{\hat{p}(1-\hat{p})/n} \ \leq \ \hat{p}-p \ \leq \ z_{1-\alpha/2}\sqrt{\hat{p}(1-\hat{p})/n}) \\
&= \mathbb{P}(-\hat{p}-z_{1-\alpha/2}\sqrt{\hat{p}(1-\hat{p})/n} \ \leq \ -p \ \leq \ -\hat{p}+z_{1-\alpha/2}\sqrt{\hat{p}(1-\hat{p})/n}) \\
&= \mathbb{P}(\hat{p}-z_{1-\alpha/2}\sqrt{\hat{p}(1-\hat{p})/n} \ \leq \ p \ \leq \ \hat{p}+z_{1-\alpha/2}\sqrt{\hat{p}(1-\hat{p})/n}).
\end{aligned}$$

We thus arrive at the following (approximate) *confidence interval for proportions* formula,

$$\hat{p} \pm z_{1-\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}. \tag{6.21}$$

Observe that this confidence interval formula agrees with the general form of (6.1), where the standard error depends only on the statistic $\hat{p}$ and is represented as,

$$s_{\text{err}} = \sqrt{\frac{\hat{p}(1-\hat{p})}{n}}. \tag{6.22}$$

Similar more complex confidence interval formulas also exist for the case of two populations. Say one is interested in comparing the proportion of females in two different sub-species populations of crocodiles. By sampling n_1 crocodiles from one sub-species and n_2 crocodiles for the other subspecies, one can form the point estimators $\hat{p}_1$ and $\hat{p}_2$, each in the same manner as (6.17). The point estimator for the difference in proportions is then simply $\hat{p}_1 - \hat{p}_2$. An approximate $1-\alpha$ confidence interval for this parameter is,

$$\hat{p}_1 - \hat{p}_2 \pm z_{1-\alpha/2}\sqrt{\frac{\hat{p}_1(1-\hat{p}_1)}{n_1} + \frac{\hat{p}_2(1-\hat{p}_2)}{n_2}}. \tag{6.23}$$

Compare this formula with the general form (6.6) and other formulas for two populations presented in the previous section. You can see that (6.23) follows a similar stucture. We now return to examples and discussions of (6.21) and don't discuss (6.23) further.

In Listing 6.7 we demonstrate basic usage of the confidence interval for proportions formula (6.21). We consider the `Grade` column of `purchaseData.csv`. As the code demonstrates, here the possible grades are '`A`'—'`E`'. We obtain a point estimate and a confidence interval for the proportion of observations with level '`E`'. You may modify line 11 of the code to carry out inference for other levels. Note that this code also deals with missing observations by culling the missing observations and only uses observations for which `Grade` is not `missing`.

Listing 6.7: Confidence interval for a proportion

```
1  using CSV, DataFrames, CategoricalArrays
2
3  data = CSV.read("../data/purchaseData.csv", copycols = true)
4  println("Levels of Grade: ", levels(data.Grade))
5  println("Data points: ", nrow(data))
6
7  n = sum(.!(ismissing.(data.Grade)))
8  println("Non-missing data points: ", n)
9  data2 = dropmissing(data[:,[:Grade]],:Grade)
10
11 gradeInQuestion = "E"
12 indicatorVector = data2.Grade .== gradeInQuestion
13 numSuccess = sum(indicatorVector)
14 phat = numSuccess/n
15 serr = sqrt(phat*(1-phat)/n)
16
17 alpha = 0.05
18 confidencePercent = 100*(1-alpha)
19 zVal = quantile(Normal(),1-alpha/2)
20 confInt = (phat - zVal*serr, phat + zVal*serr)
21
22 println("\nOut of $n non-missing observations, "*
23         "$numSuccess are at level $gradeInQuestion.")
24 println("Hence a point estimate for the proportion "*
25         "of grades at level $gradeInQuestion is $phat.")
26 println("A $confidencePercent% confidence interval for "*
27         "the proprotion of level $gradeInQuestion is:\n$confInt.")
```

```
Levels of Grade: ["A", "B", "C", "D", "E"]
Data points: 200
Non-missing data points: 187

Out of 187 non-missing observations, 61 are at level E.
Hence a point estimate for the proportion of grades at level E is 0.3262.
A 95.0% confidence interval for the proprotion of level E is:
(0.2590083767381328, 0.3933980403741667).
```

Lines 3-5 load and describe the data, focusing on the `Grade` column. Note the use of `levels()` from `CategoricalArrays`. Lines 7-9 handle missing values. Note the use of '`.!()`' to broadcast negation on the output of a broadcasted `ismissing()`. Summing this yields the number of (non-missing) observations `n`. The new data frame, `data2` is comprised of a single variable `Grade` after `dropmissing()` is applied with `:Grade` as a second argument. In line 11 we choose to carry out proportion estimation for `"E"`. Line 12 creates $I_1, \ldots, I_n$. Line 14 calculates $\hat{p}$ and line 15 calculates s_{err} as in (6.22). Lines 17-20 determine the confidence interval (6.21), with `confInt` represented as a `Tuple`. The remainder of the code prints the output describing the results.

Sample Size Planning

Denote the confidence interval (6.21) as $\hat{p} \pm E$ where E is the *margin of error* or half the *width of the confidence interval*, denoted by,

$$E = z_{1-\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}.$$

You may often want to plan an experiment, or a sampling scheme such that E is not too wide. For example, 'not more than 0.1'. For this, you need to choose a sample size n prior to sampling. We now illustrate a crude yet effective way for such *sample size planning*.

First observe that for typical values of α we have that $z_{1-\alpha/2} \approx 2$. In fact, when $\alpha = 0.0455$ we have that $z_{1-\alpha/2} = 2$ almost exactly. Values ranging between 1.5 and 2.5 are common for most chosen confidence levels in practice. Hence in general, crudely taking $z_{1-\alpha/2}$ as '2' helps simplify expressions.

Say we want $E \leq \varepsilon$, e.g. with the maximal margin of error $\varepsilon = 0.1$, or any other similar value. Then taking $z_{1-\alpha/2} = 2$ for simplicity we get,

$$2\sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \leq \varepsilon, \qquad \text{or} \qquad 4\frac{\hat{p}(1-\hat{p})}{\varepsilon^2} \leq n. \tag{6.24}$$

Now also observe that $x(1-x)$ is maximized at $x = 1/2$ with a maximal value of $1/4$. Hence,

$$4\frac{\hat{p}(1-\hat{p})}{\varepsilon^2} \leq \frac{1}{\varepsilon^2}.$$

This means that by taking $n \geq \varepsilon^{-2}$ we ensure (6.24). For seeking a whole number of observations we use the $\lceil \cdot \rceil$ 'ceiling' (rounding up) operator, to get the *proportions sample size formula*,

$$n^* = \left\lceil \frac{1}{\varepsilon^2} \right\rceil. \tag{6.25}$$

In Listing 6.8 we create a simple table implementing (6.25) to get a sense for the magnitude of samples needed. As you can see for $\varepsilon = 0.1$ we need 100 observations. However if we seek a more accurate confidence interval with $\varepsilon = 0.01$ then $10,000$ observations are needed! Again, keep in mind that these calculations are assuming $\alpha = 0.0455$.

Listing 6.8: Sample size planning for proportions

```
1  for eps in [0.1, 0.05, 0.02, 0.01]
2      n = ceil(1/eps^2)
3      println("For epsilon = $eps set n = $n")
4  end
```

```
For epsilon = 0.1 set n = 100.0
For epsilon = 0.05 set n = 400.0
For epsilon = 0.02 set n = 2500.0
For epsilon = 0.01 set n = 10000.0
```

The listing is a straightforward implementation of (6.25). Observe the use of `ceil()` in line 3.

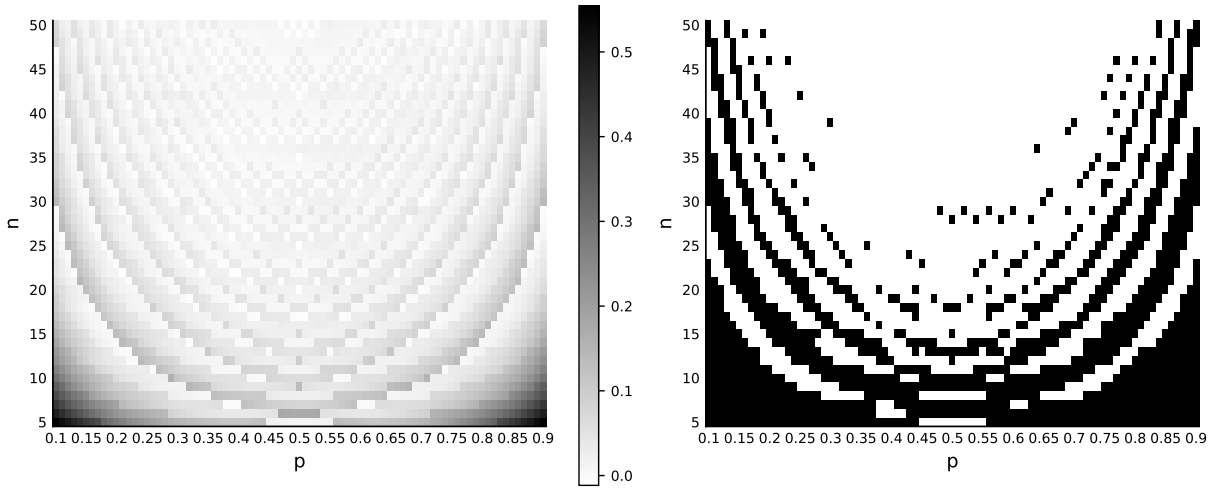

Figure 6.2: Approximation error of a confidence interval for proportion. The heatmaps are for various n and p combinations for $\alpha = 0.05$.

Validity of the Approximation

The key to the derivation of (6.21) is the distributional approximation in (6.20). In many cases this approximation works well, however for small sample sizes n or values of p near 0 or 1, this is often too crude of an approximation. A consequence is that one may obtain a confidence interval that isn't actually a $1-\alpha$ confidence interval, but rather has a different coverage probability.

One common rule of thumb used to decide if (6.21) is valid is to require that both the product np and the product $n(1-p)$ be at least 10. For $p = 0.5$ this rule specifies a minimal sample size of $n = 20$, and for other values of p higher values of n are required.

How does such a rule come about? To explore this we now present a computational experiment, aiming to asses when (6.21) is valid. For this we explore a grid of n ranging from 5 to 50 and p in the interval $[0.1, 0.9]$. For each combination we repeat $N = 5,000$ Monte Carlo experiments and calculate the following:

$$(1-\alpha) \quad - \quad \frac{1}{N}\sum_{k=1}^{N} \mathbf{1}\left\{ p \in \left[\hat{p} - z_{1-\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}, \quad \hat{p} + z_{1-\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \right] \right\}. \tag{6.26}$$

This estimated difference of the actual coverage probability of the confidence interval (6.21) and the desired confidence level $1-\alpha$ is a measure of the accuracy of the confidence level. We expect this difference to be almost 0 if the approximation is 'good'. Otherwise, a higher absolute difference is observed.

Listing 6.9 creates Figure 6.2 which presents the results of this simulation experiment for $\alpha = 0.05$. The left plot illustrates the estimated difference between the actual coverage probability and $1-\alpha$. Observe that in general for p values around 0.5 there is less error than p values closer to 0 or 1. Also, as expected when n is increased the error probabilities drop. There is also a periodic effect due to the fact that for small n, the proportion estimator $\hat{p}$ only falls on a small finite set of values.

The right hand plot of Figure 6.2 compares the absolute value of (6.26) to 0.04 (an ad-hoc tolerance that we selected). For (n, p) where the absolute error is less than 0.04 we can say the confidence error is 'small' and conclude that using the confidence interval formula is satisfactory. As seen from the plot, this occurs when p is closer to 0.5 and n is at around 20 or more. This may give some insight to the heuristic rule described above and generally agrees with it.

Listing 6.9: Coverage accuracy of a confidence interval for proportions

```
1  using Random, Plots, Distributions, Measures; pyplot()
2
3  N = 5*10^3
4  alpha = 0.05
5  confLevel = 1 - alpha
6  z = quantile(Normal(),1-alpha/2)
7
8  function randCI(n,p)
9      sample = rand(n) .< p
10     pHat = sum(sample)/n
11     serr = sqrt(pHat*(1-pHat)/n)
12     (pHat - z*serr, pHat + z*serr)
13 end
14 cover(p,ci) = ci[1] <= p && p <= ci[2]
15
16 pGrid = 0.1:0.01:0.9
17 nGrid = 5:1:50
18 errs = zeros(length(nGrid),length(pGrid))
19
20 for i in 1:length(nGrid)
21     for j in 1:length(pGrid)
22         Random.seed!(0)
23         n, p = nGrid[i], pGrid[j]
24         coverageRatio = sum([cover(p,randCI(n,p)) for _ in 1:N])/N
25         errs[i,j] = confLevel - coverageRatio
26     end
27 end
28
29 default(xlabel = "p", ylabel = "n",
30     xticks =([1:5:length(pGrid);], minimum(pGrid):0.05:maximum(pGrid)),
31     yticks =([1:5:length(nGrid);], minimum(nGrid):5:maximum(nGrid)))
32
33 p1 = heatmap(errs, c=cgrad([:white, :black]))
34 p2 = heatmap(abs.(errs) .<= 0.04, legend = false, c=cgrad([:black, :white]))
35 plot(p1,p2, size = (1000,400), margin = 5mm)
```

In line 3 we set the number of Monte Carlo repetitions, `N`. Lines 4-6 define constants for the confidence interval based on `alpha` from line 4. In lines 8-13 we define the function `randCI()` for generating a random sample and an associated confidence interval. The function `cover()` that we define in line 14 checks if `p` is covered by the given confidence interval, `ci`. Lines 16-17 define the grid of p values and n values on which we estimate (6.26). In line 18 we initialize the matrix `errs` using `zeros()`. The simulation repetitions are in lines 20-27 where, after resetting the seed in line 22, for each (n, p) combination we compute the sum in (6.26) into `coverageRatio` in line 24 by composing `cover()` on `randCI()`. Then in line 25 we record the estimated difference in the matrix `errs`. Lines 29-35 create Figure 6.2.

6.4 Confidence Interval for the Variance of Normal Population

We now consider confidence intervals when the parameter in question is the variance. We also use this as an example to show how model assumptions may strongly affect the accuracy of the confidence interval. Consider sampling from a population that follows a normal distribution. A point estimator for the population variance is the sample variance,

$$S^2 = \frac{1}{(n-1)} \sum_{i=1}^{n} (X_i - \overline{X})^2.$$

As illustrated in Section 5.2, when multiplied by the constant $(n-1)/\sigma^2$, the sample variance follows a chi-squared distribution with $n-1$ degrees of freedom,

$$\frac{(n-1)S^2}{\sigma^2} \sim \chi^2_{n-1}.$$

Therefore, denoting the γ-quantile of this distribution via $\chi^2_{\gamma,n-1}$, we have,

$$\mathbb{P}\left(\chi^2_{\frac{\alpha}{2},n-1} < \frac{(n-1)S^2}{\sigma^2} < \chi^2_{1-\frac{\alpha}{2},n-1} \right) = 1-\alpha. \tag{6.27}$$

Hence we can re-arrange to obtain a two-sided $100(1-\alpha)\%$ confidence interval for the variance of a normal population where we denote the observed estimator by s^2:

$$\frac{(n-1)s^2}{\chi^2_{1-\frac{\alpha}{2},n-1}} < \sigma^2 < \frac{(n-1)s^2}{\chi^2_{\frac{\alpha}{2},n-1}}. \tag{6.28}$$

Note that (6.27) only holds when sampling from data that is normally distributed. If the data is not normally distributed, then our confidence intervals will be inaccurate. Such sensitivity to assumptions is explored later in this section. However first we demonstrate a simple example for using the confidence interval (6.28) in Listing 6.10.

Listing 6.10: Confidence interval for the variance

```
1  using CSV, Distributions, HypothesisTests
2
3  data = CSV.read("../data/machine1.csv", header=false)[:,1]
4  n, s, alpha = length(data), std(data), 0.1
5  ci = (  (n-1)*s^2/quantile(Chisq(n-1),1-alpha/2),
6          (n-1)*s^2/quantile(Chisq(n-1),alpha/2)  )
7
8  println("Point estimate for the variance: ", s^2)
9  println("Confidence interval for the variance: ", ci)
```

```
Point estimate for the variance: 1.3928282706110504
Confidence interval for the variance: (0.8779243703322502, 2.6157658366723124)
```

The code is similar to Listing 6.2 and uses the same dataset. Lines 5-6 implement (6.28) based on the sample variance standard deviation `s` and a Chi-squared distribution, `Chisq()`.

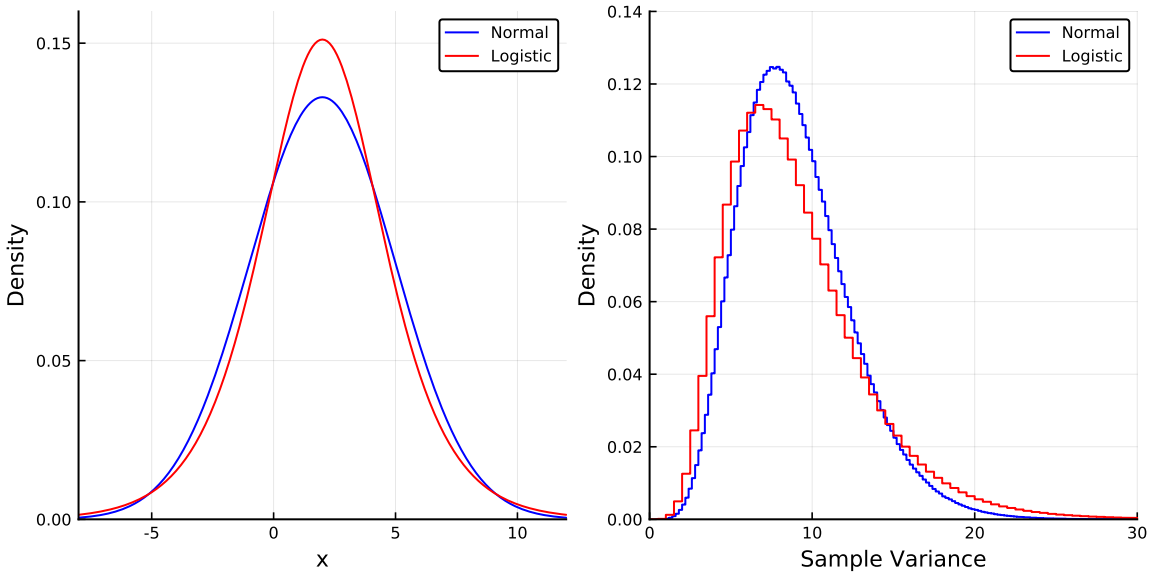

Figure 6.3: PDF's of the normal and logistic distributions, along with histograms of the sample variances from the corresponding distributions.

