APPENDIX E

Using Julia

Chapters 6 and 7 relied heavily on computer simulations. All those simulations were conducted in Julia, a new, free, and open-source computer language, distributed under the MIT license. Julia is a general-purpose language geared toward high-performance computing. It was originally developed at MIT by Alan Edelman with a team of colleagues, post docs, and doctoral students; now it has contributors from around the globe. Three of the main developers were awarded the James H. Wilkerson Prize for Numerical Software for their contributions to the creation of Julia. Importantly, Julia is written almost entirely in Julia, so that once you have mastered Julia, you are also able to understand (and if needed, adapt) the code for all predefined Julia functions and for the Julia packages. This is very different from languages such as Python or R, large parts of which are written in C or C++.

You should be able to follow the arguments in this book without even switching on your computer. Nevertheless, I recommend that you do have a look at Julia and maybe even try your hand at the code for the simulations that I am making available. According to the mathematician Gregory Chaitin,

> the computer changes epistemology, it changes the meaning of "to understand." To me, you understand something only if you can program it. (You, not someone else!) Otherwise you don't really understand it, you only think you understand it. (Chaitin, 2006, p. xiii)

I could not agree more. Because Julia is so new, I am assuming that most readers will not be familiar with it. I therefore start with a brief introduction to the language, focusing on the elements of the language that are most essential to the simulations.

If you have heard about Julia previously, then probably you have heard about its speed, which is generally seen as one of Julia's main selling points. In

many applications, one should not be overly concerned about one's code not being optimally fast: as computer scientists like to say, your own time is more valuable than the computer's time. Indeed, that performance is not always important is especially true now that we can run computations in the cloud for little money. In my experience, though, sometimes speed really does matter. Especially in the exploratory stage of one's research, when the code is still being developed and bugs easily slip in, it can be helpful to see the outcome of a computation *quickly*. It may just take too much of one's patience if it takes hours to get some first results that might indicate whether the ideas one is pursuing have merit or not. Even when one is happy with one's code and deems it ready to go—if it takes a week to run in the cloud, then that will cost you more than just a little money. Depending on the software one uses, a week may be a realistic expectation for the simulations that were reported in this book. When Sylvia Wenmackers and I were working on our joint 2017 paper, we wrote the simulations mentioned in chapter 7 in *Mathematica*. That code could then be run only on a supercomputer to which we had institutional access at the time. Even on that computer, the simulations took four days to complete. Running the same simulations, now written in Julia, on my own desktop machine (which, admittedly, is quite a beefy machine, with twelve physical and twenty-four logical cores) takes under half an hour.[1] The Julia website (https://julialang.org/) exhibits interesting benchmarks, comparing Julia with a host of other languages, including *Mathematica*, R, and Python, and showing that, for some common computational operations, Julia is at least one hundred times faster than each of those other widely used languages.

What makes Julia a fantastic language to work with is not simply its speed but rather the combination of speed and user-friendliness. Languages like C, C++, FORTRAN, and some others are fast as well, sometimes even still a bit faster than Julia. But all these languages have a steep learning curve, and even professional programmers working in them on a daily basis tend to spend a fair amount of their time debugging code, simply because it is so easy to make mistakes in these languages. Julia code is very different in this respect. While Julia code can run as fast, or nearly as fast, as code written in the aforementioned languages, it looks almost like pseudocode, and users of *Mathematica*,

1. In fairness, it should be mentioned that Sylvia Wenmackers and I used *Mathematica 8*, the most recent version being *Mathematica 12*. I have limited experience with the latter, but it is clearly more performant than the older version that we used.

Python, or R should have little difficulty familiarizing themselves with Julia and can become productive in the language within a day or two.

I started using Julia in mid-2017, when it was still in beta; to be precise, I started using version 0.5.2. Julia 1.0 was released in August 2018, the current version (November 2020) being 1.5.3. Over those years, I have not encountered any downside of using Julia. To be sure, precisely because the language is so new, its "ecosystem" is still not quite at the level of Python's or R's, both of which have been around for decades. For instance, there are not nearly as many packages for Julia as exist for Python or R, and many of the Julia packages are still in beta. But, first, it is just a matter of time for Julia to catch up, and second, there are easy ways to call Python and R from within Julia, which allows one to use any Python or R package also in Julia, subsequently explained. The available documentation for Julia, in print or online, is also rather limited still. That situation, too, will soon improve. Besides, the community of Julia users is very friendly, and I have received valuable help by asking questions on the main online forum for Julia (https://discourse.julialang.org/).

