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

D R A F T

Yoni Nazarathy, Hayden Klok

August 14, 2020

Preface to this DRAFT version

This DRAFT version of our book includes a complete structure of the contents and a complete code-base using Julia v1.4 and almost 40 packages. We have settled on a total of 212 code examples and these are available at `https://github.com/h-Klok/StatsWithJuliaBook`. Note that with the exception of chapter introductions, the written 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 specific feedback that you have.

Yoni Nazarathy,
Hayden Klok
August, 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. 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, 4 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.

This book is about statistics, probability, data science, machine learning and artificial intelligence. By reading it you should be able to gain a basic understanding of the concepts that underpin these fields. However in contrast to books that focus on theory, this book is code example centric. Almost all of the concepts that we introduce are backed by illustrative code examples. Similarly almost all of the figures are generated via the code examples. 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. 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.

As you read this book, you can also run the code examples yourself. You may experiment by modifying parameters in the code examples or making any other modification that you can think of. With the exception of a few introductory examples, most of the code examples rarely focus on the Julia language directly but are rather meant to illustrate statistical concepts. They are then followed by a brief description dealing with specific Julia language issues. Nevertheless, if learning Julia is your focus, by using and experimenting with the examples you can learn the basics of Julia as well. The code examples can be downloaded from the book's GitHub repository:

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

Further, an erratum, and an electronic version of Appendix A's how-to guide, can be found in the book's website:

`https://statisticswithjulia.org/`

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

Chapter 1 is an introduction to Julia, including its setup, package manager, and a list of the main packages used in the book. The reader is introduced to some basic Julia syntax, and programmatic structure through code examples that aim to illustrate some of the language's basic features. As it is central to the book, basics of random number generation are also introduced. Further, examples dealing with integration with other languages including R and Python are presented.

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, and quantiles.

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 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.

Chapter 8 covers least squares, statistical linear regression models, generalized models, and a touch of time series. 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. Generalized linear models are introduced and an introduction to time series analysis is also presented.

Chapter 9 provides a broad overview of machine learning concepts. The concepts presented include supervised learning, unsupervised learning, reinforcement learning, and generative adversarial networks. In a sprint presenting methods for dealing with such problems, examples illustrate multiple machine learning methods including random forests, support vector machines, clustering, principal component analysis, deep learning, and more.

Chapter 10 moves on to dynamic 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. It also includes examples of basic epidemic modeling.

In addition to the core material, the book also contains 3 appendices. Appendix A is particularly useful as it can serve as a basic "how to guide" for Julia, calling upon the many examples in the chapters for reference.

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 deal 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 related to statistics, machine learning, data science, and artificial intelligence that were not used in this book.

Whether you are an industry professional, a student, an educator, a researcher, or an enthusiast, we hope that you find this book useful. Use it to 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 many 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. We also thank Joe Grotowski and Matt Davis from The University of Queensland for additional help dealing with the publishing process. Yoni Nazarathy would also like like to acknowledge the Australian Research Council (ARC) for supporting part of this work via Discovery Project grant DP180101602.

Yoni Nazarathy and Hayden Klok.

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 or C/C++ 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 in 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?

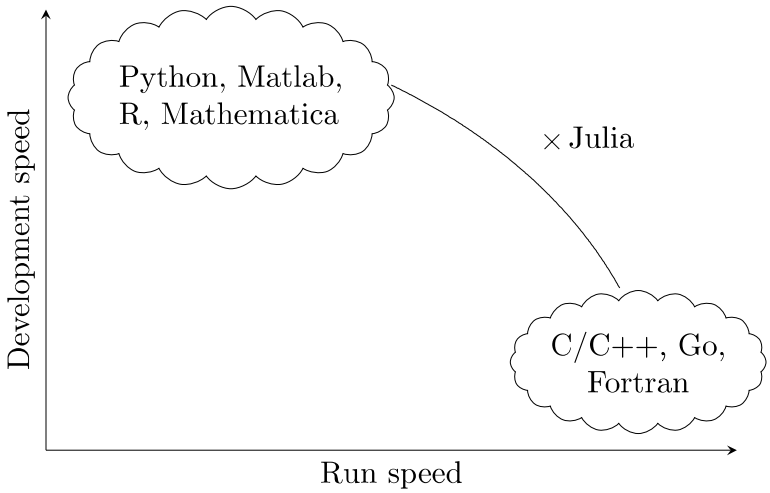

Figure 1.1: A schematic of run speed vs. development speed.
Observe the Pareto-optimal frontier existing prior to Julia.