Sensitivity of the Normality Assumption

We now look at the sensitivity of the normality assumption on the confidence interval for the variance. As part of this example we first introduce the *logistic distribution* which has a "bell curved" shape somewhat similar to the normal distribution. It is defined by the location and scale parameters, μ and η. The PDF of the logistic distribution is,

$$f(x) = \frac{e^{-\frac{x-\mu}{\eta}}}{\eta\left(1+e^{-\frac{x-\mu}{\eta}}\right)^2}, \tag{6.29}$$

with the variance given by $\eta^2\pi^2/3$.

In Listing 6.11 we create Figure 6.3 where in the left plot, the PDF of a normal distribution with mean $\mu = 2$ and standard deviation $\sigma = 3$ is plotted against that of a logistic distribution with the same mean and variance. To achieve that we require $\eta^2\pi^2/3 = \sigma^2$ and hence,

$$\eta = \frac{\sqrt{3}}{\pi}\sigma. \tag{6.30}$$

While both of these symmetric (about the mean) distributions share the same mean and variance and hence are somewhat similar, in the right plot, we show via Monte Carlo that the distributions of their sample variances with $n = 15$ are actually significantly different. This gives a first hint at the fact that the confidence interval formula (6.28) may be very sensitive to the normality assumption. Later, in the example that follows, we investigate the effect of this on the confidence interval.

Listing 6.11: Comparison of sample variance distributions

```
1  using Distributions, Plots; pyplot()
2
3  mu, sig = 2, 3
4  eta = sqrt(3)*sig/pi
5  n, N = 15, 10^7
6  dNormal   = Normal(mu, sig)
7  dLogistic = Logistic(mu, eta)
8  xGrid = -8:0.1:12
9
10 sNormal   = [var(rand(dNormal,n)) for _ in 1:N]
11 sLogistic = [var(rand(dLogistic,n)) for _ in 1:N]
12
13 p1 = plot(xGrid, pdf.(dNormal,xGrid), c=:blue, label="Normal")
14 p1 = plot!(xGrid, pdf.(dLogistic,xGrid), c=:red, label="Logistic",
15         xlabel="x",ylabel="Density", xlims= (-8,12), ylims=(0,0.16))
16
17 p2 = stephist(sNormal, bins=200, c=:blue, normed=true, label="Normal")
18 p2 = stephist!(sLogistic, bins=200, c=:red, normed=true, label="Logistic",
19         xlabel="Sample Variance", ylabel="Density", xlims=(0,30), ylims=(0,0.14))
20
21 plot(p1, p2, size=(800, 400))
```

In line 3 we define the mean and standard deviation that will be used for both distributions. Then in line 4 we calculate `eta` according to (6.30). In line 5 the number of sample observations, `n`, and total number of experiments, `N`, are specified. In lines 6-7 we define the two distributions with matched moments and variance. Note that the Julia `Logistic()` function uses the same parametrization as that of equation (6.29). In lines 10-11 comprehensions are used to generate `N` sample variances from the normal and logistic distributions `dNormal` and `dLogistic`, with the values assigned to the arrays `sNormal` and `sLogistic` respectively. The remainder of the code creates the plots with lines 13-15 creating the left plot by using `pdf()` broadcasted over `xGrid` and lines 17-19 creating histograms of the sample variances.

Having seen that the distribution of the sample variance heavily depends on the shape of the actual distribution, we now investigate the effect that this has on the accuracy of the confidence interval. Specifically we show that while usage of the confidence interval formula (6.28) yields $1-\alpha$ coverage for normally distributed data, it strongly deviates for the logistic distribution case.

In Listing 6.12 we cycle through different values of α from 0.001 to 0.1, and for each value, we perform N of the following identical experiments: calculate the sample variance of n observations and evaluate the confidence interval (6.28). We then calculate the proportion of times that the actual (unknown) variance of the distribution is contained within the confidence interval in a similar manner to what we did in the context of proportions in (6.26). The effective α values are then plotted against the actual values in Figure 6.4.

It can be observed that in the case of the normal distribution, the simulated α values align with those of the actual α used. However, in the case of logistic distribution there is a strong discrepancy. This illustrates that model assumptions are critical for the correctness of the confidence interval (6.28). Note that in general, confidence intervals for the mean such as (6.5), would be less sensitive to model assumptions than the confidence interval for the variance.

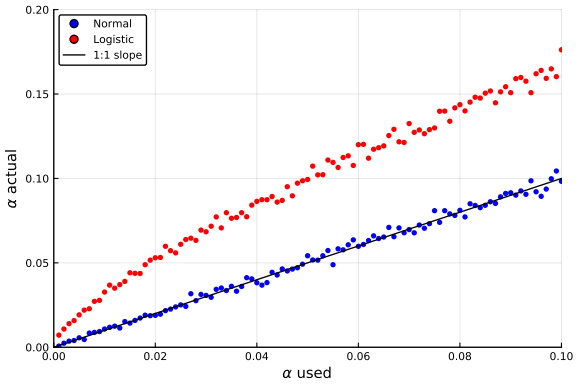

Figure 6.4: Actual α values vs. α values used in confidence intervals.

Listing 6.12: Actual α vs. α used in variance confidence intervals

```
1  using Distributions, Plots, LaTeXStrings; pyplot()
2
3  mu, sig = 2, 3
4  eta = sqrt(3)*sig/pi
5  n, N = 15, 10^4
6  dNormal   = Normal(mu, sig)
7  dLogistic = Logistic(mu, eta)
8  alphaUsed = 0.001:0.001:0.1
9
10 function alphaSimulator(dist, n, alpha)
11     popVar        = var(dist)
12     coverageCount = 0
13     for _ in 1:N
14         sVar = var(rand(dist, n))
15         L = (n - 1) * sVar / quantile(Chisq(n-1),1-alpha/2)
16         U = (n - 1) * sVar / quantile(Chisq(n-1),alpha/2)
17         coverageCount +=  L < popVar && popVar < U
18     end
19     1 - coverageCount/N
20 end
21
22 scatter(alphaUsed, alphaSimulator.(dNormal,n,alphaUsed),
23         c=:blue, msw=0, label="Normal")
24 scatter!(alphaUsed, alphaSimulator.(dLogistic, n, alphaUsed),
25         c=:red, msw=0, label="Logistic")
26 plot!([0,0.1],[0,0.1],c=:black, label="1:1 slope",
27         xlabel=L"\alpha"*" used", ylabel=L"\alpha"*" actual",
28         legend=:topleft, xlim=(0,0.1), ylims=(0,0.2))
```

Lines 3-7 are identical to the previous listing. In line 8 we define a grid of α values over which we carry out the experiment. Lines 10-20 define the function `alphaSimulator()`. This function takes a distribution, the total number of sample observations and a value of alpha as input. It then generates `N` separate confidence intervals for the variance via equation (6.28) and returns the corresponding proportion of times the confidence intervals do not contain the actual variance of the distribution. We apply `alphaSimulator()` directly in lines 22 and 24 as part of the scatter plots. Lines 26-28 plot the 1:1 line on which α used equals α actual.

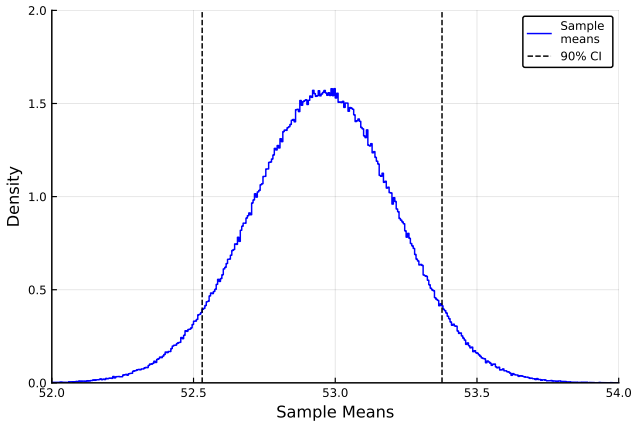

Figure 6.5: A single confidence interval for the mean generated by bootstrapped data.

6.5 Bootstrap Confidence Intervals

When developing confidence intervals, the main goal is to make some sort of inference about the population based on sample data. However, in some cases a statistical model may not be readily available, or as we saw in Listing 6.9 and Listing 6.12, model error may cause inaccuracies. Hence it is useful to have an alternative method for finding confidence intervals. One such general method is the method of *bootstrap confidence intervals*.

Bootstrap, also called *empirical bootstrap*, is a useful technique which relies on resampling from the observed data $x_1, \ldots, x_n$ in order to empirically construct the distribution of the point estimator. One way in which this resampling can be conducted is to apply the inverse probability transform on the empirical cumulative distribution function. However from an implementation perspective a simpler alternative is to consider the data points $x_1, \ldots, x_n$,and then randomly sample n discrete uniform indexes, $j_1, \ldots, j_n$ each in the range $\{1, \ldots, n\}$. The *resampled data* denoted via $x^* = (x_1^*, \ldots, x_n^*)$ is then,

$$x^* = (x_{j_1}, \ldots, x_{j_n}).$$

That is, each point in the resampled data is a random observation from the original data, where we allow to 'sample with replacement'. In Julia, if the sample is represented by an array called `sampleData`, say of length `n`, we create an instance of x^* by executing `rand(sampleData,n)`. This method of the `rand()` function will uniformly sample `n` random copies of elements from `sampleData` with replacement.

The idea of empirical bootstrap is now to repeat the resampling a large number of times, say N, and for each resampled data vector, $x^*(1), \ldots, x^*(N)$ to compute the parameter estimate. If the parameter estimate is denoted by the function $h : \mathbb{R}^n \mapsto \mathbb{R}$, then we end up with values,

$$\begin{aligned} h^*(1) &= h\big(x_1^*(1), \ldots, x_n^*(1)\big), \\ h^*(2) &= h\big(x_1^*(2), \ldots, x_n^*(2)\big), \\ &\vdots \\ h^*(N) &= h\big(x_1^*(2), \ldots, x_n^*(N)\big). \end{aligned}$$

A bootstrap confidence interval is then determined by computing the respective lower and upper $(\frac{\alpha}{2}, 1-\frac{\alpha}{2})$ quantiles of the sequence $h^*(1), \ldots, h^*(N)$. The beauty of this method is that if n is not too small, the resulting quantiles are quite close to the actual quantiles of the distribution of the point estimate. Hence in general, bootstrap confidence intervals provide a very generic and general method to obtain confidence intervals for parameters in question.

In Listing 6.13 we generate a bootstrap confidence interval for the mean, using the same data that was used in Section 6.1. The listing also creates Figure 6.5 which illustrates the empirical distribution of $h^*(1), \ldots, h^*(N)$ where $h(\cdot)$ is the sample mean. The 90% confidence interval is $(52.53, 53.38)$. Compare this with the output of Listing 6.2 where the 90% confidence interval using the T-distribution and formula (6.5) for the population mean was $(52.5, 53.41)$. Clearly, the results are similar. While bootstrap requires more computational effort and doesn't come with a neat simple formula as (6.5), it is useful because we can use it to generate confidence intervals for other point estimators. For example, as observed in the output, we also generate a 90% confidence interval for the median.

Listing 6.13: Bootstrap confidence interval

```
1  using Random, CSV, Distributions, Plots; pyplot()
2  Random.seed!(0)
3
4  sampleData = CSV.read("../data/machine1.csv", header=false)[:,1]
5  n, N = length(sampleData), 10^6
6  alpha = 0.1
7
8  bootstrapSampleMeans = [mean(rand(sampleData, n)) for i in 1:N]
9  Lmean = quantile(bootstrapSampleMeans, alpha/2)
10 Umean = quantile(bootstrapSampleMeans, 1-alpha/2)
11
12 bootstrapSampleMedians = [median(rand(sampleData, n)) for i in 1:N]
13 Lmed = quantile(bootstrapSampleMedians, alpha/2)
14 Umed = quantile(bootstrapSampleMedians, 1-alpha/2)
15
16 println("Bootstrap confidence interval for the mean: ", (Lmean, Umean) )
17 println("Bootstrap confidence interval for the median: ", (Lmed, Umed) )
18
19 stephist(bootstrapSampleMeans, bins=1000, c=:blue,
20     normed=true, label="Sample \nmeans")
21 plot!([Lmean, Lmean],[0,2], c=:black, ls=:dash, label="90% CI")
22 plot!([Umean, Umean],[0,2],c=:black, ls=:dash, label="",
23     xlims=(52,54), ylims=(0,2), xlabel="Sample Means", ylabel="Density")
```

```
Bootstrap confidence interval for the mean: (52.530497748, 53.376643266)
Bootstrap confidence interval for the median: (52.373195891, 53.49007500)
```

In line 4 we load our sample observations, and store them in the array `sampleData`. In line 5, the total number of sample observations is assigned to `n` and the number of repetitions of the bootstrap, `N`, is specified. In line 6 we specify the level of our confidence interval `alpha`. In line 8 we generate `N` bootstrapped sample means which are assigned to the array `bootstrapSampleMeans`. In lines 9-10 the lower and upper quantiles of our bootstrapped sample data is calculated, and stored as the variables `Lmean` and `Umean` respectively. Then lines 12-14 repeat the process for the sample median using `median()`. The resulting confidence intervals are printed in lines 16-17. Lines 19-23 plot a histogram of bootstrapped sample means with an illustration of the resulting confidence interval.

One may ask, how accurate are bootstrap confidence intervals? We now carry out a computational experiment and see that if the number of sample observations is not very large, then the *coverage probability* of bootstrapped confidence interval is only approximately $1-\alpha$, but not exactly. However as the sample size n grows the coverage probability converges to the desired $1-\alpha$.

In Listing 6.14 we create a series of confidence intervals based on different numbers of sample observations from an exponential distribution with $\lambda = 0.1$. Our confidence intervals are for the median which theoretically equals $\log(2)/\lambda = 6.931$. By increasing the sample size n and repeating many sampling scenarios, $M = 10^3$, each with a bootstrap computation using $N = 10^4$, we estimate the coverage probability. We see that as n increases the coverage probability approaches $1-\alpha$.

Listing 6.14: Coverage probability for bootstrap confidence intervals

```
1  using Random, Distributions
2  Random.seed!(0)
3
4  lambda = 0.1
5  dist = Exponential(1/lambda)
6  actualMedian = median(dist)
7
8  M = 10^3
9  N = 10^4
10 nRange = 2:2:10
11 alpha = 0.05
12
13 for n in nRange
14     coverageCount = 0
15     for _ in 1:M
16         sampleData = rand(dist, n)
17         bootstrapSampleMeans = [median(rand(sampleData, n)) for _ in 1:N]
18         L = quantile(bootstrapSampleMeans, alpha/2)
19         U = quantile(bootstrapSampleMeans, 1-alpha/2)
20         coverageCount += L < actualMedian && actualMedian < U
21     end
22     println("n = ",n,"\t coverage = ", coverageCount/M)
23 end
```

```
n = 2    coverage = 0.483
n = 4    coverage = 0.881
n = 6    coverage = 0.936
n = 8    coverage = 0.939
n = 10   coverage = 0.949
```

In line 4 we specify λ. In line 5 we create `dist` remembering that in Julia the exponential distribution is parameterized by the inverse of λ. The actual median is then computed in line 6 via `median()` method implemented in the `Distributions` package. In line 8 we specify the number of repetitions we make to evaluate the coverage probability, `M`. Then in line 9 the number of bootstrap repetitions, `N`, is specified. In line 10 we specify the range of number of observations to consider, `nRange`. We then loop over this range in lines 13-23 where in each iteration we count `coverageCount`, counting the number of times the bootstrap confidence interval contained `actualMedian`. Each actual bootstrap confidence interval procedure is in lines 17-19. Note the boolean value to the right of += in line 20 which evaluates to `true` if the median is covered by `(L,U)`.

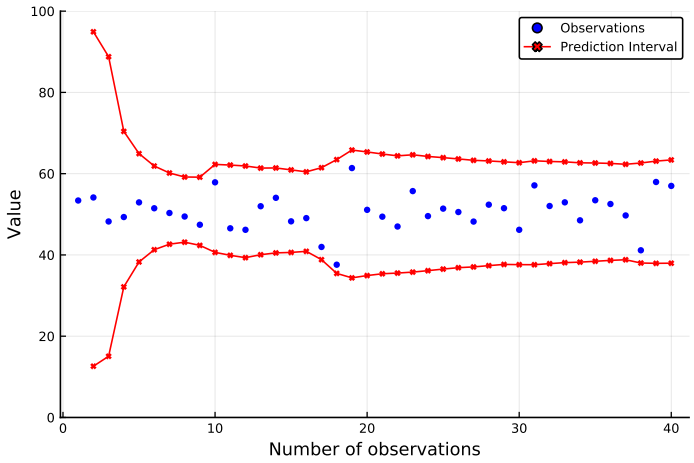

Figure 6.6: As the number of observations increases, the width of the prediction interval decreases to a constant.

6.6 Prediction Intervals

We now look at the concept of a *prediction interval* which is somewhat related to confidence intervals, however has a different meaning. A prediction interval tells us a predicted range that a single next observation of data is expected to fall within. This differs from a confidence interval which indicates how confident we are of a particular parameter that we are trying to estimate. For a given distributional model, the bounds of a prediction interval are always wider than those of a confidence interval, as the prediction interval must account for the uncertainty in knowing the population mean, as well as the spread of the data due to variance.

The example that we use is for a sequence of data points $x_1, x_2, x_3, \ldots$, which come from a normal distribution and are assumed i.i.d. Further assume that we observed $x_1, \ldots, x_n$ but have not yet observed X_{n+1}. Note that we use 'little' x for values observed and 'upper case' X for (yet) unobserved random variables.

In this case, a $100(1-\alpha)\%$ prediction interval for the single future observation, X_{n+1}, is given by,

$$\overline{x} - t_{1-\frac{\alpha}{2},n-1} s\sqrt{1+\frac{1}{n}} \leq X_{n+1} \leq \overline{x} + t_{1-\frac{\alpha}{2},n-1} s\sqrt{1+\frac{1}{n}}, \tag{6.31}$$

where, $\overline{x}$ and s are respectively the sample mean and sample standard deviation computed from $x_1, \ldots, x_n$. Note that as the number of observations, n, increases, the bounds of the prediction interval decreases towards,

$$\overline{x} - z_{1-\frac{\alpha}{2}} s \leq X_{n+1} \leq \overline{x} + z_{1-\frac{\alpha}{2}} s. \tag{6.32}$$

In Listing 6.15 we illustrate the use of prediction intervals based on a series of observations made from a normal distribution. We start with $n = 2$ observations and calculate the corresponding prediction interval for the next observation. The sample size n is then progressively increased, and the prediction interval for each next observation calculated for each subsequent case. The listing creates Figure 6.6 which illustrates that as the number of observations increases, the prediction interval width decreases. Ultimately it follows (6.32). and has an expected width close to $2\, z_{1-\frac{\alpha}{2}}\, \sigma$.

Listing 6.15: Prediction interval with unknown population mean and variance

```
1  using Random, Statistics, Distributions, Plots; pyplot()
2  Random.seed!(0)
3
4  mu, sig = 50, 5
5  dist = Normal(mu, sig)
6  alpha = 0.01
7  nMax = 40
8
9  observations = rand(dist,1)
10 piLarray, piUarray = [], []
11
12 for _ in 2:nMax
13     xNew = rand(dist)
14     push!(observations,xNew)
15
16     xbar, sd = mean(observations), std(observations)
17     n = length(observations)
18     tVal = quantile(TDist(n-1),1-alpha/2)
19     delta = tVal * sd * sqrt(1+1/n)
20     piL, piU = xbar - delta, xbar + delta
21
22     push!(piLarray,piL); push!(piUarray,piU)
23 end
24
25 scatter(1:nMax, observations,
26         c=:blue, msw=0, label="Observations")
27 plot!(2:nMax, piUarray,
28         c=:red, shape=:xcross, msw=0, label="Prediction Interval")
29 plot!(2:nMax, piLarray,
30         c=:red, shape=:xcross, msw=0, label="",
31         ylims=(0,100), xlabel="Number of observations", ylabel="Value")
```

In line 4-7 we setup the distributional parameters, choose α, and also set the limiting number of observations we will make, `nMax`. In line 9 we sample the first sample observation and store it in the array `observations`. In line 10, we create the arrays `piLarray` and `piUarray`, which will be used to store the lower and upper bounds of the prediction interval. Lines 12-23 contain the main logic of this example where in lines 13-14 a new date point is obtained and stored, in lines 16-18 updated prediction interval is calculated and in line 22, the prediction interval is stored for plotting afterwards. Lines 25-31 create Figure 6.6. Observe the use of `shpae=:xcross` for setting tick marks,

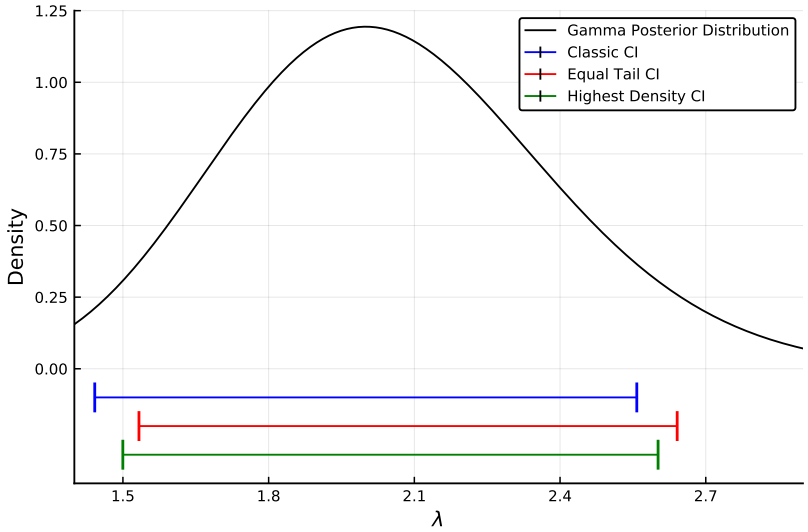

Figure 6.7: The posterior distribution and three forms of 90% credible intervals.

6.7 Credible Intervals

This section presents two related concepts, *credible intervals* and *intervals on asymmetric distributions*. The concept of credible intervals comes from the field of Bayesian statistics which we overviewed in Section 5.7 of Chapter 5. It is the analog of a confidence interval in the Bayesian setting. Before we explain this concept further we first deal with various ways of finding intervals on asymmetric distributions.

In general, we often need to find an interval $[\ell, u]$ such that given some probability density $f(x)$, the interval satisfies,

$$\int_{\ell}^{u} f(x)\,dx = 1 - \alpha. \tag{6.33}$$

This is needed for confidence intervals which were the focus for most of this chapter, for prediction intervals which were discussed in the previous section, or for credible intervals which are discussed below. However, as long as $\alpha < 1$ there is never a single unique interval $[\ell, u]$ that satisfies (6.33).

In certain cases there is a "natural" interval. For example for the normal distribution, using equal tail quantiles is natural. We do this by choosing $\ell = z_{\alpha/2}$ and $u = z_{1-\alpha/2}$ as was used throughout this chapter. Similarly for a T-distribution with $n-1$ degrees of freedom we used $\ell = t_{\alpha/2,n-1}$ and $u = t_{1-\alpha/2,n-1}$. Such choices for $[\ell, u]$ are natural because in both the case of a normal and the T-distribution, the density $f(x)$ is symmetric about the mean. In such cases, the mean is also the median and further since the density is *unimodal* (has a single maximum) then the mean and median are also the mode. With such symmetry and unimodallity, while there exist other choices of ℓ and u, they don't appeal to applications.

However, consider *asymmetric distributions* such as the one presented in Figure 6.7. In this case, there isn't an immediate "natural" choice for ℓ and u. Without going into the actual meaning of the density in the figure just yet, you can already observe three different plotted intervals. They all

satisfy (6.33) A blue *classic CI*, a red *equal tail CI*, and a green *highest density CI*. We now describe each of these intervals.

Classic interval – This type of interval has the mode of the density (assuming the density is unimodal), at its center between ℓ and u. An alternative is to use mean or median at the center. That is, assuming the centrality measure (mode, median or mean) is m, we have, $[\ell, u] = [m - E, m + E]$. One way to define E is via

$$E = \max\{\varepsilon \geq 0 \ : \ \int_{m-\varepsilon}^{m+\varepsilon} f(x)\, dx \leq 1 - \alpha\}. \tag{6.34}$$

That is, we can search for the highest ε such that the integral over $[m - \varepsilon, m + \varepsilon]$ doesn't exceed $1 - \alpha$. This is crudely implemented in Listing 6.16 in the function `classicalCI()`.

Equal tail interval – This type of interval simply sets ℓ and u as the $\alpha/2$ and $1 - \alpha/2$ quantiles respectively. Namely,

$$\frac{\alpha}{2} = \int_{-\infty}^{\ell} f(x)\, dx, \qquad \text{and} \qquad \frac{\alpha}{2} = \int_{u}^{\infty} f(x)\, dx. \tag{6.35}$$

Such an interval was implicitly used with the asymmetric Chi-squared distribution in Section 6.4. See for example formula (6.27).

Highest density interval – This type of interval seeks to cover the part of the support that is most probable. Define the smallest probability densities falling over an interval $[\ell, u]$ via $M(\ell, u) = \min\{f(x) \ : \ x \in [\ell, u]\}$. Then the highest density interval seeks for an interval $[\ell, u]$ that satisfies (6.33) while maximizing $M(\ell, u)$. A consequence is that if the density is unimodal then this highest density interval is also the narrowest possible confidence interval.

There are multiple computational ways to find such a confidence interval. In Listing 6.16 the function `highestDensityCI()` crudely does so by starting with a high density value and decreasing it gradually while seeking for the associated interval $[\ell, u]$. An alternative would be to gradually increment ℓ each time finding a corresponding u that satisfies (6.33) and within this search to choose the interval that minimizes the width $u - \ell$.

For a symmetric and unimodal distribution such as the Normal distribution or T-distribution, all three of these confidence intervals agree. However in general they don't. In a Bayesian context as we describe below, one often prefers the highest density interval. However in other settings, equal tail intervals are common. For example when considering confidence intervals for the variance, we used equal tail intervals in (6.27) because it yielded the simple confidence interval formula (6.28) which uses tabulated quantiles (of chi-squared distributions).

We now explain *credible intervals* These come instead of confidence intervals in a Bayesian setting. Recall that in the Bayesian setting, we treat the unknown parameter, θ, as a random variable. As described in Section 5.7, the process of inference is based on observing data, $x = (x_1, \ldots, x_n)$ and fusing it with the prior distribution $f(\theta)$ to obtain the posterior distribution $f(\theta \mid x)$. Here too, as in the frequentist case, we may wish to describe an interval where it is likely that our parameter lies. Then for a fixed confidence level, $1 - \alpha$, seek $[\ell, u]$ such that,

$$\int_{\ell}^{u} f(\theta \mid x)\, d\theta = 1 - \alpha.$$

Compare this with (5.18) of Chapter 5 where $[L, U]$ denotes the confidence interval. In the Bayesian case, ℓ and u are deterministic values determined from the prior distribution of the random θ, where as in the frequentist case, θ is deterministic with L and U random. This is a conceptual difference between confidence intervals and credible intervals. However practically there isn't a difference.

Nevertheless, in a Bayesian context, unless dealing with special cases of conjugacy (see Listing 5.19), the posterior distribution of the parameter is often only available numerically via MCMC or similar methods (see Listing 5.20). Hence there is no general motivation to use equal tail intervals or classical intervals. Instead, highest density intervals are often a prime choice.

In Listing 6.16 we generate Figure 6.7 and compute alternative credible intervals using classical, equal tail, and highest density methods. We deal with the same small data set that was used in Section 5.7 of Chapter 5. Here the unknown parameter is λ, the mean of a Poisson distribution. In this simple example, for simplicity we use gamma-Poisson conjugacy and thus update hyper-parameters of with simple rules as in (5.24) of Chapter 5.

Observe that by design all three confidence intervals have the same $1 - \alpha = 0.9$ coverage probability. Slight differences only appear due to numerical inaccuracy. Also note that the width of the highest density is lowest, at 1.102 Indeed, there isn't a 90% confidence interval over this prior distribution narrower. We finally note that our implementation aims to be simple and readable but not the most time-efficient nor the most numerically precise.

Listing 6.16: Credible intervals on a posterior distribution

```
using Distributions, Plots, LaTeXStrings; pyplot()

alpha, beta = 8, 2
data = [2,1,0,0,1,0,2,2,5,2,4,0,3,2,5,0]

newAlpha, newBeta = alpha + sum(data), beta + length(data)
post = Gamma(newAlpha, 1/newBeta)

xGrid = quantile(post,0.01):0.001:quantile(post,0.99)
significance = 0.9; halfAlpha = (1-significance)/2

coverage(l,u) = cdf(post,u) - cdf(post,l)

function classicalCI(dist)
    l, u = mode(dist),mode(dist)
    bestl, bestu = l, u
    while  coverage(l,u) < significance
        l -= 0.00001; u += 0.00001
    end
    (l,u)
end
equalTailCI(dist) = (quantile(dist,halfAlpha), quantile(dist,1-halfAlpha))
function highestDensityCI(dist)
    height = 0.999 * maximum(pdf.(dist,xGrid))
    l,u = mode(dist),mode(dist)
    while coverage(l,u) <= significance
        range = filter(theta -> pdf(dist,theta) > height, xGrid)
        l,u = minimum(range), maximum(range)
        height -= 0.00001
    end
    (l,u)
end

l1, u1 = classicalCI(post)
l2, u2 = equalTailCI(post)
l3, u3 = highestDensityCI(post)
println("Classical: ", (l1,u1), "\tWidth: ",u1-l1,
        "\tCoverage: ", coverage(l1,u1))
println("Equal tails: ", (l2,u2), "\tWidth: ",u2-l2,
        "\tCoverage: ", coverage(l2,u2))
println("Highest density: ", (l3,u3), "\tWidth: ",u3-l3,
        "\tCoverage: ", coverage(l3,u3))

plot(xGrid,pdf.(post,xGrid),  yticks=(0:0.25:1.25),
        c=:black, label="Gamma Posterior Distribution",
        xlims=(1.4, 2.9), ylims=(-0.4,1.25))
plot!([l1,u1],[-0.1,-0.1], label="Classic CI",
        c=:blue, shape=:vline, ms=16)
plot!([l2,u2],[-0.2,-0.2], label="Equal Tail CI",
        c=:red, shape=:vline, ms=16)
plot!([l3,u3],[-0.3,-0.3], label="Highest Density CI",
        c=:green, shape=:vline, ms=16, xlabel=L"\lambda", ylabel="Density")
```

```
Classical: (1.44, 2.56)         Width: 1.1146   Coverage: 0.90000
Equal tails: (1.53, 2.64)       Width: 1.1081   Coverage: 0.89999
Highest density: (1.51, 2.60)   Width: 1.1020   Coverage: 0.90018
```

In line 3 we define the prior hyper-parameters, `alpha` and `beta`. Line 4 defines the observations. In line 6 the posterior hyper-parameters are calculated using the Poisson-gamma conjugacy update rule. In line 7 we create an object, `post`, for the posterior distribution. In line 9 we define a fine grid for carrying out computation. In line 10 we define the confidence level $1-\alpha$ via `significance` and also use `halfAlpha` to denote $\alpha/2$. In line 12 the function `coverage()` is defined to compute the coverage probability in the posterior distribution on the interval ranging between `l` and `u`. Lines 14-32 define the three main functions of this code block, `classicalCI()`, `equalTailCI()`, and `highestDensityCI()`, each returning the respective confidence/credible interval, classical, equal tail, and highest density. The implementation of `classicalCI()` in lines 14-21 starts with `(l,u)` at the mode of the distribution and then expands outwards in small steps until the desired coverage is reached. The implementation of `equalTailCI()` in line 22 simply returns the $\alpha/2$ and $1-\alpha/2$ quantiles. The implementation of `highestDensityCI()` in lines 23-32 works by starting with `height` at the maximal possible value and decreasing it until coverage is satisfied. In this implementation, in each iteration we use `filter()` in line 27 with the anonymous function `theta -> pdf(dist,theta) > height`. This sets `range` to be the array of all values of `theta` for which the density exceeds `height`. In lines 34-36, we use our confidence/credible interval functions to obtain credible intervals for the posterior distribution. The intervals, their widths and coverage probabilities are printed in lines 37-42. Then lines 44-52 generate Figure 6.7. Note the use of `:vline` with a specification of `ms` to create the confidence intervals in the figure.

Chapter 7

Hypothesis Testing - DRAFT

In this chapter we explore hypothesis testing through a few specific practical hypothesis tests. Recall the general hypothesis test formulation first introduced in Section 5.6, where we partition the parameter space Θ as follows,

$$H_0 : \theta \in \Theta_0, \qquad H_1 : \theta \in \Theta_1.$$

One of the most common cases for a single population is to consider θ as μ, the population mean, in which case $\Theta = \mathbb{R}$. Often, we wish to test if the population mean is equal to some value, μ_0, hence we can construct a *two sided hypothesis test* as follows,

$$H_0 : \mu = \mu_0, \qquad H_1 : \mu \neq \mu_0. \tag{7.1}$$

However, one could instead chose to construct a *one sided hypothesis test*, as,

$$H_0 : \mu \leq \mu_0, \qquad H_1 : \mu > \mu_0, \tag{7.2}$$

or alternatively, in the opposite direction,

$$H_0 : \mu \geq \mu_0, \qquad H_1 : \mu < \mu_0, \tag{7.3}$$

where the choice of setting up (7.1), (7.2), or (7.3) depends on the context of the problem.

As covered in Section 5.6, once the hypothesis is established, the general approach involves calculating the *test statistic*, along with the corresponding *p-value*, and then finally making some statement about the null hypothesis based on some chosen level of significance. In this chapter we present some specific common examples often used in practice.

In Section 7.1 we introduce hypothesis testing via several examples involving a single population. In Section 7.2, we present extensions of these concepts and related ideas by looking at inference for the difference in means of two populations. In Section 7.3 we focus on methods of *Analysis of Variance (ANOVA)*. Then in Section 7.4 we explore *Chi-squared tests* and *Kolmogorov-Smirnov tests*. These latter two procedures are often used to assess goodness of fit, independence, or both. We then close with Section 7.5, where we illustrate how power curves can aid in experimental design.

As in the previous chapters, we try to strike a balance between use of `HypothesisTests` and reproducing results from fundamental calculations, with the purpose of highlighting key phenomena. Several of the examples make use of the datasets `machine1.csv` and `machine2.csv`, which represent the diameter (in mm) of pipes produced via two separate machines.

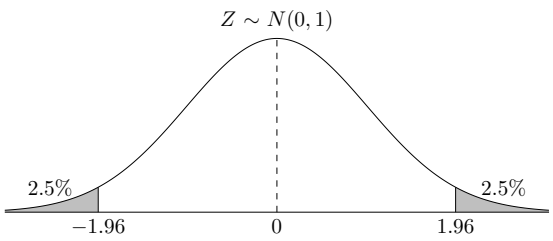

Figure 7.1: The standard normal distribution and rejection regions for the two sided hypothesis test (7.4) at significance level $\alpha = 5\%$.

7.1 Single Sample Hypothesis Tests for the Mean

As an introduction, consider the case where we wish to make inference on the mean diameter of pipes produced by a machine. Specifically, assume that we are interested in checking if the machine is producing pipes of the specified diameter $\mu_0 = 52.2$ mm. In this case, using a hypothesis testing methodology, we may wish to set-up the hypothesis as,

$$H_0 : \mu = 52.2, \qquad H_1 : \mu \neq 52.2. \tag{7.4}$$

Here, H_0 represents the situation where the machine is functioning properly, and deviation from H_0 in either the positive or negative direction is captured by H_1, which represents that the machine is malfunctioning. Alternatively, one could have treated $\mu_0 = 52.2$ as a specified upper limit on the pipe diameter, in which case the hypothesis would be formulated as,

$$H_0 : \mu \leq 52.2, \qquad H_1 : \mu > 52.2. \tag{7.5}$$

Similarly, one could envision a case where (7.3) was used instead. In most of this chapter, we do not dive deeply into the aspects of formulating the hypotheses themselves but rather the hypotheses are introduced and treated as given. If you are interested in the *experimental design* aspect of hypothesis testing, you may consult [Mon17] or similar texts.

Once the hypothesis is formulated, the next step is the collection of data, which in this section is taken from `machine1.csv`. We now separate the inference of the mean of a single population into the two cases of variance known and unknown, similarly to what was done in Section 6.1. Note also that, similarly to Chapter 6, it is assumed that the observations $X_1, \ldots, X_n$ are normally distributed. Finally, at the end of this section, we consider a simple *non-parametric* test, where we make no assumptions about the distribution of the observations.

Population Variance Known

Consider the case where we wish to test whether a single machine in a factory is producing pipes of a specified diameter. For this example, we set up the hypothesis as two sided according to (7.4), and assume that σ is a known value. Recall from Section 5.2 that, under H_0, $\overline{X}$ follows a normal distribution with mean μ_0 and variance σ^2/n. Hence it holds that under H_0 the *Z-statistic*,

$$Z = \frac{\overline{X} - \mu_0}{\sigma/\sqrt{n}}, \tag{7.6}$$

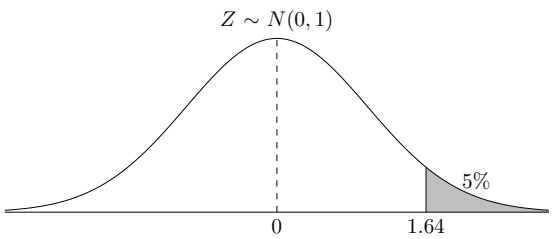

Figure 7.2: The standard normal distribution and rejection region for the one sided hypothesis test (7.5) at significance level $\alpha = 5\%$.

follows a standard normal distribution.