The *Julia language and framework* developed in the last decade 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, style, and feel 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. In contrast to Python, R and other high level languages, the Julia standard libraries, and almost all of the Julia code base is 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 knowledge of statistics, machine learning, data science, and artificial intelligence. 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 200), 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 or build your understanding of probability, statistics, machine learning, data science, and artificial intelligence. In fact, the book contains a self contained overview of these fields, taking the reader through a tour of many concepts, illustrated via Julia code examples. Even if you are a seasoned statistician, data-scientist, machine learner, or probabilist, we are confident that you will find some of our discussions and examples interesting and gain further insight.

Question: *Do I need to have any statistics or probability knowledge to read this book?*
Answer: Statistics or probability knowledge is not pre-assumed. Hence, this book is a self-contained guide for the core principles of probability, statistics, machine learning, data science, and artificial intelligence. It is ideally suited for engineers, data-scientists, or science professionals, wishing to strengthen their core probability, statistics, and data science knowledge while exploring the Julia language. However, general mathematical notation and results including basics from linear algebra, calculus, and discrete mathematics are used.

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 at a basic level, will be able to follow the code examples 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. This book is code example centric. The code examples are the backbone of the story with each example illustrating a statistical concept together with the text, figures, and formulas that surround it. Y In any case, feel free to use the code-repository on GitHub:

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

As you do so, you can try 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. For ad-hoc Julia help, you may also use Appendix A, "How-to in Julia". it directs you to individual code listings that contain specific examples of "how to". It is also searchable online.

Question: *What are the unifying features of the code examples?*
Answer: With the exception of a few examples focusing on Julia basics, most code examples in this book are meant to illustrate statistical concepts. Each example is designed to run autonomously and to fit on a single page. Hence the code examples are often not optimized for efficiency and modularity. Instead, the goal is always to "get the job done" in the clearest, cleanest, and simplest way possible. With the aid of the code, you will pick up Julia syntax, structure, and package usage. However, you should not treat the code as ideal scientific programming code but rather as illustrative code for presenting and exploring basic concepts.

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, then it is possible that some 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 elementary probability is introduced starting with basic principles. The content then 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.

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 1.4 as of the middle of 2020. 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.

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 code lines does. 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 of future code examples, we generally omit comments. 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])
```

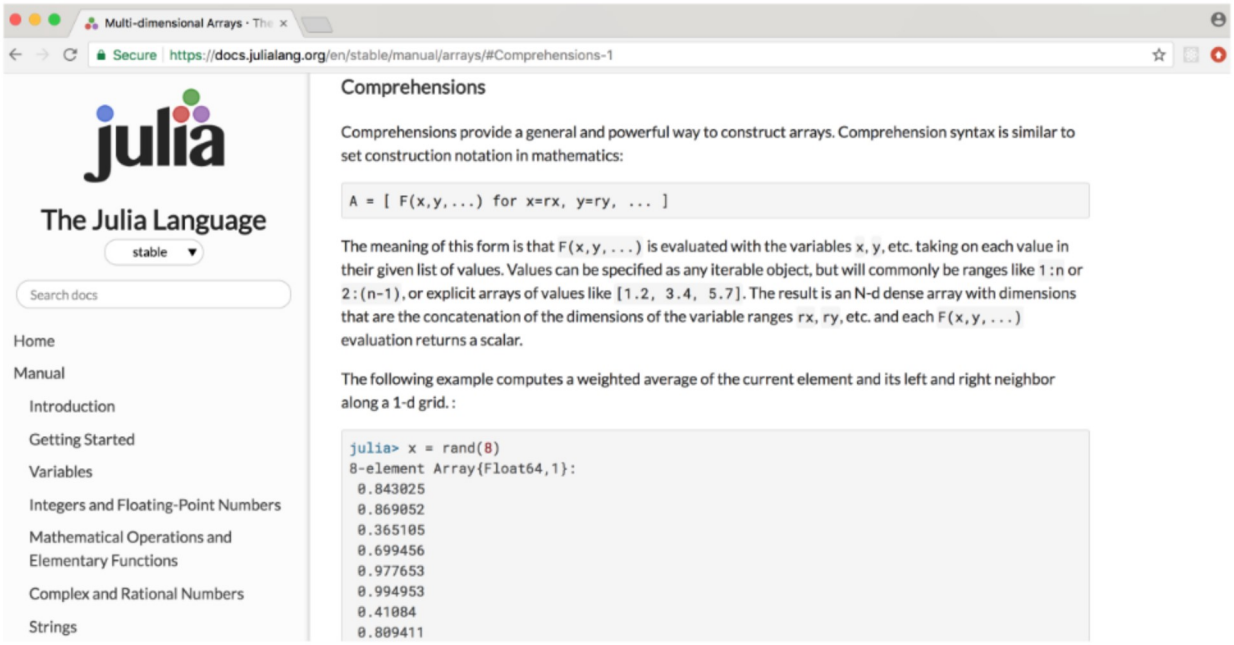

Figure 1.2: Visit `https://docs.julialang.org` for official language documentation.

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 $\star$ 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.

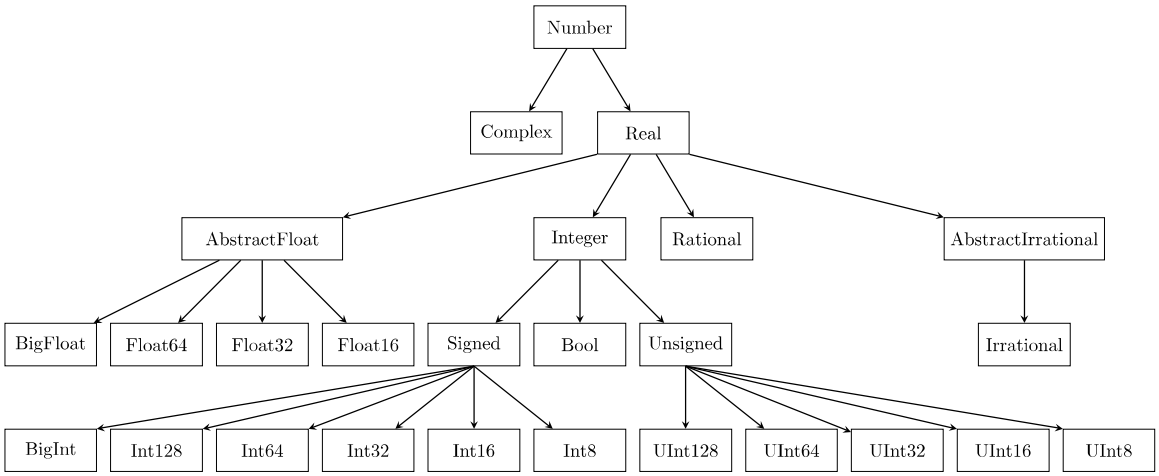

Figure 1.4: Type hierarchy for Julia numbers.

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.

Variable Scope, Local Variables, and Global Variables

The *scope of a variable* is the region of code in which the variable is visible. Like almost any other programming language, Julia has rules about variable scope implying that not all variables can be accessed from everywhere within the program. Such restrictions serve several purposes including driving the programmer to create clear readable code, allowing the compiler to optimize execution, supporting concurrent operation, and reducing the possibility of name clashes.

When discussing scope, a key distinction lies between *local variables* and *global variables*. The former refers to variables defined within a `function` definition or a block of code such as a `for` loop or `while` loop. The latter refers to variables that can potentially accessed from anywhere in the program. A detailed description of variable scope rules is in the Julia documentation, see `https://docs.julialang.org/en/v1/manual/variables-and-scoping/` . Here we only focus on a few key issues that are relevant to many of our code examples.

In general a global variable is a variable defined outside of a function or another block of code. Since global variables have a much less restricted domain than local variables, it is typically good programming practice to minimize or even eliminate their use. This is especially true for programs that span more than a single file and/or multiple lines of code (hundreds or thousands). Some Julia mechanisms that allow to organize variables include *structures* using the `struct` keyword and *modules* using the `module` keyword. However our code examples aim to be short self contained scripts and don't explicitly try to encapsulate data. We don't explicitly use structures and modules