As we will see through the various examples in this chapter, the test statistic often follows a general form similar to that of (7.6). In this case, under the null hypothesis, the random variable Z is normally distributed, and hence to carry out a hypothesis test we observe its position relative to a standard normal distribution. Specifically, we check if it lies within the *rejection region* or not, and if it does, we reject the null hypothesis, otherwise we don't. In Figure 7.1 we present the rejection region corresponding to $\alpha = 5\%$. It is obtained by considering the $\alpha/2$ and $1-\alpha/2$ quantiles of the standard normal distribution.

With the hypothesis test and rejection region specified, we are ready to collect data, calculate the test statistic, and make a conclusion. For this example, the data is taken from `machine1.csv` where we assume $\sigma = 1.2$ (i.e. is known). After collecting the data, the observed Z-statistic is calculated via,

$$z = \frac{\bar{x} - \mu_0}{\sigma/\sqrt{n}}. \tag{7.7}$$

Since this is a two sided test, we aim for a symmetric rejection region. We then reject H_0 if $|z| > z_{1-\alpha/2}$ where $z_{1-\alpha/2}$ is the quantile of the standard normal distribution for a specific confidence level α. We may also compute the p-value of the test via,

$$p = 2\,\mathbb{P}(Z > |z|). \tag{7.8}$$

That is, we consider the observed test statistic, z, and determine the maximal significance level for which we would reject H_0. Hence, for a fixed significance level α, if $p < \alpha$, we reject H_0 and otherwise not. The calculation of critical values such as $z_{1-\alpha/2}$ and p-values is typically done via software, or more traditionally via *statistical tables*, which list the area under the normal curve along with different quantiles of a standard normal. For example $z_{0.025} = -1.96$, or $z_{0.975} = 1.96$. For $\alpha = 0.05$, we reject the null hypothesis if $z > 1.96$ or $z < -1.96$, otherwise we don't reject.

If the null hypothesis is rejected then we conclude the test by stating, "*there is sufficient evidence to reject the null hypothesis at the* 5% *significance level*". Otherwise we conclude by stating, "*there is insufficient evidence to reject the null hypothesis at the* 5% *significance level*". Note that if a different hypothesis test setup was used, such as (7.5), then the rejection region would not be symmetric as in Figure 7.1, but rather would cover only one tail of the distribution. This is illustrated in Figure 7.2, where $z_{0.95} = 1.645$ is used to determine the boundary of the rejection region. In such a case, the p-value is calculated via $p = \mathbb{P}(Z > z)$.

In Listing 7.1, we present an example containing two hypothesis tests (using the same data)

where the first (`mu0A`) is rejected and the second (`mu0B`) is not-rejected. For the `mu0A` case, the test statistic is first calculated via (7.7) along with the corresponding p-value via (7.8). Then the `HypothesisTests` package is used to perform the same hypothesis test for both `mu0A` and `mu0B`. The default test assumes $\alpha = 5\%$. Observe that the p-value in the `mu0A` case is less than 0.05 and hence H_0 is rejected. In comparison, for the `mu0B` case, the p-value is greater than 0.05 and hence H_0 is not rejected.

Listing 7.1: Inference with single sample, population variance is known

```
using CSV, Distributions, HypothesisTests

data = CSV.read("../data/machine1.csv", header=false)[:,1]
xBar, n = mean(data), length(data)
sigma = 1.2
mu0A, mu0B = 52.2, 53

testStatistic = ( xBar - mu0A ) / ( sigma/sqrt(n) )
pVal = 2*ccdf(Normal(), abs(testStatistic))

testA = OneSampleZTest(xBar, sigma, n, mu0A)
testB = OneSampleZTest(xBar, sigma, n, mu0B)

println("Results for mu0 = ", mu0A,":")
println("Manually calculated test statistic: ", testStatistic)
println("Manually calculated p-value: ", pVal,"\n")
println(testA)

println("\n In case of  mu0 = ", mu0B, " then p-value = ", pvalue(testB))
```

```
Results for mu0 = 52.2:

Manually calculated test statistic: 2.8182138203055467
Manually calculated p-value: 0.004829163880878602

One sample z-test
-----------------
Population details:
    parameter of interest:   Mean
    value under h_0:         52.2
    point estimate:          52.95620612127991
    95% confidence interval: (52.4303, 53.4821)

Test summary:
    outcome with 95% confidence: reject h_0
    two-sided p-value:           0.0048

Details:
    number of observations:   20
    z-statistic:              2.8182138203055467
    population standard error: 0.2683281572999747


 In case of  mu0 = 53 then p-value = 0.870352975060586
```

In lines 3-6 we load the data, calculate the sample mean, and specify the values of `mu0A` and `mu0B` under H_0 (there are two separate tests in this example). Note that importantly, the standard deviation, `sigma`, is specified as `1.2`, as it is 'known'. In line 8 we calculate the test statistic for case `mu0A` according to (7.7). In line 9 we calculate the p-value according to (7.8). Note that the `ccdf()` function is used to find the area to the right of the absolute value of the test statistic. In lines 11 and 12, `OneSampleZTest()` from `HypothesisTests` is used to perform the same calculations for both the `mu0A` and `mu0B` case. The results are stored in `testA` and `testB`. These objects can then be printed or queried. Note that `OneSampleZTest()` was called with 4 arguments. If the last argument (`mu0A` or `mu0B`) was excluded, then the function would have performed the one sample z test assuming $\mu_0 = 0$. There is also an additional method for `OneSampleZTest()`, which simply takes a single argument of an array of values. In this case it will use the sample standard deviation as the population standard deviation, and will assume $\mu_0 = 0$. Further information is available in the documentation of the `HypothesisTests` package. Lines 14–17 print the results for the `mu0A` case. The p-value of 0.0048 merits rejection of H_0 for $\alpha = 5\%$. The output from line 17 also lists the value of the parameter under H_0, the point estimate of the parameter (`xBar`), as well as the corresponding 95% confidence interval. Line 19 prints the p-value for the second hypothesis test which uses `mu0B`, and since the p-value is greater than 0.05, we do not reject H_0. Note the use of the `pvalue()` function applied to `testB`. This way of using the `HypothesisTests` package is based on creating an object (`testB` in this case) and then querying it using a function like `pvalue()`.

Population Variance Unknown

Having covered the case of variance known, we now consider the more realistic scenario where the population variance is unknown. Informally called the *T-test*, this is perhaps the most famous and widely used hypothesis test in elementary statistics. Here the test statistic is the *T-statistic*,

$$T = \frac{\overline{X} - \mu_0}{S/\sqrt{n}}. \tag{7.9}$$

Notice that it is similar to (7.6), however the sample standard deviation, S, is used instead of the population standard deviation σ, since σ is unknown. As presented in Section 5.2, in the case where the data is normally distributed with mean μ_0, the random variable T follows a T-distribution with $n-1$ degrees of freedom and this is the basis for the *T-test*. The procedure is the same as the Z-test presented above, except that a T-distribution is used instead of a normal distribution. Note that for non-small n, the T-distribution is almost identical to a standard normal distribution.

The observed test statistic from the data is then,

$$t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}}, \tag{7.10}$$

and the corresponding p-value for a two sided test is

$$p = 2\,\mathbb{P}(T_{n-1} > |t|), \tag{7.11}$$

where T_{n-1} is a random variable distributed according to a T-distribution with $n-1$ degrees of freedom. Note that standardized tables present *critical values* of the T-distribution, namely t_γ where γ is typically $0.9, 0.95, 0.975, 0.99$ and 0.995. These are typically presented in detail for degrees of freedom ranging from $n = 2$ to $n = 30$, after which the T-distribution is very similar to a normal

distribution. These values are then compared to the T-statistic (7.10) where $\gamma = 1 - \alpha$ in the one sided case or $\gamma = 1 - \alpha/2$ in the two sided case. However, for precise calculation of p-values, software must be used.

In Listing 7.2 we first calculate the test statistic via (7.10), and then use this to manually calculate the corresponding p-value via (7.11). Then `OneSampleTTest()` from the `HypothesisTests` package is used to perform the same hypothesis. The output from our manual calculation matches that of `OneSampleTTest()`.

Listing 7.2: Inference with single sample, population variance unknown

```
1  using CSV, Distributions, HypothesisTests
2
3  data = CSV.read("../data/machine1.csv", header=false)[:,1]
4  xBar, n = mean(data), length(data)
5  s = std(data)
6  mu0 = 52.2
7
8  testStatistic = ( xBar - mu0 ) / ( s/sqrt(n) )
9  pVal = 2*ccdf(TDist(n-1), abs(testStatistic))
10
11 println("Manually calculated test statistic: ", testStatistic)
12 println("Manually calculated p-value: ", pVal,"\n")
13 OneSampleTTest(data, mu0)
```

```
Manually calculated test statistic: 2.86553950269453
Manually calculated p-value: 0.009899631865162935

One sample t-test
-----------------
Population details:
    parameter of interest:   Mean
    value under h_0:         52.2
    point estimate:          52.95620612127991
    95% confidence interval: (52.4039, 53.5085)

Test summary:
    outcome with 95% confidence: reject h_0
    two-sided p-value:           0.0099

Details:
    number of observations:   20
    t-statistic:              2.86553950269453
    degrees of freedom:       19
    empirical standard error: 0.2638965962845154
```

Lines 1–9 are similar to Listing 7.1. In line 5 the sample standard deviation is calculated and stored as `s`. In lines 8 and 9 the test statistic and p-value are calculated according to (7.10) and (7.11) respectively. Here the `ccdf()` function is used on a T-distribution with $n - 1$ degrees of freedom, `TDist(n-1)`. The manual calculations are output in lines 11 and 12. In line 13 `OneSampleTTest()` is used to perform the same hypothesis test on the data. Note that in this case we only specify two arguments, the array of our data, and the value of μ_0, `mu0`.

A Non-parametric Sign Test

The validity of the T-test relies heavily on the assumption that the sample $X_1, \ldots, X_n$ is comprised of independent normal random variables. This is because only under this assumption does the T-statistic follow a T-distribution. This assumption may often be safely made, however in certain cases we cannot assume a normal population and we need an alternative test.

Here we present a particular type of *non-parametric test* known as the *sign test*. The phrase "non-parametric" implies that the distribution of the test statistic does not depend on any particular distributional assumption for the population.

For the sign test, we begin by denoting the random variables,

$$X^+ = \sum_{i=1}^{n} \mathbf{1}\{X_i > \mu_0\} \qquad \text{and} \qquad X^- = \sum_{i=1}^{n} \mathbf{1}\{X_i < \mu_0\} = n - X^+. \tag{7.12}$$

where $\mathbf{1}\{\cdot\}$ is the indicator function. The variable X^+ is a count of the number of observations that exceed μ_0, and similarly, X^- is a count of the number of observations that are below μ_0.

Observe that under $H_0 : \mu = \mu_0$, it holds that $\mathbb{P}(X_i > \mu_0) = \mathbb{P}(X_i < \mu_0) = 1/2$. Note that here we are actually taking μ_0 as the median of the distribution and assuming that $\mathbb{P}(X_i = \mu_0) = 0$ as is the case for a continuous distribution. Hence under H_0 the random variables X^+ and X^- both follow a binomial$(n, 1/2)$ distribution (see Section 3.5). Given the symmetry of this binomial distribution we define the test statistic to be,

$$U = \max\{X^+, X^-\}. \tag{7.13}$$

Hence with observed data, and an observed test statistic u, the p-value can be calculated via,

$$p = 2\,\mathbb{P}(B > u), \tag{7.14}$$

where B is a binomial$(n, 1/2)$ random variable. Here, under H_0, p is the probability of getting an extreme number of signs greater than u (either too many via X^+ or a very small number via X^-). The test procedure is then to reject H_0 if $p < \alpha$.

In Listing 7.3 we present an example where we calculate the value of the test statistic and its corresponding p-value manually. We then use these to make conclusions about the null hypothesis at the 5% significance level. As was done in Listing 7.1 we compare two hypothetical cases. In the first case $\mu_0 = 52.2$, and the second $\mu_0 = 53.0$. As can be observed from the output, the former case is significant (H_0 is rejected) while the latter is not, as the test statistic of 11 is not non-plausible under H_0.

Listing 7.3: Non-parametric sign test

```
1  using CSV, Distributions, HypothesisTests
2
3  data = CSV.read("../data/machine1.csv", header=false)[:,1]
4  n = length(data)
5  mu0A, mu0B = 52.2, 53
6
7  xPositiveA = sum(data .> mu0A)
8  testStatisticA = max(xPositiveA, n-xPositiveA)
9
10 xPositiveB = sum(data .> mu0B)
11 testStatisticB = max(xPositiveB, n-xPositiveB)
12
13 binom = Binomial(n,0.5)
14 pValA = 2*ccdf(binom, testStatisticA)
15 pValB = 2*ccdf(binom, testStatisticB)
16
17 println("Binomial mean: ", mean(binom))
18
19 println("Case A: mu0: ", mu0A)
20 println("\tTest statistc: ", testStatisticA)
21 println("\tP-value: ", pValA)
22
23 println("Case B: mu0: ", mu0B)
24 println("\tTest statistc: ", testStatisticB)
25 println("\tP-value: ", pValB)
```

```
Binomial mean: 10.0

Case A: mu0: 52.2
        Test statistc: 15
        p-value: 0.011817932128906257
Case B: mu0: 53
        Test statistc: 11
        p-value: 0.5034446716308596
```

In line 5 the value of the population mean under the null hypothesis for both cases, `mu0A` and `mu0B`, is specified. In lines 7–11 the observed test statistics for both cases are calculated via (7.13). Note the use of `.>` for comparing the array `data` element wise with the scalars `mu0A` and `mu0B`. In lines 13–15, `Binomial()` and `ccdf()` are used used to compute the p-values for both cases in via (7.14). The results are printed in lines 19–25. As there are $n = 20$ observations the binomial mean is 10.

Sign Test vs. T-Test

With the sign test presented as a robust alternative to the T-test, one may ask why not simply always use the sign test. After all, the validity of the T-test rests on the assumption that $X_1, \ldots, X_n$ are normally distributed. Otherwise, T of (7.9) does not follow a T-distribution, and conclusions drawn from the test may be potentially imprecise.

One answer is due to the statistical powerof the test. As we show in the example below, the T-test is a more powerful test than the sign test when the normality assumption holds. That is,

for a fixed α, the probability of detecting H_1 is higher for the T-test than for the sign test. This makes it a more effective test to use, if the data can be assumed normally distributed. The concept of power was first introduced in Section 5.6, and is further investigated in Section 7.5

In Listing 7.4 we perform a two-sided hypothesis test for $H_0 : \mu = 53$ vs. $H_1 : \mu \neq 53$ via both the T-test and sign test. We consider a range of scenarios where we let the actual μ vary over $[51.0, 55.0]$. When $\mu = 53$, H_0 is the case, however all other cases fall in H_1. On a grid of such cases we use Monte Carlo to estimate the power of the tests (for $\sigma = 1.2$). The resulting curves in Figure 7.3 show that the T-test is more powerful than the sign test.

Listing 7.4: Comparison of sign test and T-test

```
using Random, Distributions, HypothesisTests, Plots; pyplot()

muRange = 51:0.02:55
n = 20
N = 10^4
mu0 = 53.0
powerT, powerU = [], []

for muActual in muRange

    dist = Normal(muActual, 1.2)
    rejectT, rejectU = 0, 0
    Random.seed!(1)

    for _ in 1:N
        data = rand(dist,n)
        xBar, stdDev = mean(data), std(data)

        tStatT = (xBar - mu0)/(stdDev/sqrt(n))
        pValT  = 2*ccdf(TDist(n-1), abs(tStatT))

        xPositive = sum(data .> mu0)
        uStat = max(xPositive, n-xPositive)
        pValSign = 2*cdf(Binomial(n,0.5), uStat)

        rejectT += pValT < 0.05
        rejectU += pValSign < 0.05
    end

    push!(powerT, rejectT/N)
    push!(powerU, rejectU/N)

end

plot(muRange, powerT, c=:blue, label="t test")
plot!(muRange, powerU, c=:red, label="Sign test",
        xlims=(51,55), ylims=(0,1),
        xlabel="Different values of muActual",
        ylabel="Proportion of times H0 rejected", legend=:bottomleft)
```

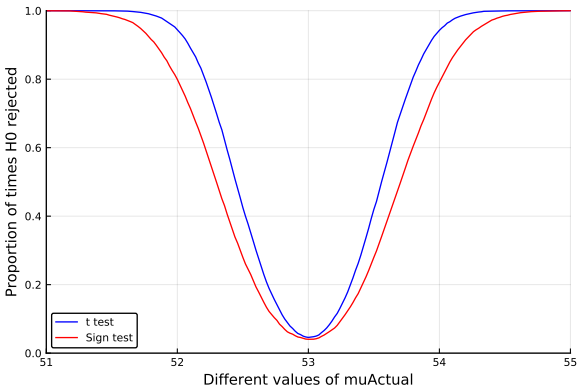

Figure 7.3: Power of the T-test vs. power of the sign-test.

In lines 3–6 we setup the basic parameters. The sample size `n` is 20. The range `muRange` represents the range $[51.0, 55.0]$ in discrete steps of 0.02. The number `N` is the number of simulation repetitions to carry out for each value of μ in that range. The value `mu0` is the value under H_0. In line 7 we initialize empty arrays that are to be populated with power estimates. Lines 9–33 contain the main loop where each discrete step in `muRange` is tested. In line 11 for each value a distribution `dist` is created with the same standard deviation, `1.2`, but with a different mean, `muActual`. The inner loop of lines 15–28 is a repetition of the sampling experiment `N` times where for the same `data` we compute `tStat`, `uStat`, and the corresponding p-values. Rejection counts are accumulated in lines 26 and 27, where `pValT < 0.05` and `pValSign < 0.05` constitutes a rejection. Note `Random.seed!()` is used in Line 13 so to obtain a smoother curve via common random numbers. See Section 10.6. Lines 30–31 record the power for the respective `muActual` by appending to the arrays using `push!()`. Lines 35–49 plot the power curves showing the superiority of the T-test. Observe that at $\mu = 53$ the power is identical to $\alpha = 0.05$.

7.2 Two Sample Hypothesis Tests for Comparing Means

Having dealt with several examples involving one population, we now present some common hypothesis tests for the inference on the difference in means of two populations. As with all hypothesis tests we start by first establishing the testing methodology. Commonly we wish to investigate if the population difference, Δ_0, takes on a specific value. Hence we may wish to set up a two sided hypothesis test as,

$$H_0 : \mu_1 - \mu_2 = \Delta_0, \qquad H_1 : \mu_1 - \mu_2 \neq \Delta_0. \tag{7.15}$$

Alternatively, one could formulate a one sided hypothesis test, such as,

$$H_0 : \mu_1 - \mu_2 \leq \Delta_0, \qquad H_1 : \mu_1 - \mu_2 > \Delta_0, \tag{7.16}$$

or the reverse if desired. It is common to consider $\Delta_0 = 0$, in which case (7.15) would be stated as $H_0 : \mu_1 = \mu_2$, and similarly (7.16) as $H_0 : \mu_1 \leq \mu_2$. Once the testing methodology has been established, the approach then follows the same outline as that covered previously, the test statistic is calculated along with its corresponding p-value, which is then used to make some conclusion about the hypothesis for some significance level α.

For the tests introduced in this section we assume that the observations $X_1^{(1)}, \ldots, X_{n_1}^{(1)}$ from population 1 and $X_1^{(2)}, \ldots, X_{n_2}^{(2)}$ from population 2 are all normally distributed, where $X_i^{(j)}$ has mean μ_j and variance σ_j^2. The testing methodology then differs based on the following three cases,

(I) The population variances σ_1 and σ_2 are known.

(II) The population variances σ_1 and σ_2 are unknown and assumed equal.

(III) The population variances σ_1 and σ_2 are unknown, and not assumed equal.

In each of these cases, the test statistic is given by,

$$\frac{\overline{X}_1 - \overline{X}_2 - \Delta_0}{S_{\text{err}}}, \tag{7.17}$$

where $\overline{X}_j$ is the sample mean of $X_1^{(j)}, \ldots, X_{n_j}^{(j)}$, and the standard error S_{err} varies according to the case (**I**–**III**). In each example, the two datasets `machine1.csv` and `machine2.csv` are used, and it is considered that H_0 implies that both machines are identical (i.e. we use (7.15) with $\Delta_0 = 0$).

Population Variances Known

In case **(I)**, where the population variances σ_1^2 and σ_2^2 are known, we set,

$$S_{\text{err}} = \sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}.$$

In this case, S_{err} is not a random quantity, and hence the test statistic (7.17) follows a standard normal distribution under H_0. This is due to the distribution of $\overline{X}_j$ as described in Section 5.2. For this case, the observed test statistic is,

$$z = \frac{(\overline{x}_1 - \overline{x}_2) - \Delta_0}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}. \tag{7.18}$$

At this point, z is used for hypothesis tests in a manner analogous to the single sample test for the population mean when the variance known, as described at the start of Section 7.1.

Note that in reality it is highly unlikely that both the population variances would be known, hence the `HypothesisTests` package does not contain functionality for this test. Nevertheless, in Listing 7.5 we perform this hypothesis test manually for pedagogical completeness. The output shows there is a very significant difference between the machines with the p-value almost zero.

Listing 7.5: Inference on difference of two means with variances known

```
1  using CSV, Distributions, HypothesisTests
2
3  data1 = CSV.read("../data/machine1.csv", header=false)[:,1]
4  data2 = CSV.read("../data/machine2.csv", header=false)[:,1]
5  xBar1, n1 = mean(data1), length(data1)
6  xBar2, n2 = mean(data2), length(data2)
7  sig1, sig2 = 1.2, 1.6
8  delta0 = 0
9
10 testStatistic = ( xBar1-xBar2 - delta0 ) / ( sqrt( sig1^2 / n1 + sig2^2 / n2 ) )
11 pVal = 2*ccdf(Normal(), abs(testStatistic))
12
13 println("Sample mean machine 1: ",xBar1)
14 println("Sample mean machine 2: ",xBar2)
15 println("Manually calculated test statistc: ", testStatistic)
16 println("Manually calculated p-value: ", pVal)
```

```
Sample mean machine 1: 52.95620612127991
Sample mean machine 2: 50.94739671468099
Manually calculated test statistc: 4.340167327618076
Manually calculated p-value: 1.423742605667141e-5
```

In lines 3–7 we load our data, calculate the sample means, and specify the values of the population variances. In line 8 we specify the value of our test parameter under the null hypothesis, `delta0`, as 0. In line 10 we calculate the test statistic via (7.18), and in line 11 we calculate the p-value.

Population Variances Unknown and Assumed Equal

We now consider case **(II)** where the population variances are unknown, and assumed equal ($\sigma^2 := \sigma_1^2 = \sigma_2^2$). In this case the pooled sample variance, S_p^2, is used to estimate σ^2 based on both samples. As covered in Section 6.2, it is given by,

$$S_p^2 = \frac{(n_1 - 1)S_1^2 + (n_2 - 1)S_2^2}{n_1 + n_2 - 2}, \tag{7.19}$$

where S_j^2 is the sample variance of sample j. It can be shown that under H_0, if we set,

$$S_{\text{err}} = S_p\sqrt{\frac{1}{n_1} + \frac{1}{n_2}},$$

the test statistic is distributed according to a T-distribution with $n_1 + n_2 - 2$ degrees of freedom. For this case the observed test statistic is,

$$t = \frac{(\overline{x}_1 - \overline{x}_2) - \Delta_0}{s_p\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}}, \tag{7.20}$$

where s_p is the observed pooled sample variance. At this point the procedure follows similar lines to the single sample T-test described in the previous section. Note that this two sample T-test with equal variance is one of the most commonly used tests in statistics.

.

In Listing 7.6 we present an example where we perform a two sided hypothesis test on the difference in means of pipes produced from machines 1 and 2. First the test is performed manually, and then the `EqualVarianceTTest()` function from the `HypothesisTests` package is used. The resulting output shows that the manually calculated values match those given by `EqualVarianceTTest()`.

Listing 7.6: Inference on difference of means, variances unknown, assumed equal

```
using CSV, Distributions, HypothesisTests

data1 = CSV.read("../data/machine1.csv", header=false)[:,1]
data2 = CSV.read("../data/machine2.csv", header=false)[:,1]
xBar1, s1, n1 = mean(data1), std(data1), length(data1)
xBar2, s2, n2 = mean(data2), std(data2), length(data2)
delta0 = 0

sP = sqrt( ( (n1-1)*s1^2 + (n2-1)*s2^2 ) / (n1 + n2 - 2) )
testStatistic = ( xBar1-xBar2 - delta0 ) / ( sP * sqrt( 1/n1 + 1/n2) )
pVal = 2*ccdf(TDist(n1+n2 -2), abs(testStatistic))

println("Manually calculated test statistic: ", testStatistic)
println("Manually calculated p-value: ", pVal, "\n")
println(EqualVarianceTTest(data1, data2, delta0))
```

```
Manually calculated test statistic: 4.5466542394674425
Manually calculated p-value: 5.9493058655043084e-5

Two sample t-test (equal variance)
----------------------------------
Population details:
    parameter of interest:   Mean difference
    value under h_0:         0
    point estimate:          2.008809406598921
    95% confidence interval: (1.1128, 2.9049)

Test summary:
    outcome with 95% confidence: reject h_0
    two-sided p-value:           <1e-4

Details:
    number of observations:   [20,18]
    t-statistic:              4.5466542394674425
    degrees of freedom:       36
    empirical standard error: 0.44182145832893077
```

Lines 3-7 are similar to Listing 7.5, however note the calculations of the sample standard deviations `s1` and `s2`. In line 7 we specify the value of our test parameter under the null hypothesis as 0. Line 9 calculates the square root of the pooled sample variance, `sP`, via (7.19). Note the use of `sqrt()`, as the test statistic (7.20) makes use of s_p not s_p^2. Lines 10 and 11 calculate the test statistic via (7.20), and the corresponding p-value respectively. These are printed as output in lines 13 and 14. Line 15 uses the `EqualVarianceTTest()` function to perform the hypothesis test. Note three arguments are given, the two arrays `data1` and `data2`, and the value of the test parameter under H_0. Note the function defaults to $\Delta_0 = 0$ if only two arguments are given, but here we demonstrate the general use of the function.

Population Variances Unknown, and not Assumed Equal

In case **(III)** where the population variances are unknown and not assumed equal ($\sigma_1^2 \neq \sigma_2^2$), we set,

$$S_{\text{err}} = \sqrt{\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}}.$$

Then the observed test statistic is given by

$$t = \frac{(\overline{x}_1 - \overline{x}_2) - \Delta_0}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}. \tag{7.21}$$

As covered in Section 6.2, the distribution of the test statistic does not follow an exact T-distribution. Instead, we use the *Satterthwaite approximation*, and determine the degrees of freedom via,

$$v = \frac{\left(s_1^2\ n_1 + s_2^2\ n_2\right)^2}{\frac{\left(s_1^2/n_1\right)^2}{n_1 - 1} + \frac{\left(s_2^2/n_2\right)^2}{n_2 - 1}}. \tag{7.22}$$

In Listing 7.7 we perform a two sided hypothesis test that the difference between the population means is zero ($\Delta_0 = 0$). We first manually calculate the test statistic and p-value, and then make use of the `UnequalVarianceTTest()` function from the `HypothesisTests` package. The output shows both methods are in agreement.

Listing 7.7: Inference on difference of means, variances unknown, not assumed equal

```
1  using CSV, Distributions, HypothesisTests
2
3  data1 = CSV.read("../data/machine1.csv", header=false)[:,1]
4  data2 = CSV.read("../data/machine2.csv", header=false)[:,1]
5  xBar1, s1, n1 = mean(data1), std(data1), length(data1)
6  xBar2, s2, n2 = mean(data2), std(data2), length(data2)
7  delta0 = 0
8
9  v = ( s1^2/n1 + s2^2/n2 )^2 / ( (s1^2/n1)^2/(n1-1) + (s2^2/n2)^2/(n2-1)  )
10 testStatistic = ( xBar1-xBar2 - delta0 )  / sqrt( s1^2/n1 + s2^2/n2 )
11 pVal = 2*ccdf(TDist(v), abs(testStatistic))
12
13 println("Manually calculated degrees of freedom, v: ", v)
14 println("Manually calculated test statistic: ", testStatistic)
15 println("Manually calculated p-value: ", pVal, "\n")
16 println(UnequalVarianceTTest(data1, data2, delta0))
```

```
Manually calculated degrees of freedom, v: 31.82453144280283
Manually calculated test statistic: 4.483705005611673
Manually calculated p-value: 8.936189820683007e-5

Two sample t-test (unequal variance)
------------------------------------
Population details:
    parameter of interest:   Mean difference
    value under h_0:         0
    point estimate:          2.008809406598921
    95% confidence interval: (1.096, 2.9216)

Test summary:
    outcome with 95% confidence: reject h_0
    two-sided p-value:           <1e-4

Details:
    number of observations:   [20,18]
    t-statistic:              4.483705005611673
    degrees of freedom:       31.82453144280282
    empirical standard error: 0.4480244360600785
```

Lines 3–8 are similar to Listing 7.6. In line 9 the degrees of freedom, `v`, is calculated via (7.22). In line 10 the test statistic is calcualted via (7.21). In line 11 these are both used to calcualte the p-value. Lines 13-16 outputs the degrees of freedom, test statistic, and p-value calculated. Line 16 uses the `UnequalVarianceTTest()` function to perform the hypothesis test.

7.3 Analysis of Variance (ANOVA)

The methods presented in Section 7.2 handle the problem of comparing means of two populations. However, what if there are more than two populations that need to be compared? This is often the case in biological, agricultural, and medical trials, among other fields, where it is of interest to see if various 'treatments', also known as 'groups', have an effect on some mean value or not. In these cases each type of treatment is considered as a different population.

More formally, assume that there is some *overall mean* μ and there are $L \geq 2$ treatments, where each treatment may potentially alter the mean by τ_i. In this case, the mean of the population under treatment i can be represented by $\mu_i = \mu + \tau_i$, with μ an overall mean and,

$$\sum_{i=1}^{L} \tau_i = 0.$$

This condition on the parameters $\tau_1, \ldots, \tau_L$, ensures that given $\mu_1, \ldots, \mu_L$, the overall mean μ and $\tau_1, \ldots, \tau_L$ are well defined.

The question is then: *Do the treatments have any effect or not*? Such a question is presented via the hypothesis formulation:

$$H_0 : \tau_1 = \tau_2 = \ldots = \tau_L = 0, \qquad \text{vs.} \qquad H_1 : \exists\ i \ \mid\ \tau_i \neq 0. \tag{7.23}$$

Note that H_0 is equivalent to the statement that $\mu_1 = \mu_2 = \ldots = \mu_L$, indicating that the treatments do not have an effect. Furthermore, H_1 stating that there exists an i with $\tau_i \neq 0$, is equivalent to the case where there exist at least two treatments, i and j such that $\mu_i \neq \mu_j$. In other words this means that the choice of treatment has an effect, at least between some treatments.

In conducting hypotheses tests such as (7.23), we collect observations (data) as follows,

$$\begin{aligned}
&\text{Treatment 1:} && x_{11},\ x_{12}, \ldots,\ x_{1n_1},\\
&\text{Treatment 2:} && x_{21},\ x_{22}, \ldots,\ x_{2n_2},\\
& && \vdots\\
&\text{Treatment L:} && x_{L1},\ x_{L2}, \ldots,\ x_{Ln_L},
\end{aligned}$$

where $n_1, n_2, \ldots, n_L$ are the sample sizes for each of the treatments (or groups). If all samples are the same size (say $n_i = n$ for all i) then this is called a *balanced design* problem. However, often different treatments have different sample sizes. It is also convenient to denote the total number of observations via,

$$m = \sum_{i=1}^{L} n_i. \tag{7.24}$$

Note that in a balanced design we have $m = L\,n$.

We focus on an example for three treatments where data from `machine1.csv`, `machine2.csv`, and `machine3.csv` represent sample measurements of the diameter of pipes produced by three different machines. Here each machine is a different treatment. The diameters of the pipes vary due to imprecision of machines but also potentially (if H_0 does not hold) due to variability between the

machines. A box plot of this dataset was already presented in Figure 4.9 of Chapter 4. Looking at that plot, while there are differences between the groups, it isn't immediately clear if the machine affects the pipe diameter or if the differences are just due to noise in manufacturing within each machine.

Once the data is collected, in addition to displaying a box-plot as in Figure 4.9, we also consider the values of the sample means for each individual treatment,

$$\bar{x}_i = \frac{1}{n_i}\sum_{j=1}^{n_i} x_{ij}.$$

These values can then be compared with the overall sample mean,

$$\overline{x} = \frac{1}{m}\sum_{i=1}^{L}\sum_{j=1}^{n_i} x_{ij} = \sum_{i=1}^{L}\frac{n_i}{m}\,\bar{x}_i. \tag{7.25}$$

In Listing 7.8 the sample means are computed for three datasets of pipe diameters. Note the use of the broadcast operator in lines 4 and 7.

Listing 7.8: Sample means for ANOVA

```
1  using CSV, Statistics
2
3  rfile(name) = CSV.read(name, header=false)[:,1]
4  data = rfile.(["../data/machine1.csv",
5                 "../data/machine2.csv",
6                 "../data/machine3.csv"])
7  println("Sample means for each treatment: ",round.(mean.(data),digits=2))
8  println("Overall sample mean: ",round(mean(vcat(data...)),digits=2))
```

```
Sample means for each treatment: [52.96, 50.95, 51.43]
Overall sample mean: 51.82
```

Even though the sample mean values are not exactly the same (they are at 52.96, 50.95 and 51.43), without looking at variability we can't conclude if the machine type (treatment) affects the diameter size or not. Here is where ANOVA comes into play.

The typical way to establish whether or not an effect between the treatments exists is to examine the variability of the individual treatment means, and compare these to the overall variability of the observations. If the variability of means significantly exceeds the variability of the individual observations, then H_0 is rejected, otherwise it is not. Such an approach is called *ANOVA*, which stands for *analysis of variance*, and it is based on the decomposition of the sum of squares. In fact ANOVA is a broad collection of statistical methods, and here we only provide an introduction to ANOVA by covering the *one-way ANOVA* test. In this test, the statistical model assumes that the observations of each treatment group come from an underlying model of the following form,

$$X_i = \mu_i = \mu + \tau_i + \varepsilon \qquad \text{where} \qquad \varepsilon \sim N(0,\sigma^2), \tag{7.26}$$

where X_i is the model for the ith treatment group and ε is some noise term with common unknown variance across all treatment groups, independent across measurements. In this sense, the

ANOVA model (7.26) generalizes the assumptions of the T-test applied to case **II** (comparison of two population means with variance unknown and assumed equal), as presented in the previous section.

The process of conducting a *one-way ANOVA test* follows the same general approach as any other hypothesis test. First the test statistic is calculated, then the corresponding p-value is obtained, and finally the p-value is used to make some conclusion about whether or not to reject H_0 at some chosen confidence level α. The test statistic for ANOVA, known as *F-statistic*, is the ratio of the average variance between the groups, divided by the average variance within the groups. Under the null hypothesis, the F-value is distributed according to the *F-distribution*, covered at the end of Section 5.2. Hence the ANOVA test is sometimes referred to as the *F-test*.

Before we present the details of carrying out an F-test, we present the mathematical motivation used to calculate the variability within the groups (or treatments) and the variability between the groups.

Decomposing Sum of Squares

A key idea of ANOVA is the decomposition of the total variability into two components: the variability between the treatments, and the variability within the treatments. There are explicit expressions for both, and here we show how to derive them by performing what is known as the *decomposition of the sum of squares*.

The total sum of squares, also known as the *sum of squares total* (SS_{Total}), is a measure of the total variability of all observations, and is calculated as follows,

$$SS_{\text{Total}} = \sum_{i=1}^{L}\sum_{j=1}^{n_i} \big(x_{ij} - \bar{x}\big)^2, \tag{7.27}$$

where $\bar{x}$ is given by (7.25). Now through algebraic manipulation (adding and subtracting treatment means) we can show that SS_{Total} can be decomposed as follows,

$$\begin{aligned}
\sum_{i=1}^{L}\sum_{j=1}^{n_i} \big(x_{ij} - \bar{x}\big)^2 &= \sum_{i=1}^{L}\sum_{j=1}^{n_i} \big(x_{ij} - \bar{x}_i + \bar{x}_i - \bar{x}\big)^2 \\
&= \sum_{i=1}^{L}\sum_{j=1}^{n_i} \Big(\big(x_{ij} - \bar{x}_i\big)^2 - 2\big(x_{ij} - \bar{x}_i\big)\big(\bar{x}_i - \bar{x}\big) + \big(\bar{x}_i - \bar{x}\big)^2 \Big) \\
&= \underbrace{\sum_{i=1}^{L}\sum_{j=1}^{n_i} \big(x_{ij} - \bar{x}_i\big)^2}_{SS_{\text{Error}}} + \underbrace{\sum_{i=1}^{L} n_i\big(\bar{x}_i - \bar{x}\big)^2}_{SS_{\text{Treatment}}}.
\end{aligned} \tag{7.28}$$

Note that on the second line, the middle term reduces to zero, since $\sum_{j=1}^{n_i}(x_{ij} - \bar{x}_i) = 0$. Hence we have shown that the total sum of squares, SS_{Total}, can be decomposed to,

$$SS_{\text{Total}} = SS_{\text{Error}} + SS_{\text{Treatment}}. \tag{7.29}$$

Note that the *sum of squares error*, SS_{Error}, is also known as the sum of the variability within the groups, and that the *sum of squares Treatment*, $SS_{\text{Treatment}}$, is also known as the variability between the groups. The decomposition (7.29) holds under both H_0 and H_1, and hence allows us to construct a test statistic. Intuitively, under H_0, both SS_{Error} and $SS_{\text{Treatment}}$ should contribute to SS_{Total} in the same manner (once properly normalized). Alternatively, under H_1 it is expected that $SS_{\text{Treatment}}$ would contribute more heavily to the total variability.

Before proceeding with the construction of a test statistic, we present Listing 7.9, where the decomposition of (7.29) is demonstrated for the purpose of showing how to compute its individual components in Julia. Note that this verification of the decomposition is not something one would normally carry out in practice as it is already proven in (7.28).