and we define functions only if these are critically needed. Our aim is for simple code that fits on a minimal footprint. As a consequence our code examples often use global variables. If you wish to integrate parts of our code examples in larger projects, then it is good practice to eliminate or significantly reduce the use of globals as you carry out such integration. In less trivial coding situations, you should aim to hardly use global variables and the `global` keyword. Still, for the purposes of illustrative code snippets, our heavy use of globals is justified.

As a minimal introduction to variable scope we present Listing 1.5. One of the main goals of this listing is to show the use of the `global` keyword which is prevalent in many of the code listings that follow. You may wish to skip reading the details of this listing and then refer back to it when thinking and considering variable scope. The important point is simply to observe that the `global` keyword is sometimes needed and is thus present in quite a few of the examples that follow.

The key aspect of Listing 1.5 is the execution of a `for` loop in global scope followed by a similar loop wrapped within a function. The global variables `data`, `s`, `beta`, and `gamma` are potentially visible in all parts of the code, including in the `for` loop of lines 5-12 and the function `sumData()`. However, in certain cases the `global` keyword needs to be used to mark the variable as "coming from" global scope. This is the case for the variable `s` that is marked as global in line 7. Interestingly, when using a Jupyter notebook for such code (Jupyter notebooks are described in the next section), such usage of the `global` keyword is not needed. The listing presents multiple other aspects of variable scope. We detail some of these aspects in the code comments that follow. A full description of variable scope rules is in the Julia documentation. For more details, visit:

Listing 1.5: Variable scope and the `global` keyword

```
1  data = [1,2,3]
2  s = 0
3  beta, gamma = 2, 1
4
5  for i in 1:length(data)
6      print(i," ")
7      global s    #This usage of the `global` keyword is not needed in Jupyter
8                  #But elsewhere without it:
9                  #ERROR: LoadError: UndefVarError: s not defined
10     s += beta*data[i]
11     data[i] *= -1
12 end
13 # print(i)       #Would cause ERROR: LoadError: UndefVarError: i not defined
14 println("\nSum of data in external scope: ", s)
15
16 function sumData(beta)
17     s = 0           #try adding the prefix global
18     for i in 1:length(data)
19         s += data[i] + gamma
20     end
21     return s
22 end
23 println("Sum of data in a function: ", sumData(beta/2))
24 @show s
```

```
1 2 3
Sum of data in external scope: 12
Sum of data in a function: -3
s = 12
```

In line 1 we define an array, `data`. It is a variable defined in global scope and is hence a global variable. Similarly for the variables `s`, `beta`, and `gamma` in lines 2-3. Lines 5-12 loop over the range `1:length(data)` where in each iteration the variable `i` takes the next value. The scope of the variable `i` is within the block of the for loop (lines 5-12). Note that if you were to uncomment line 13, the attempt to access `i` at that line would cause an error. Because the for loop is not inside a function, it defines a new local scope. This means that accessing the global variable `s` for modification requires an explicit declaration with the `global` keyword as is done in line 7. For code in Jupyter notebooks this can be avoided but otherwise not. Notice however that `global` declarations are not always needed. For example the global variable `beta` is used in line 10, but as it isn't modified there is no need to declare it as global with `global`. The variable (array) `data` also doesn't need to be declared even though the contents of the array is modified in line 11. In lines 16-22 we define the function `sumData()`. Here the name of the function argument is `beta` and is not the global variable `beta`. Hence when the function is called in line 23, we can pass any argument to it for the local `beta`. In this case the argument is half of the global `beta`, i.e. a value of 1. Note that we define a local variable `s` in line 17. It is a different variable from the global `s` defined in line 2. If we were to add a `global` keyword in line 17 then it would be the global variable `s`. You can try doing that and see how the `@show` macro in line 24 that displays the value of the global `s` would change. Note again that the global variables `data` and `gamma` are used inside the body of the function for read only purposes.

1.2 Setup and Interface

There are multiple ways to run Julia including the *REPL command line interface*, *Jupyter notebooks*, the *Juno IDE* (Integrated Development Environment) on the *Atom* editor, as well as several other working environments. Here we focus on the REPL and Jupyter notebooks as these are the most mature environments to date. We also mention that in developing the code examples, we used both Jupyter notebooks and the Juno IDE. The latter is also available directly from Julia Computing and is packaged as *Julia Pro*.

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. We first describe the *REPL* and *Jupyter notebooks* environments. We then describe the *package manager* which allows one to extend Julia's basic functionality by installing additional packages.

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.

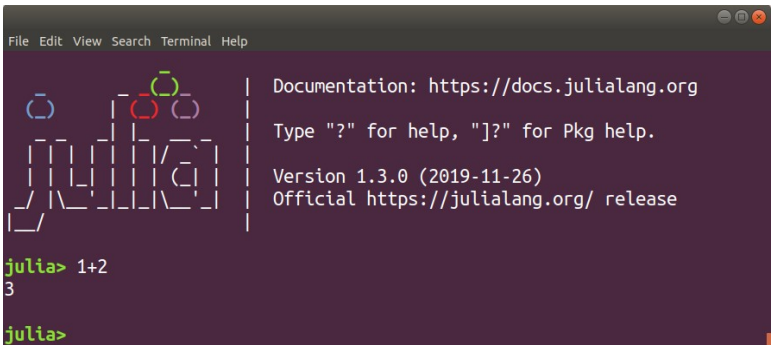

Figure 1.5: Julia's REPL interface.

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 in 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.

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.

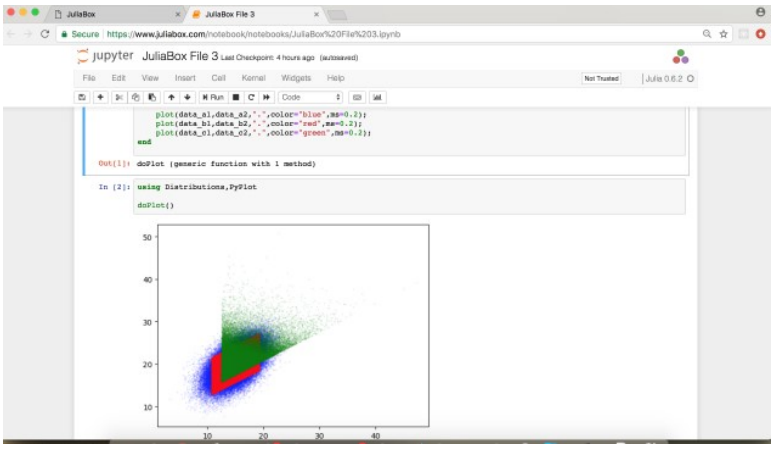

Figure 1.6: An example of a Jupyter notebook accessed via JuliaBox.

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://julialang.org/packages/`.

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`	adds the package `Foo.jl` to the current Julia build.
`] status`	lists what packages and versions are currently installed.
`] update`	updates existing packages.
`] remove Foo`	removes package `Foo.jl` from the current Julia build.

An alternative which works both in the REPL and in Jupyter notebooks is to use functions from the `Pkg` package. The standard usage is of the form `using Pkg` followed by `Pkg.add("Foo")`. This adds the package `Foo.jl`. Similar functions exist via the `Pkg` package for other package operations.

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. We have placed a '*' next to every package that is part of the basic installation.

`*Base.jl` is the basic Julia package sitting at the base of the language.

`BSON.jl` allows to store and read data using the common Binary JSON format.

`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` allows to enumerate 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` 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.

`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.

`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.

Juno.jl is a package needed for using the Juno development envionrment.

KernelDensity.jl is a kernel density estimation package.

Lasso.jl implements LASSO model fitting..

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

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

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

**LinearAlgebra.jl* 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.

MLDatasets.jl provides an interface for accessing common Machine Learning (ML) datasets..

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.

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 Julia.

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* 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 R's more popular packages.

Roots.jl contains routines for finding roots of continuous scalar functions of a single 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* contains common statistics functions such as mean and standard deviation.

StatsBase.jl provides basic support for statistics including high-order moment computation, counting, ranking, covariances, sampling and cumulative distribution function estimation.

StatsModels.jl allows to specify models using formulas as common in linear models.

StatsPlots.jl provides extensive statistical plotting recipes.

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

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, [K18] focuses on *operations research* using Julia. Finally, the book [MP2018] is an applied *data science* resource, as is [V16].

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 1.6. 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 1.6: Bubble sort

```
1  function bubbleSort!(a)
2      n = length(a)
3      for i in 1:n-1
4          for j in 1:n-i
5              if a[j] > a[j+1]
6                  a[j], a[j+1] = a[j+1], a[j]
7              end
8          end
9      end
10     return a
11 end
12
13 data = [65, 51, 32, 12, 23, 84, 68, 1]
14 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 1.7 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 1.7: 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.