Listing 7.9: Decomposing the sum of squares

```
1  using Random, Statistics
2  Random.seed!(1)
3  allData = [rand(24), rand(15), rand(73)]
4
5  xBarArray = mean.(allData)
6  nArray = length.(allData)
7  xBarTotal = mean(vcat(allData...))
8  L = length(nArray)
9
10 ssBetween=sum([nArray[i]*(xBarArray[i] - xBarTotal)^2 for i in 1:L])
11 ssWithin=sum([sum([(ob - xBarArray[i])^2 for ob in allData[i]]) for i in 1:L])
12 ssTotal=sum([sum([(ob - xBarTotal)^2 for ob in allData[i]]) for i in 1:L])
13
14 println("Sum of squares between groups: ", ssBetween)
15 println("Sum of squares within groups: ", ssWithin)
16 println("Sum of squares total: ", ssTotal)
```

```
Sum of squares between groups: 0.2941847110381936
Sum of squares within groups: 8.50335257006105
Sum of squares total: 8.797537281099242
```

The data is generated in line 3 in an array of arrays, `allData`. In line 5 the mean of each treatment is calculated via the `mean()` function with the broadcast operator '`.`', which performs the operation over each of the three elements of `allData`. In line 6 we retrieve the length of each array via the `length()` function and the broadcast operator. In line 7 the point estimate for the total population mean is calculated and stored as `xBarTotal`. Note that in contrast to line 5, here we first vertically concatenate all the groups into a single array. This is done via the `vcat()` function, and the splat operator `...`. In line 12, the number of treatments is stored as `L`. In line 10, $SS_{\text{Treatment}}$ is calculated. A comprehension is used, and the point estimate of the population mean `xbarTotal` is subtracted from the ith element of each array, and the results squared. These are each multiplied by the length of their respective arrays, and the results for each of the arrays summed together, and stored as `ssBetween`. Note that 'between' is sometimes used as an alternative name to 'treatments'. In line 11, SS_{Error} is calculated. The inner comprehension is used to square the difference between each observation, `ob`, and the group mean `xBarArray[i]`. The outer comprehension is used to repeat this process from the `1:L` th group. The results for all groups are summed. In line 12, SS_{Total} is calculated via (7.27). The difference between each observation, `ob`, and the point estimate for the population mean, `xBarTotal`, is calculated and each result squared. This is first performed for the i'th array, in the inner comprehension, and then repeated for all arrays via the outer comprehension. Finally all the squares are summed, via the outer `sum()` function.

Carrying out ANOVA

Having understood the sum of squares decomposition we now present the F-statistic of ANOVA:

$$F = \frac{SS_{\text{Treatment}}/(L-1)}{SS_{\text{Error}}/(m-L)}. \tag{7.30}$$

It is a ratio of the two sum of squares components of (7.29) normalized by their respective *degrees of freedom*, $L-1$ and $m-L$. These normalized quantities are respectively denoted by $MS_{\text{Treatment}}$ and MS_{Error} standing for 'Mean Squared'. Hence $F = MS_{\text{Treatment}}/MS_{\text{Error}}$.

Under H_0 and with the model assumptions presented in (7.26), the ratio F follows an F-distribution (first introduced in Section 5.2) with $L-1$ degrees of freedom for the numerator and $m-L$ degrees of freedom for the denominator. Intuitively, under H_0 we expect the numerator and denominator to have similar values, and hence expect F to be around 1 (indeed most of the mass of F distributions is concentrated around 1). However, if $MS_{\text{Treatment}}$ is significantly larger, then it indicates that H_0 may not hold. Hence the approach of the F-test is to reject H_0 if the F-statistic is greater than the $1-\alpha$ quantile of the respective F-distribution. Similarly, the p-value for an observed F-statistic f_o, is given by,

$$p = \mathbb{P}(F_{L-1,m-L} > f_o).$$

where $F_{L-1,m-L}$ is an F-distributed random variable with $L-1$ numerator degrees of freedom and $m-L$ denominator degrees of freedom.

It is often customary to summarize both the intermediate and final results of an ANOVA F-test in an *ANOVA table* as shown in Table 7.1 below, where 'T' and 'E' are shorthand for 'Treatments' and 'Error' respectively. Such tables also generalize to more complex ANOVA procedures not covered here.

Source of variance:	DOF:	Sum of sq's:	Mean sum of sq's:	F-value:
Treatments (between treatments)	$L-1$	SS_T	$MS_T = \dfrac{SS_T}{L-1}$	$\dfrac{MS_T}{MS_E}$
Error (within treatments)	$m-L$	SS_E	$MS_E = \dfrac{SS_E}{m-L}$	
Total	$m-1$	SS_{Total}		

Table 7.1: A one-way ANOVA table.

We now return to the three machines example and carry out a one-way ANOVA F-test. This is carried out in Listing 7.10 where we implement two alternative functions for ANOVA. The first function, `manualANOVA()`, extends the sum of squares code presented in Listing 7.9 above. The second function, `glmANOVA()`, utilizes the `GLM` package that is described in detail in Chapter 8. Note that `GLM` requires the `DataFrames` package. Both implementations yield identical results, returning a tuple of the F-statistic and the associated p-value. In this example, the p-value is very small and hence under any reasonable α we would reject H_0 and conclude that there is sufficient evidence that the diameter of the pipe depends on the type of machine used. Related is Listing 1.17 in Chapter 1 where we carry out ANOVA for the same data using the R language.

Listing 7.10: Executing one-way ANOVA

```
using Distributions, Plots; pyplot()

function anovaFStat(allData)
    xBarArray = mean.(allData)
    nArray = length.(allData)
    xBarTotal = mean(vcat(allData...))
    L = length(nArray)

    ssBetween = sum( [nArray[i]*(xBarArray[i] - xBarTotal)^2 for i in 1:L] )
    ssWithin = sum([sum([(ob - xBarArray[i])^2 for ob in allData[i]])
                                for i in 1:L])
    return (ssBetween/(L-1))/(ssWithin/(sum(nArray)-L))
end

case1 = [13.4, 13.4, 13.4, 13.4, 13.4]
case2 = [12.7, 11.8, 13.4, 12.7, 12.9]
stdDevs = [2, 2, 2, 2, 2]
numObs = [24, 15, 13, 23, 9]
L = length(case1)

N = 10^5

mcFstatsH0 = Array{Float64}(undef, N)
for i in 1:N
    mcFstatsH0[i] = anovaFStat([ rand(Normal(case1[j],stdDevs[j]),numObs[j])
                for j in 1:L ])
end

mcFstatsH1 = Array{Float64}(undef, N)
for i in 1:N
    mcFstatsH1[i] = anovaFStat([ rand(Normal(case2[j],stdDevs[j]),numObs[j])
                for j in 1:L ])
end

stephist(mcFstatsH0, bins=100,
        c=:blue, normed=true, label="Equal group means case")
stephist!(mcFstatsH1, bins=100,
        c=:red, normed=true, label="Unequal group means case")

dfBetween = L - 1
dfError = sum(numObs) - 1
xGrid = 0:0.01:10
plot!(xGrid, pdf.(FDist(dfBetween, dfError),xGrid),
            c=:black, label="F-distribution analytic")
critVal = quantile(FDist(dfBetween, dfError),0.95)
plot!([critVal, critVal],[0,0.8],
        c=:black, ls=:dash, label="Critical value boundary",
        xlims=(0,10), ylims=(0,0.8), xlabel="F-value", ylabel="Density")
```

```
Manual ANOVA: (10.516968568709117, 0.00014236168817139249)
GLM ANOVA: (10.516968568708988, 0.0001)
```

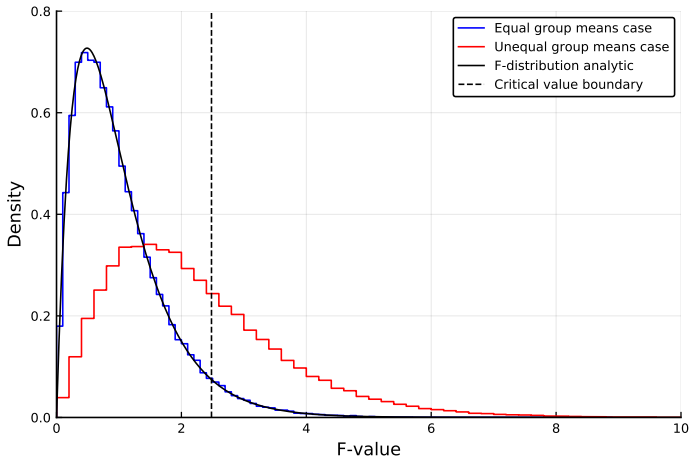

Figure 7.4: Histograms of the F-statistic for the case of equal group means (with analytic F-distribution), and not all equal group means.

In lines 7-25 the function `manualANOVA()` is implemented, which calculates the sum of squares in the same manner as in Listing 7.9. The sums of squares are normalized by their corresponding degrees of freedom `dfBetween` and `dfError`, and then in line 22 the F-statistic `fStat` is calculated. The p-value is then calculated in line 23 via the `ccdf()` function and the F-distribution `FDist()` with the degrees of freedom calculated above. The function returns a tuple of values, comprising the F-statistic and the corresponding p-value. In lines 27-38 the function `glmANOVA()` is defined. This function calculates the F-statistic and p-value, via functionality of the `GLM` package which is heavily discussed in Chapter 8. In lines 30-32 a `DataFrame` (see Chapter 4) is set-up in the manner required by the `GLM` package. Then in lines 34-35 two 'model objects' are created via the `lm()` function from the `GLM` package. Note that `modelH0` is constructed on the assumption that the machine type has no effect on the response, while `modelH1` is constructed on the assumption that treatment has an effect. Finally, the `ftest()` function from the `GLM` package is used to compare if `modelH1a` fits the data 'better' than `modelH0`. Also note that the `model` fields of the model objects are used. Finally, the F-statistic and p-value are returned in line 37. The results of both functions are printed in lines 40-41 and it can be observed that the F-statistics and p-values calculated are identical to within the numerical error expected due to the different implementations.

More on the Distribution of the F-Statistic

Having explored the basics of ANOVA, we now use Monte Carlo simulation to illustrate that under H_0 the F-statistic is indeed distributed according to the F-distribution. In Listing 7.11 below, we present an example where Monte Carlo simulation is used to empirically generate the distribution of the F-statistic for two different cases where the number of groups is $L = 5$. In the first case, the means of each group are all the same and are at 13.4, but in the second case, the means are not all the same. The first case represents H_0, while the latter is one possibility within H_1. For both cases the standard deviation of each group is identical (2).

In this example, for each of the two cases, N sample runs are generated, where each run consists of a separate random collection of sample observations for each group. Hence by using a large number of sample runs N, histograms can be used to empirically represent the theoretical distributions of

the F-statistics for both cases. The results presented in Figure 7.4, show that the distribution of the F-statistics for the equal group means case is in agreement with the analytically expected F-distribution, while the F-statistic for the case of unequal group means is not. The figure also illustrates the critical value for rejection with $\alpha = 0.05$. The area under the red curve to the left of that boundary is the power of the test under the specific point in H_1 that is simulated.

Listing 7.11: Monte Carlo based distributions of the ANOVA F-statistic

```
using Distributions, Plots; pyplot()

function anovaFStat(allData)
    xBarArray = mean.(allData)
    nArray = length.(allData)
    xBarTotal = mean(vcat(allData...))
    L = length(nArray)

    ssBetween = sum( [nArray[i]*(xBarArray[i] - xBarTotal)^2 for i in 1:L] )
    ssWithin = sum([sum([(ob - xBarArray[i])^2 for ob in allData[i]])
                                    for i in 1:L])
    return (ssBetween/(L-1))/(ssWithin/(sum(nArray)-L))
end

case1 = [13.4, 13.4, 13.4, 13.4, 13.4]
case2 = [12.7, 11.8, 13.4, 12.7, 12.9]
stdDevs = [2, 2, 2, 2, 2]
numObs = [24, 15, 13, 23, 9]
L = length(case1)

N = 10^5

mcFstatsH0 = Array{Float64}(undef, N)
for i in 1:N
    mcFstatsH0[i] = anovaFStat([ rand(Normal(case1[j],stdDevs[j]),numObs[j])
                for j in 1:L ])
end

mcFstatsH1 = Array{Float64}(undef, N)
for i in 1:N
    mcFstatsH1[i] = anovaFStat([ rand(Normal(case2[j],stdDevs[j]),numObs[j])
                for j in 1:L ])
end

stephist(mcFstatsH0, bins=100,
        c=:blue, normed=true, label="Equal group means case")
stephist!(mcFstatsH1, bins=100,
        c=:red, normed=true, label="Unequal group means case")

dfBetween = L - 1
dfError = sum(numObs) - 1
xGrid = 0:0.01:10
plot!(xGrid, pdf.(FDist(dfBetween, dfError),xGrid),
            c=:black, label="F-statistic analytic")
critVal = quantile(FDist(dfBetween, dfError),0.95)
plot!([critVal, critVal],[0,0.8],
        c=:black, ls=:dash, label="Critical value boundary",
        xlims=(0,10), ylims=(0,0.8), xlabel="F-value", ylabel="Density")
```

In lines 3-13 we create the function `anovaFStat()`, which takes an array of arrays as input, calculates the sums of squares and mean sums of squares as per Table 7.1, and returns the F-statistic of the data. It is similar to Listing 7.9. In lines 15 and 16 we create two arrays where `case1` represents an array of means for the case of all means being equal, and `case2`, represents an array of means for the case of all means not equal. In line 17 we crate the array of group standard deviations, `stdDevs`. Note that in both cases of equal group means and unequal group means, the standard deviations of all the groups are equal as per the model assumption. In line 18 we create the array `numObs`, where each element represents the number of observations of the i^{th} group, or level. In line 21 we specify the total number of Monte Carlo runs to be performed, `N`. In line 23 we preallocate the array `mcFStatsH0`, which will store `N` Monte Carlo generated F-statistics, for the case of all group means equal. In lines 24-27, we use a loop to generate `N` F-statistics via the `anovaFStat()` function defined earlier. We use the `rand()` and `Normal()` functions within a comprehension to generate data for each of the sample groups, using the group means, standard deviations, and number of observations. The comprehension generates an array of arrays, where each of the five elements of the outermost array is another array containing the observations for that group, 1 to5. This array of arrays is then used as the argument for `anovaFStat()`, which carries out a one-way ANOVA test on the data and outputs the corresponding F-value. Lines 29-33 are similar, using the `case2` means. In lines 35-38 histograms of the F-statistics are generated. In lines 40-41 the degrees of freedom of the treatments `dfBetween`, and the degrees of freedom of the error `dfError` are calculated. In lines 43-44 the analytic PDF of the is plotted. In line 45 the `quantile()` function is used to calculate the 95^{th} quantile of the F-distribution, `FDist()` and this is then used in lines 46-48 to plot the critical value.

Extensions

We have only touched on the very basics of ANOVA via the one-way ANOVA case. This stands at the basis of *experimental design*. However, there are many more aspects to ANOVA, and related ideas that one can explore. These include, but are not limited to:

- Extensions to *two-way ANOVA* where there are two treatment categories, for example 'machine type' and 'type of lubricant used in the machine', each having multiple treatments.
- Higher dimensional extensions, which are often considered in *block factorial design*.
- Comparison of individual factors to determine which specific treatments have an effect and in which way.
- Using ANOVA for *longitudinal data analysis* using *repeated measures*.
- Aspects of optimal experimental design.

These and many more aspects can be found in design and analysis of experiment texts such as [Mon17]. At the time of writing, many such procedures are not implemented directly in Julia. However, one alternative is the R software package, which contains many different implementations of these ANOVA extensions, among others. One can call these R packages directly from Julia as in Listing 1.17 of Chapter 1.

7.4 Independence and Goodness of Fit

We now consider a different group of hypothesis tests and associated procedures that deal with *checking for independence* and more generally checking *goodness of fit*. One question often posed is: *Does the population follow a specific distributional form?* We may hypothesize that the distribution is normal, exponential, Poisson, or that it follows any other form (see Chapter 3 for an extensive survey of probability distributions). Checking such a hypothesis is loosely called goodness of fit. Furthermore, in the case of observations over multiple dimensions, we may hypothesize that the different dimensions are independent. Checking for such independence is similar to the goodness of fit check.

In order to test for goodness of fit against some hypothesized distribution F_0, we setup the hypothesis test as,

$$H_0 : \ X \sim F_0, \qquad \text{vs.} \qquad H_1 : \text{otherwise.} \tag{7.31}$$

Here X denotes an arbitrary random variable from the population. In this case, we consider the parameter space associated with the test as the space of all probability distributions. The hypothesis formulation then partitions this space into $\{F_0\}$ (for H_0) and all other distributions in H_1.

For the independence case, assume for simplicity that X is a vector of two random variables, say $X = (X_1, X_2)$. Then for this case the hypothesis test setup would be,

$$H_0 : \ X_1 \text{ independent of } X_2, \qquad \text{vs.} \qquad H_1 : \ X_1 \text{ not independent of } X_2. \tag{7.32}$$

This sets the space of H_0 as the space of all distributions of independent random variable pairs, and H_1 as the complement.

To handle hypotheses such as (7.31) and (7.32) we introduce two different test procedures, the *Chi-squared test* and the *Kolmogorov-Smirnov test*. The Chi-squared test is used for goodness of fit of discrete distributions and for checking independence, while the Kolmogorov-Smirnov test is used for goodness of fit for arbitrary distributions based on the empirical cumulative distribution function. Before we dive into the individual test examples, we explain how to construct the corresponding test statistics.

In the Chi-squared case, the approach involves looking at counts of observations that match disjoint categories $i = 1, \ldots, M$. For each category i, we denote O_i as the number of observations that match that category. In addition, for each category there is also an expected number of observations under H_0, which we denote as E_i. With these, one can express the test statistic as,

$$\chi^2 = \sum_{i=1}^{M} \frac{(O_i - E_i)^2}{E_i}. \tag{7.33}$$

Notice that under H_0 of (7.31), we expect that for each category i, both O_i and E_i will be relatively close, and hence it is expected that the sum of relative squared differences, χ^2, will not be too big. Conversely, a large value of χ^2 may indicate that H_0 is not plausible. Later in this section, we show how to use χ^2 to construct the test to check for both goodness of fit (7.31), and to check for independence (7.32).

In the case of Kolmogorov-Smirnov, a key aspect is the empirical cumulative distribution function (ECDF), which was introduced in Section 4.3. Recall that for a sample of observations, $x_1, \ldots, x_n$

the ECDF is,

$$\hat{F}(x) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}\{x_i \leqslant x\}, \qquad \text{where } \mathbf{1}\{\cdot\} \text{ is the indicator function.} \tag{7.34}$$

The approach of Kolmogorov-Smirnov test is to check the closeness of the ECDF to the CDF hypothesized under H_0 in (7.31). This is done via the *Kolmogorov-Smirnov statistic*,

$$\tilde{S} = \sup_x |\hat{F}(x) - F_0(x)|, \tag{7.35}$$

where $F_0(\cdot)$ is the CDF under H_0 and sup is the supremum over all possible x values. Similar to the case of Chi-squared, under H_0 it is expected that $\hat{F}(\cdot)$ does not deviate greatly from $F_0(\cdot)$, and hence it is expected that $\tilde{S}$ is not very large.