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 1.8. 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,

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

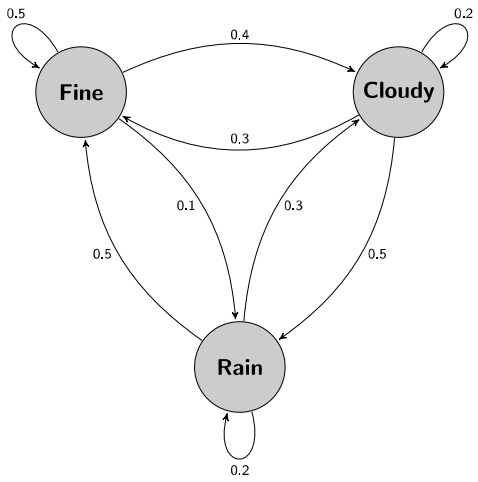

Figure 1.7: Three-state Markov chain of the weather.
Notice the sum of the arrows leaving each state is 1.

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 1.8: 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

display([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. 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 1.9 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 1.9: 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.

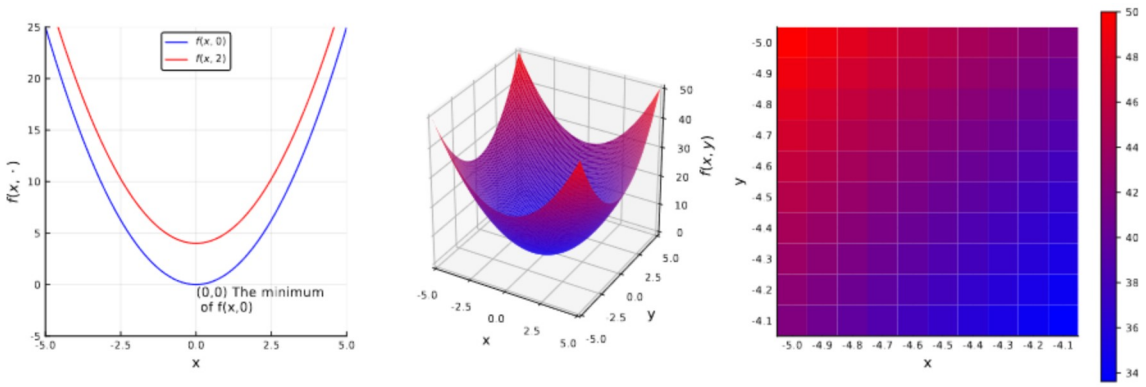

Figure 1.8: An introductory `Plots` example.

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 1.10. 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 1.10: 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 1.11, 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.

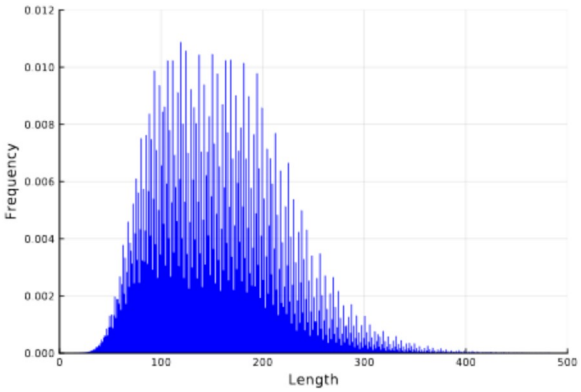

Figure 1.9: Histogram of hailstone sequence lengths.

Listing 1.11: 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 1.12 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 1.12: 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)
```

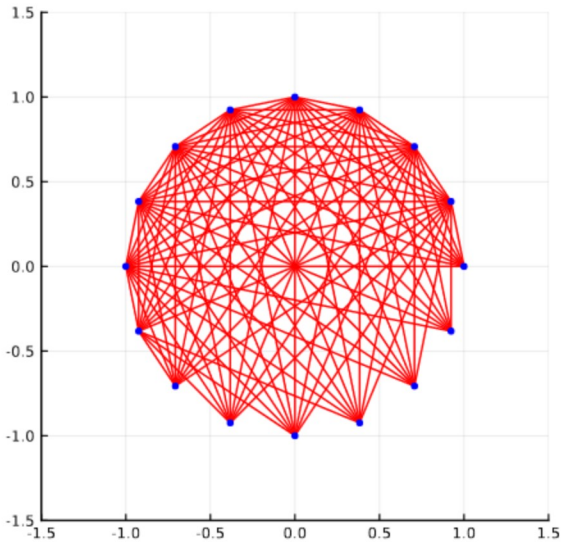

Figure 1.10: Sample frame from a graph animation.

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 1.13 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 passing 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.

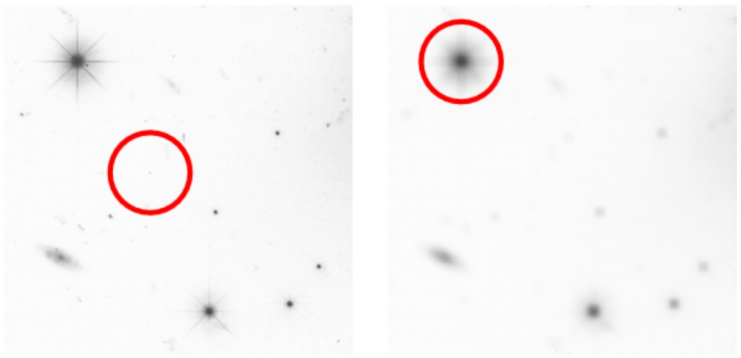

Figure 1.11: Left: Original image.
Right: Smoothed image after noise removal.

Listing 1.13: 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 1.14 illustrates setting the seed using Julia's `Random.seed!()` function.

Listing 1.14: 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

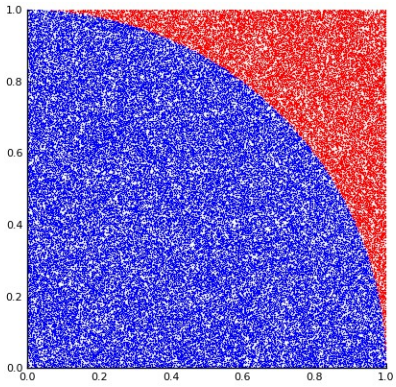

Figure 1.12: Estimating π via Monte Carlo.

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 1.15 for 10^5 points. The listing also creates Figure 1.12.

Listing 1.15: 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 1.16 we generate values based on this LCG, see also Figure 1.13.

Listing 1.16: 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)
```

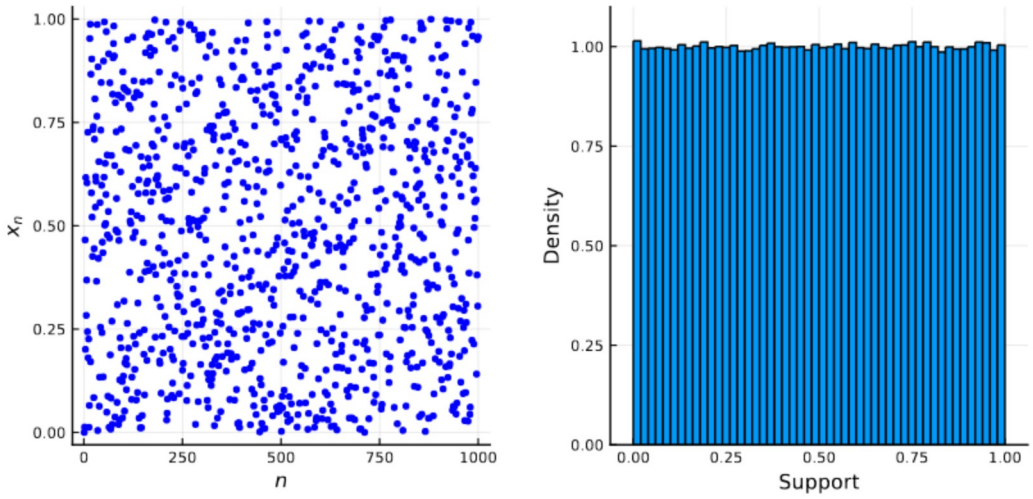

Figure 1.13: Left: The first 1,000 values generated by a linear congruential generator, plotted sequentially. Right: A histogram of 10^6 random values.

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.