The key to both the Chi-Squared and Kolmogorov-Smirnov tests is that under H_0 there are tractable known approximations to the distribution of the test statistics of both (7.33) and (7.35)). These approximations allow us to obtain an approximate p-value in the standard way via,

$$p = \mathbb{P}(W > u), \tag{7.36}$$

where W denotes a random variable distributed according to the approximate distribution and u is the observed test statistic of either (7.33) or (7.35). We now elaborate on the details.

Chi-squared Test for Goodness of Fit

Consider the hypothesis (7.31) and assume that the distribution F_0 can be partitioned into categories $i = 1, \ldots, M$. Such a partition naturally occurs when the distribution is discrete with a finite number of outcomes. It can also be artificially introduced in other cases. With such a partition, having n sample observations, we denote by E_i the expected number of observations satisfying category i. These values are theoretically computed. Then, based on observations $x_1, \ldots, x_n$, we denote by O_i as the number of observations that satisfy category i. Note that,

$$\sum_{i=1}^{M} E_i = n, \qquad \text{and} \qquad \sum_{i=1}^{M} O_i = n.$$

Now, based on $\{E_i\}$ and $\{O_i\}$, we can compute the χ^2 test statistic (7.33).

It turns out that under H_0, the χ^2 test statistic of (7.33) approximately follows a Chi-squared distribution with $M - 1$ degrees of freedom. Hence this allows us to approximate the p-value via (7.36), where W is taken as such a Chi-squared random variable and u as the test statistic. This is also sometimes called *Pearson's chi-squared test*.

We now present an example where we assume under H_0 that a die is biased, with the probabilities for each side (1 to 6) given by the following vector $\mathbf{p}$,

$$\mathbf{p} = (0.08, \quad 0.12, \quad 0.2, \quad 0.2, \quad 0.15, \quad 0.25). \tag{7.37}$$

Note that if there are then n observations, we have that $E_i = n\, p_i$. For this example $n = 60$, and hence the vector of expected values for each side is,

$$\mathbf{E} = (4.8, \quad 7.2, \quad 12, \quad 12, \quad 9, \quad 15).$$

Now imagine that the die is rolled $n = 60$ times, and the following count of outcomes (1 to 6) is observed,

$$\mathbf{O} = (3, \quad 2, \quad 9, \quad 11, \quad 8, \quad 27).$$

In Listing 7.12 below, we use this data to compute the test statistic and p-value. This is done first manually, and then the `ChisqTest()` function from the `HypothesisTests` package is used. From the output we see that the p-value is around 0.0105. Hence at the $\alpha = 0.05$ level we would reject H_0 and conclude the distribution does not follow (7.37). That is we would conclude there is sufficient evidence to believe the die is weighted differently to the weights $\mathbf{p}$. However at $\alpha = 0.01$ we will fail to reject H_0.

Listing 7.12: Chi-squared test for goodness of fit

```
using Distributions, HypothesisTests

p = [0.08, 0.12, 0.2, 0.2, 0.15, 0.25]
O = [3, 2, 9, 11, 8, 27]
M = length(O)
n = sum(O)
E = n*p

testStatistic = sum((O-E).^2 ./E)
pVal = ccdf(Chisq(M-1), testStatistic)

println("Manually calculated test statistic: ", testStatistic)
println("Manually calculated p-value: ", pVal,"\n")

println(ChisqTest(O,p))
```

```
Manually calculated test statistic: 14.974999999999998
Manually calculated p-value: 0.010469694843220351

Pearson's Chi-square Test
-------------------------
Population details:
    parameter of interest:   Multinomial Probabilities
    value under h_0:         [0.08, 0.12, 0.2, 0.2, 0.15, 0.25]
    point estimate:          [0.05, 0.0333333, 0.15, 0.183333, 0.133333, 0.45]
    95% confidence interval: Tuple{Float64,Float64}[(0.0, 0.1828), (0.0, 0.1662),
        (0.0333, 0.2828), (0.0667, 0.3162), (0.0167, 0.2662), (0.3333, 0.5828)]

Test summary:
    outcome with 95% confidence: reject h_0
    one-sided p-value:           0.0105

Details:
    Sample size:        60
    statistic:          14.975000000000001
    degrees of freedom: 5
    residuals:          [-0.821584, -1.93793, -0.866025, -0.288675, -0.333333, 3.09839]
    std. residuals:     [-0.85656, -2.06584, -0.968246, -0.322749, -0.361551, 3.57771]
```

In line 3 the array `p` is created, which represents the probabilities of each side occurring under H_0. In line 4 the array `O` is created, which contains the frequencies, or counts, of each side outcome observed. In line 5 the total number of categories (or side outcomes) is stored as `M`. In line 6 the total number of observations is stored as `n`. In line 7, the array of expected number of observed outcomes for each side is calculated by multiplying the vector of expected probabilities under H_0 by the total number of observations `n`. The resulting array is stored as `E`. In line 9 (7.33) is used to calculate the Chi-squared test statistic. In line 10 the test statistic is used to calculate the p-value. Since under the null hypothesis the test statistic is asymptotically distributed according to a chi-squared distribution, the `ccdf()` function is used on a `Chisq()` distribution with `M-1` degrees of freedom. In lines 12 and 13 the manually calculated test statistic and p-value are printed. In line 15 the `ChisqTest()` function from the `HypothesisTests` package is used to perform the chi-squared test on the frequency data in array `p`.

Chi-squared Test Used to Check Independence

We now show how a Chi-squared statistic can use used to check for independence, as in (7.32). Consider an example where 373 individuals are categorized as Male/Female, and Smoker/Non-smoker, as in the following *contingency table*.

	Smoker	Non-smoker
Male	18	132
Female	45	178

In this example, 18 individuals were recorded as 'male' and 'smoker', and so forth. Now under H_0, we assume that the smoking or non-smoking behavior of the individual is independent of the gender (male or female). To check for this using a Chi-squared statistic, we first setup $\{E_i\}$ and $\{O_i\}$ as in the following table.

	Smoker	Non-smoker	Total/proportion
Male	$O_{11} = 18$ $E_{11} = 25.34$	$O_{12} = 132$ $E_{12} = 124.67$	150 / 0.402
Female	$O_{21} = 45$ $E_{21} = 37.66$	$O_{22} = 178$ $E_{22} = 185.33$	223 / 0.598
Total/Proportion	63/0.169	310/0.831	373 / 1

Table 7.2: The elements $\{O_{ij}\}$ and $\{E_{ij}\}$ as in the contingency table.

Here the observed *marginal distribution* over male vs. female is based on the proportions $\mathbf{p} = (0.402, 0.598)$ and the distribution over smoking vs. non-smoking is based on the proportions $\mathbf{q} = (0.169, 0.831)$. Then, since independence is assumed under H_0, we multiply the marginal probabilities to obtain the expected observation counts,

$$E_{ij} = n\ p_i\ q_j. \tag{7.38}$$

For example, $E_{21} = 373 \times 0.169 \times 0.598 = 37.66$. Now with these values at hand, the Chi-squared test statistic can be setup as follows,

$$\chi^2 = \sum_{i=1}^{m} \sum_{j=1}^{\ell} \frac{(O_{ij} - E_{ij})^2}{E_{ij}}, \tag{7.39}$$

where m and ℓ are the respective dimensions of the contingency table ($m = \ell = 2$ in this example).

It turns out that under H_0 the test statistic (7.39) is approximately Chi-squared distributed with $(m-1) \times (\ell - 1)$ degrees of freedom. This implies 1 degree of freedom in our example. Hence (7.36) can be used to determine an (approximate) p-value for this test, just like in the previous example.

Listing 7.13 carries out a Chi-squared test in order to check if there is a relationship between gender and smoking. In this example, since the p-value is 0.0387, we conclude by saying there is some evidence that there is a relationship. That is, if $\alpha = 0.05$ we conclude that there is sufficient evidence to reject H_0. However if $\alpha = 0.01$ we conclude that there is insufficient evidence to reject H_0.

Listing 7.13: Chi-squared for checking independence

```
1  using Distributions
2
3  xObs     = [18 132; 45 178]
4  rowSums  = [sum(xObs[i,:]) for i in 1:2]
5  colSums  = [sum(xObs[:,i]) for i in 1:2]
6  n        = sum(xObs)
7
8  rowProps = rowSums/n
9  colProps = colSums/n
10
11 xExpect  = [colProps[c]*rowProps[r]*n for r in 1:2, c in 1:2]
12 testStat = sum([(xObs[r,c]-xExpect[r,c])^2 / xExpect[r,c] for r in 1:2,c in 1:2])
13 pVal = ccdf(Chisq(1),testStat)
14
15 println("Chi-squared value: ", testStat)
16 println("P-value: ", pVal)
```

```
Chi-squared value: 4.274080056208799
P-value: 0.03869790606536347
```

In line 3 the observation counts from the contingency table are stored as the 2-dimensional array, `xObs`. In line 4 the observations in each row are summed via `xObs[i,:]`, and the use of a comprehension. In line 5 the observations in each column are calculated via a similar approach to that in line 4 above. In line 6 the total number of observations is stored as `n`. In line 8 and 9 the row and column proportions are calculated. In line 11 the expected number of observations, $\{E_{ij}\}$ (shown in Table 7.2), are calculated. Note the use of the comprehension, which calculates (7.38) for each combination of sex and smoker/non-smoker. In line 12 the test statistic is calculated via (7.39) through the use of a comprehension. In line 13 the test statistic is used to calculate the p-value. Since under the null hypothesis the test statistic is asymptotically distributed according to a Chi-squared distribution, the `ccdf()` function is used on a `Chisq()` distribution with $(m-1) \times (\ell - 1)$ degrees of freedom, i.e. 1 in this example.

Kolmogorov-Smirnov Test

We now depart from the situations of a finite number of categories as in the Chi-squared test, and consider the Kolmogorov-Smirnov test, which is based on the test statistic $\tilde{S}$ from (7.35). The approach is based on the fact that, under H_0 of (7.31), the empirical cumulative distribution (ECDF) $\hat{F}(\cdot)$ is close to the actual CDF $F_0(\cdot)$. To get a feel for this notice that for every value $x \in \mathbb{R}$, the ECDF at that value, $\hat{F}(x)$, is the proportion of the number of observations less than or equal to x. Under H_0, multiplying the ECDF by n yields a binomial random variable with success probability, $F_0(x)$:

$$n\,\hat{F}(x) \quad \sim \quad \text{Bin}\big(n, F_0(x)\big).$$

Hence,

$$\mathbb{E}[\hat{F}(x)] = F_0(x), \qquad \text{Var}\big(\hat{F}(x)\big) = \frac{F_0(x)\big(1 - F_0(x)\big)}{n}.$$

See the Binomial distribution in Section 3.5. Hence, for non-small n, the ECDF and CDF should be close since the variance for every value x is of the order of $1/n$ and diminishes as n grows. The formal statement of this is, taking $n \to \infty$ and considering all values of x simultaneously is known as the *Glivenko Cantelli Theorem*.

For finite n, the ECDF will not exactly align with the CDF. However the Kolmogorov-Smirnov test statistic (7.35) is useful when it comes to measuring this deviation. This is due to the fact that under H_0 the *stochastic process* in the variable, x,

$$\sqrt{n}\big(\hat{F}(x) - F_0(x)\big) \tag{7.40}$$

is approximately identical in probability law to a standard *Brownian Bridge*, $B(\cdot)$, composed with $F_0(x)$. That is, by denoting $\hat{F}_n(\cdot)$ as the ECDF with n observations, we have that

$$\sqrt{n}\big(\hat{F}_n(x) - F_0(x)\big) \quad \overset{d}{\approx} \quad B\big(F_0(x)\big), \tag{7.41}$$

which asymptotically converges to equality in distribution as $n \to \infty$. Note that a Brownian Bridge, $B(t)$, is a form of a variant of *Brownian Motion*, constrained to equal 0 both at $t = 0$ and $t = 1$. It is a type of *diffusion process*. See [Kle12] for a good introduction to diffusion processes and *stochastic calculus*.

Now consider the supremum as in the Kolmogorov-Smirnov test statistic $\tilde{S}$, as defined in (7.35). It can be shown that, in cases where $F_0(\cdot)$ is a continuous function (distribution), as $n \to \infty$,

$$\sqrt{n}\tilde{S} \quad \overset{d}{=} \quad \sup_{t\in[0,1]} |B(t)|. \tag{7.42}$$

Importantly, notice that the right hand side does not depend on $F_0(\cdot)$, but rather is the maximal value attained by the absolute value of the Brownian bridge process over the interval $[0, 1]$. It then turns out that (see for example [Man07] for a derivation) such a random variable, denoted by K, has CDF,

$$F_K(x) = \mathbb{P}\bigg(\sup_{t\in[0,1]} |B(t)| \le x\bigg) = 1 - 2\sum_{k=1}^{\infty}(-1)^{k-1}e^{-2k^2x^2} = \frac{\sqrt{2\pi}}{x}\sum_{k=1}^{\infty} e^{-(2k-1)^2\pi^2/(8x^2)}. \tag{7.43}$$

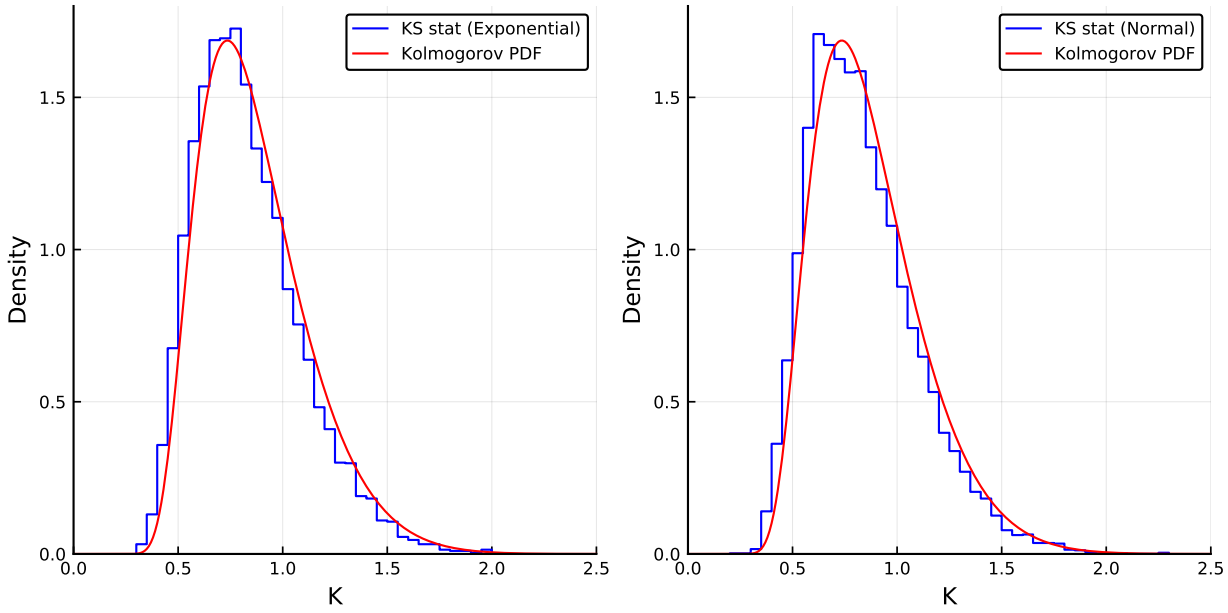

Figure 7.5: PDF of the Kolmogorov distribution, alongside histograms of K-S test statistics from normal and exponential populations.

This is sometimes called the *Kolmogorov distribution*. Thus to obtain an approximate p-value for the Kolmogorov-Smirnov test using (7.36) we calculate,

$$p = 1 - F_K(\sqrt{n}\tilde{S}). \tag{7.44}$$

Figure 7.5 generated by Listing 7.14 compares the PDF of the Kolmogorov distribution to the empirical distribution of $\tilde{S}$ scaled by $\sqrt{n}$ as on the left hand side of (7.42). This is done for two different scenarios. In the first data is sampled from an exponential distribution, and in the second, data sampled from a normal distribution. As illustrated in the resulting Figure 7.5, the distributions of the Monte Carlo generated test statistics are in close agreement with the analytic PDF, regardless of what underlying distribution $F_0(x)$ the data comes from.

Listing 7.14: Comparisons of distributions of the K-S test statistic

```
1  using Distributions, StatsBase, HypothesisTests, Plots, Random; pyplot()
2  Random.seed!(0)
3
4  n = 25
5  N = 10^4
6  xGrid = -10:0.001:10
7  kGrid = 0:0.01:5
8  dist1, dist2 = Exponential(1), Normal()
9
10 function ksStat(dist)
11     data = rand(dist,n)
12     Fhat = ecdf(data)
13     sqrt(n)*maximum(abs.(Fhat.(xGrid) - cdf.(dist,xGrid)))
14 end
15
16 kStats1 = [ksStat(dist1) for _ in 1:N]
17 kStats2 = [ksStat(dist2) for _ in 1:N]
18
19 p1 = stephist(kStats1, bins=50,
20         c=:blue, label="KS stat (Exponential)", normed=true)
21 p1 = plot!(kGrid, pdf.(Kolmogorov(),kGrid),
22         c=:red, label="Kolmogorov PDF", xlabel="K", ylabel="Density")
23
24 p2 = stephist(kStats2, bins=50,
25         c=:blue, label="KS stat (Normal)", normed=true)
26 p2 = plot!(kGrid, pdf.(Kolmogorov(),kGrid),
27         c=:red, label="Kolmogorov PDF", xlabel="K", ylabel="Density")
28
29 plot(p1, p2, xlims=(0,2.5), ylims=(0,1.8), size=(800, 400))
```

In lines 4-8 we specify the sample size `n`, number of Monte Carlo repetitions `N`, grids for computation and plotting, and two distributions of the underlying population. In lines 10-14, the function `ksStat()` is created, which takes a distribution type as input, randomly samples `n` observations from it, calculates the ECDF of the data via the `ecdf()` function, and finally returns the left hand side of (7.42) by calculating the K-S test statistic via (7.35), and multiplying this by `sqrt(n)`. Note that in line 12, the `ecdf()` function returns a cdf function type itself, which is stored as `Fhat`, and broadcasted over `xGrid` in line 13. In lines 16-17, a comprehension is used along with the `ksStat()` function to generate `N` K-S test statistics for each distribution of the population. The remainder of the code compares histograms of `kStats1` and `kStats2` against PDFs of the `Kolmogorov()` distribution.

Now that we have demonstrated that the distribution of the scaled Kolmogorov-Smirnov statistic is similar to the distribution of K as in (7.43), we demonstrate how the Kolmogorov-Smirnov statistic can be used to carry out a goodness of fit test. For this example, consider that a series of observations has been made from some unknown underlying gamma distribution with shape parameter 2 and mean 5. The question we then wish to ask is: *given the sample observations, is the underlying distribution exponential with the same mean?* The answer is false because an exponential distribution is a gamma distribution with a shape parameter of 1, not 2. However with a finite number of observations, such as $n = 100$ in our case, we can only expect to give an approximate answer.