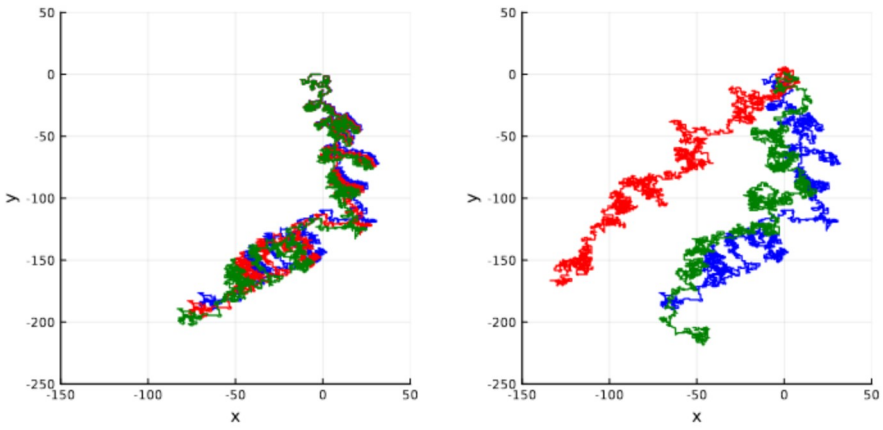

Figure 1.14: Random walks with slightly different parameters.
Left: Trajectories with same seed. Right: Different seed per trajectory.

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 1.17 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 1.17: 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 1.18. 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 1.18: 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 1.19 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 1.19 can be executed. This example only briefly touches on the `PyCall` package with more information available in the package documentation.

Listing 1.19: 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
```

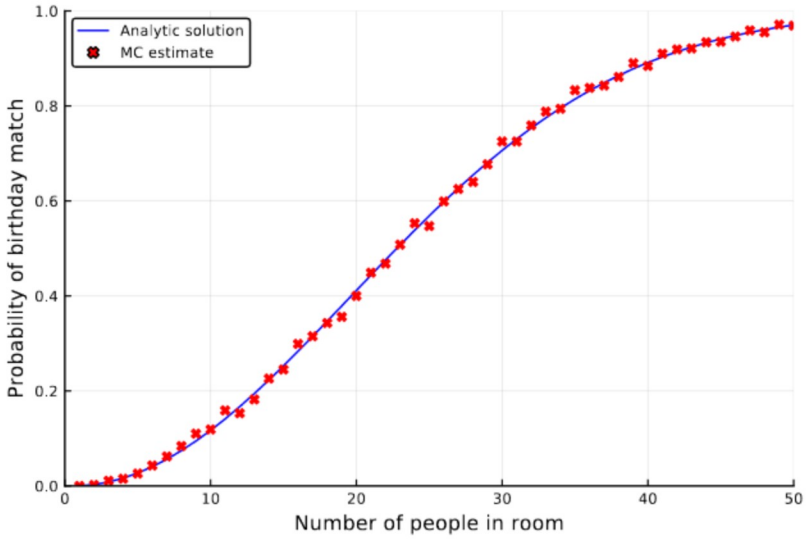

Figure 2.1: Probability that in a room of n people, at least two people share a birthday.

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)
```

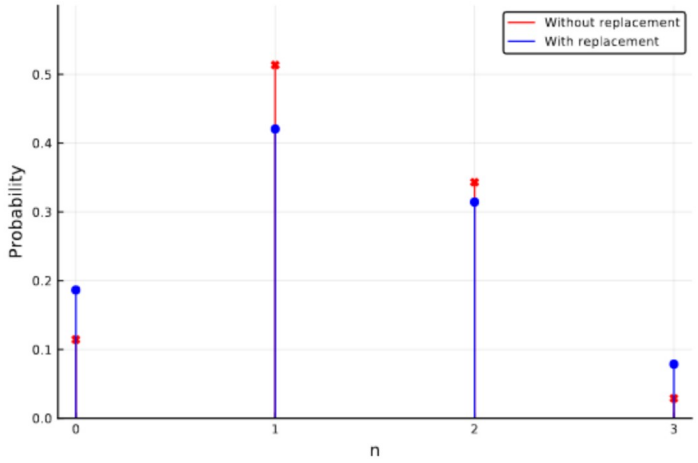

Figure 2.2: Estimated probabilities of catching n of gold fish, with and without replacement.

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