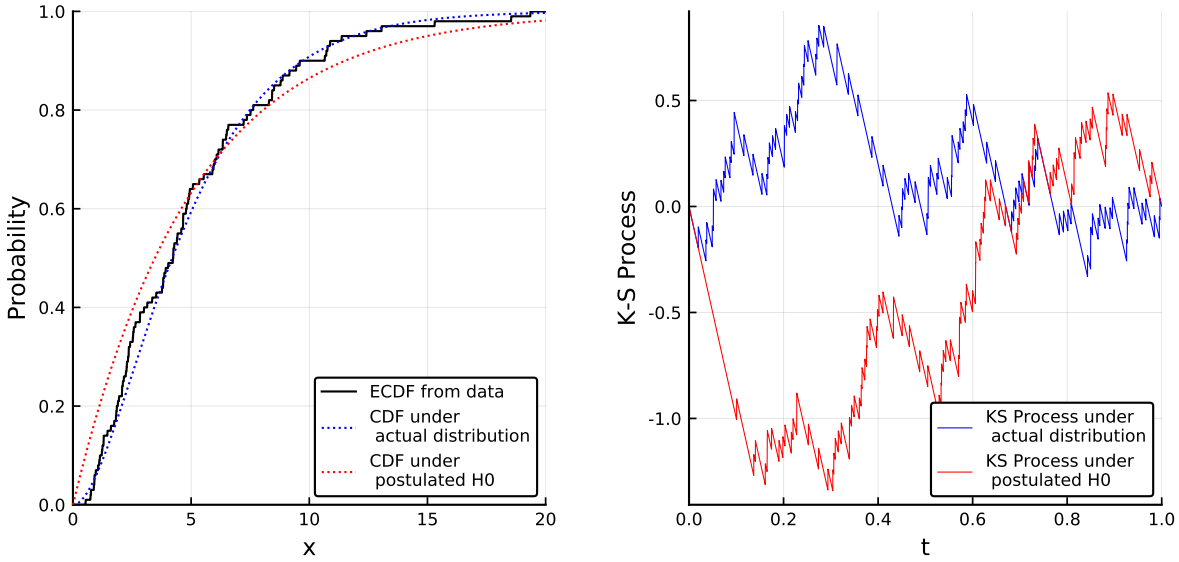

Figure 7.6: Left: CDFs and ECDF. Right: K-S processes scaled over $[0, 1]$.

To help illustrate the logic of the approach, see Figure 7.6 generated by Listing 7.15. The left plot presents the ECDF of the data plotted against the actual CDF (blue) as well as the postulated CDF (red). The ECDF follows the actual CDF quite closely but does not follow the postulated CDF well. Keep in mind that in a practical situation, we don't know the actual CDF. We only know the postulated CDF. Still we expect mild deviations under H_0, such that when composed with the CDF, behave approximately as a Brownian bridge (defined for $t \in [0, 1]$). In contrast, if irregular deviations appear we can conclude that H_0 does not hold. For this look at the right hand plot of Figure 7.6. The observed deviations (time stretched by the CDF) are in the red curve. Such a trajectory of a Brownian bridge is possible but not plausible. Instead, most trajectories will behave more like the blue curve.

Now the Kolmogorov-Smirnov statistic (7.35) is useful. Under H_0 (and for non-small n) it needs to follow a CDF as (7.43). Hence this CDF can be used to compute the p-value using (7.44). Listing 7.15 computes the p-value by manually calculating a truncation of the series in (7.43), by using the `Kolmogorov()` distribution object, and by using `ApproximateOneSampleKSTest()` from the `HypothesisTests` package. As can be observed from the output, the resulting p-values are all in agreement at approximately 0.0545. Observe now that if $\alpha = 0.05$ then we fail to reject H_0 because there isn't sufficient evidence that the distribution deviates from an exponential distribution. With this example (using the same random number generation seed), if you were to increase the number of observations to $n = 200$, then the p-value changes to 0.0004, meriting rejection under any sensible significance level.

Listing 7.15: ECDF, actual and postulated CDF's, and their differences

```
using Random,Distributions,StatsBase,Plots,HypothesisTests,Measures; pyplot()
Random.seed!(3)

dist = Gamma(2, 2.5)
distH0 = Exponential(5)
n = 100
data = rand(dist,n)

Fhat = ecdf(data)
diffF(dist, x) = sqrt(n)*(Fhat(x) - cdf.(dist,x))
xGrid = 0:0.001:30
ksStat = maximum(abs.(diffF(distH0, xGrid)))

M = 10^5
KScdf(x) = sqrt(2pi)/x*sum([exp(-(2k-1)^2*pi^2 ./(8x.^2)) for k in 1:M])

println("p-value calculated via series: ",
        1-KScdf(ksStat))
println("p-value calculated via Kolmogorov distribution: ",
        1-cdf(Kolmogorov(),ksStat),"\n")

println(ApproximateOneSampleKSTest(data,distH0))

p1 = plot(xGrid, Fhat(xGrid),
        c=:black, lw=1, label="ECDF from data")
p1 = plot!(xGrid, cdf.(dist,xGrid),
        c=:blue, ls=:dot, label="CDF under \n actual distribution")
p1 = plot!(xGrid, cdf.(distH0,xGrid),
        c=:red, ls=:dot, label="CDF under \n postulated H0",
        xlims=(0,20), ylims=(0,1), xlabel = "x", ylabel = "Probability")

p2= plot(cdf.(dist,xGrid), diffF(dist, xGrid),lw=0.5,
        c=:blue,        label="KS Process under \n actual distribution")
p2 = plot!(cdf.(distH0,xGrid), diffF(distH0, xGrid), lw=0.5,
        c=:red, xlims=(0,1), label="KS Process under \n postulated H0",
    xlabel = "t", ylabel = "K-S Process")

plot(p1, p2, legend=:bottomright, size=(800, 400), margin = 5mm)
```

```
p-value calculated via series: 0.05473084786694438
p-value calculated via Kolmogorov distribution: 0.054730847866944266

Approximate one sample Kolmogorov-Smirnov test
----------------------------------------------
Population details:
    parameter of interest:   Supremum of CDF differences
    value under h_0:         0.0
    point estimate:          0.13421930779083405

Test summary:
    outcome with 95% confidence: fail to reject h_0
    two-sided p-value:           0.0545

Details:
    number of observations:   100
    KS-statistic:             1.3421930779083404
```

In lines 4 and 5 we set the actual underlying distribution (gamma), and postulated distribution (exponential) respectively. In line 6 we set the number of observations, `n`. The data is generated in line 7. The ECDF is created in line 9 and the process (7.40) is defined in line 9 via our function `diffF()` allowing different postulated (H_0) distributions. The Kolmogorov-Smirnov statistic is calculated in line 12. Line 15 implements our function `KScdf()` by truncating the series in (7.43) to `M`. We use it, as well as the `Kolmogorov()` distribution object to print out p-values in lines 17-20. Simiarly in line 21 we use `ApproximateOneSampleKSTest()` from `HypothesisTests()`. The remainder of the code creates Figure 7.6 where a point to note are the horizontal-axis values in lines 32 and 34 reflecting the composition in (7.41).

7.5 More on Power

In this section the concept of power is covered in greater depth together with related aspects of hypothesis testing such as the distribution of the p-value. Recall that as first introduced in Section 5.6 and summarized in Table 5.1, the statistical *power* of a hypothesis test is the probability of correctly rejecting H_0 as is given by $1 - \mathbb{P}(\text{Type II error})$. We now reinforce this idea through a concrete introductory example.

Consider a normal population with unknown parameters μ and σ, and say that we wish to conduct a one-sided hypothesis test on the population mean using the following hypothesis test set-up,

$$H_0 : \mu = \mu_0 \qquad \text{and} \qquad H_1 : \mu > \mu_0. \tag{7.45}$$

Importantly, since power is the probability of a correct rejection, if the underlying (unknown) parameter μ varies greatly from the value under the null hypothesis μ_0, then the power of the test in this scenario is greater. Likewise, if the underlying parameter does not vary greatly from the value under the null hypothesis, the power of the test is lower. Similar affects occur due to the underlying (unknown) variance as well as the sample size. A lower variance implies higher power and larger sample sizes increase power. Also reducing (improving) α will decrease the power and vise-versa.

In Listing 7.16 below, several different scenarios, labelled **A**, **B**, **C**, and **D**, are considered. For each, N test statistics are calculated via Monte Carlo simulation for N sample groups and the power of the test is estimated. First the underlying mean equals the mean under the null hypothesis and hence the power equals α. Then in each subsequent scenario, the parameters or sample size are changed in a way that power is increased. First in scenario **A**, the underlying mean is increased, then in scenario **B** it is increased further. Further in scenario **C** the sample size is increased, and finally in scenario **D** the standard deviation is decreased. As you can observe from the output of Listing 7.16 each of these incremental changes increases the power up to approximately 0.91 in case **D**. In practice, if keeping α constant, it is only the sample size that can be controlled, however understanding the effect of the other parameters on the power is important in deciding how large samples should be.

For some of these scenarios Listing 7.16 employs kernel density estimation to plot the distribution of the test statistics. The resulting Figure 7.7 is similar to Figure 5.13, however in this case the focus is on power.. The power under different scenarios is given by the area under each PDF to the right of the critical value boundary. Hence the more 'separation' that we can achieve from the

distribution under H_0, the better. As a side point, note that the curves shown in Figure 7.7 could have alternatively been obtained analytically via the *non-central T-distribution*, a classical concept that we do not cover further.

Listing 7.16: Distributions under different hypotheses

```
using Random, Distributions, KernelDensity, Plots, LaTeXStrings; pyplot()
Random.seed!(1)

function tStat(mu0,mu,sig,n)
    sample = rand(Normal(mu,sig),n)
    xBar   = mean(sample)
    s      = std(sample)
    (xBar-mu0)/(s/sqrt(n))
end

mu0, mu1A, mu1B = 20, 22, 24
sig, n = 7, 5
N = 10^6
alpha = 0.05

dataH0  = [tStat(mu0,mu0,sig,n) for _ in 1:N]
dataH1A = [tStat(mu0,mu1A,sig,n) for _ in 1:N]
dataH1B = [tStat(mu0,mu1B,sig,n) for _ in 1:N]
dataH1C = [tStat(mu0,mu1B,sig,2*n) for _ in 1:N]
dataH1D = [tStat(mu0,mu1B,sig/2,2*n) for _ in 1:N]

tCrit = quantile(TDist(n-1),1-alpha)
estPwr(sample) = sum(sample .> tCrit)/N

println("Rejection boundary: ", tCrit)
println("Power under H0:  ", estPwr(dataH0))
println("Power under H1A: ", estPwr(dataH1A))
println("Power under H1B (mu's farther apart): ", estPwr(dataH1B))
println("Power under H1C (double sample size): ", estPwr(dataH1C))
println("Power under H1D (like H1C but std/2): ", estPwr(dataH1D))

kH0  = kde(dataH0)
kH1A = kde(dataH1A)
kH1D = kde(dataH1D)
xGrid = -10:0.1:15

plot(xGrid,pdf(kH0,xGrid),
        c=:blue, label="Distribution under H0")
plot!(xGrid,pdf(kH1A,xGrid),
        c=:red, label="Distribution under H1A")
plot!(xGrid,pdf(kH1D,xGrid),
        c=:green, label="Distribution under H1D")
plot!([tCrit,tCrit],[0,0.4],
        c=:black, ls=:dash, label="Critical value boundary",
        xlims=(-5,10), ylims=(0,0.4), xlabel=L"\Delta = \mu - \mu_0")
```

```
Rejection boundary: 2.131846786326649
Power under H0:  0.049598
Power under H1A: 0.134274
Power under H1B (mu's farther apart): 0.281904
Power under H1C (double sample size): 0.406385
Power under H1D (like H1C but std/2): 0.91554
```

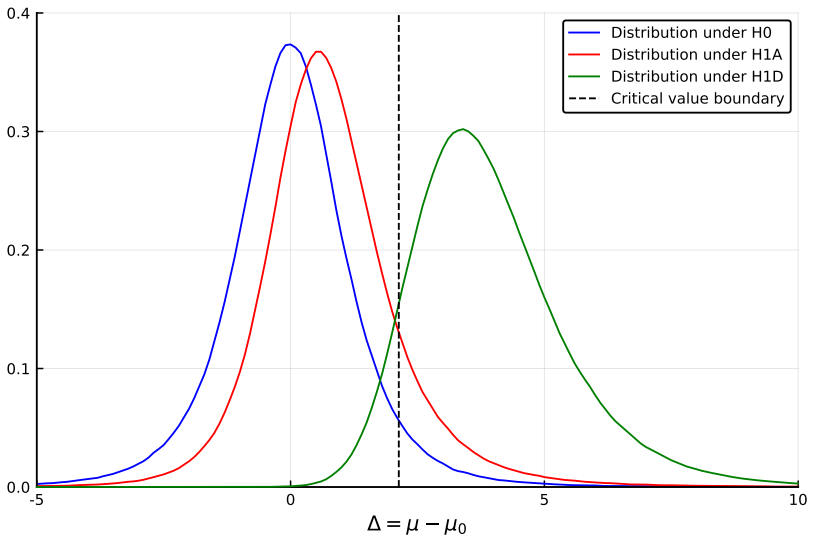

Figure 7.7: Numerically estimated distributions of the test statistic for various scenarios of values of the underlying parameter μ.

In lines 4-9 the function `tStat()` is defined, which returns the value of the test statistic for a randomly generated sample. Note here that `mu0` represent the value under the null hypothesis as used to calculate the test statistic while `mu` represents the value of the actual underlying mean, used to generate random samples. In line 11 we define different values of μ, matching H_0, scenario **A**, and scenario **B**. In line 12 the standard deviation and the number of observations is defined. In line 13 we define the number of Monte Carlo repetitions and in line 14 we define the significance level. In lines 16-20, the `tStat()` function is used along with a series of comprehensions to generate `N` test statistics for each of the scenarios as described above. In line 22, the critical value for the significance level `alpha` is calculated by using the `quantile()` function on a T-distribution `TDist()`, with `n-1` degrees of freedom. In line 23 the function `estPwr()` is defined, which takes an array of test statistics as input, and then approximates the corresponding power of the scenario as the proportion of statistics that exceeds `tCrit` calculated previously, i.e. the proportion of cases for which the null hypothesis was rejected. Note the use of the `.>` which returns an array of `true,false` values, which are then summed up and divided by `N`. Lines 25-30 print the output and estimate the power for each of the scenarios. Then lines 32-34 create `UnivariateKDE` objects representing kernel density estimates of the test statistics. These are then plotted in lines 37-42 using the `pdf()` function with a method used for kernel density estimates. Lines 43-45 plot the critical value.

Power Curves

From Listing 7.16, we can see that the statistical power of a hypothesis test can vary. It depends not only on the parameters of the test, such as the number of observations in the sample group n and the specified confidence level α, but also on the underlying parameter values μ and σ. Hence, a key aspect of *experimental design* involves determining the test parameters such that not only is the probability of a type I error controlled, but that the test is sufficiently powerful over a range of different scenarios. This is important, as in reality there are an infinite number of possibilities in H_1, any one of which could describe the underlying parameters. By designing a statistical test that has sufficient power, we aim to have confidence that if the underlying parameter deviates from

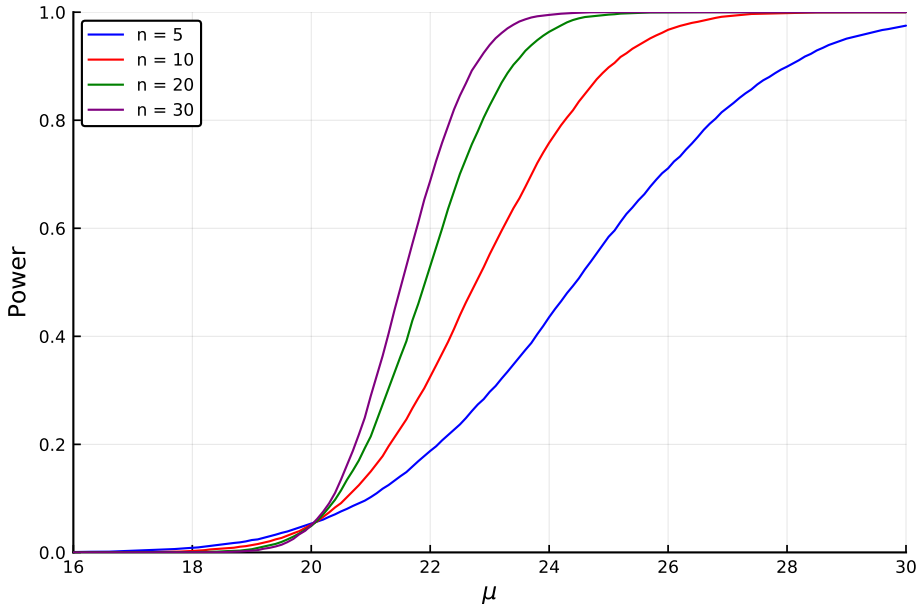

Figure 7.8: Power curves for the one-sided T-test with various sample sizes at $\alpha = 0.05$.

the null hypothesis, then this will be identified. Such planning is often aided by inspecting power curves.

A *power curve* is a plot of the power as a function of certain parameters. To illustrate this we continue with the hypothesis formulation (7.45). Listing 7.17 estimates the power of a one sided T-test. We estimate power of hypothesis test setup over a range of different values of μ, for various sample size of $n = 5, 10, 20, 30$. For each sample size, the power is estimated and the result is a power curve that can be plotted as in Figure 7.8.

Our focus is for $\mu > 20$. It can be seen that as the number of sample observations increases, the statistical power of the test increases. Similarly the large μ is (compared to $\mu_0 = 20$) the higher the power. Observe also that at $\mu = \mu_0$ the power is $\alpha = 0.05$. An interesting subtle point to note is that where $\mu < 20$, the ordering of the curves is reversed. For example, one can see that in this region the scenario where $n = 30$ has less power than that for $n = 5$ due to the fact that the probability of rejecting the null hypothesis at all is lower. Another point to note is that the x-axis could be adjusted to represent the difference between the value of mu_0 under the null hypothesis, and the various possible values of μ as was done in Figure 7.7. Furthermore one could make the axis scale invariant by dividing said difference by the standard deviation. Such curves are often seen in experimental design reference material.

With such a plot at hand, assume we are planning a costly (in the sample size) hypothesis test aiming to show that $\mu > 20$. Say that for sample size planning purposes, we have reason to believe that $\mu > 24$. Assume now that after fixing $\alpha = 0.05$ we wish to have power greater than 0.6. We then consider between the option of $n = 5$, $n = 10$, or $n = 20$ observations. Figure 7.7 illustrates that $n = 10$ is a sufficient number of observations. However, if we had plausible reason to believe that (under H_1) $\mu = 22$ then $n = 30$ observations are required to attain power > 0.6.