There are free services that allow you to run Julia in the cloud (e.g., Google's Colab, and NextJournal), but if you are serious about giving Julia a try, then I recommend that you install it on your own computer. Details about how to install Julia (which are different for different operating systems) can be found on the download page from the Julia language website: https://julialang.org/downloads/. There are various ways to interact with Julia. It is possible to run Julia from within the terminal, but I have mostly been using either Microsoft's Visual Studio Code editor or JupyterLab, both of which are free and open-source and easy to set up (although details depend on the operating system). Recently, I have also come to enjoy using Pluto, which is a new, also free and open-source, technology that was especially developed for Julia and which can be installed via Julia's package manager (see below).

Language Basics

Two key elements of virtually any computer language are what one may generally refer to as *containers* and *control flow elements*. The former are where you store, at least for a while, your data or the outcomes of your computations; the latter are elements that let you automate repetitive tasks, which is the kind of operation that many people most immediately associate with computers.

The most common type of container in Julia, and in many other languages, is the so-called array. In Julia, arrays can have any number of dimensions. Perhaps it is easiest to think of a two-dimensional array as the most basic case. Such an array is characterized by its number of rows and its number of columns. But if we have a series of such two-dimensional arrays, we may want to store them together in one container. Provided that they have the same number of rows and columns, that is possible, namely, by putting them in a *three*-dimensional array, the first two dimensions still being rows and columns, and the third dimension being an enumeration of the two-dimensional arrays that we brought together. And, of course, if we want to store a number of three-dimensional arrays together, we can do that as well (given that they have the same dimensions), namely, in a four-dimensional array (arrays with more than two dimensions are also referred to as "tensors"). And so on. Julia recognizes `Vector` as shorthand for a one-dimensional array and `Matrix` as shorthand for a two-dimensional array.

To give an example, the code `rand(3, 4)` creates a 3×4 (i.e., three rows by four columns) array of random floating point numbers (or floats) between 0 and 1 inclusive:[2]

```
julia> rand(3, 4)
3×4 Array{Float64,2}:
 0.481211  0.205386  0.968688  0.239485
 0.385465  0.691358  0.295478  0.215576
 0.911691  0.224254  0.984399  0.541366
```

We can similarly create a $3 \times 4 \times 2$ array of random floats between 0 and 1 inclusive:

```
julia> rand(3, 4, 2)
3×4×2 Array{Float64,3}:
[:, :, 1] =
 0.190611   0.656259  0.556281  0.858714
 0.0814072  0.164408  0.954222  0.92906
```

2. Although modern technology still will not let *you* run a Julia session within this book, it will let *me* do so. Well, sort of. I can run Julia from within the document that will eventually be typeset into this book, thanks to Geoffrey Poore's wonderful `pythontex` package (Poore, 2015). What *looks* like a Julia prompt actually *is* a Julia prompt, and everything printed in sans serif immediately below a prompt is actual Julia output.

```
 0.83987    0.974812  0.474558  0.0514526
[:, :, 2] =
 0.765418  0.750336  0.0654037  0.476914
 0.834557  0.95696   0.0157077  0.491732
 0.39551   0.831768  0.428758   0.279625
```

Think of this as having created two two-dimensional arrays (of size 3×4) that are then stored together in one container.

From the output above, it can already be seen that Julia is a *typed* language. For instance, the line immediately below `rand(3, 4, 2)` indicates that this code outputs an array containing 64-bits floats and that this array has three dimensions. You do not have to pay any attention to types while working in Julia. However, doing so often pays off in terms of performance. One reason why Julia is fast is that it has a clever way of reserving chunks of your computer's memory to store whatever values are going to be computed when your code is executed. Generally speaking, code runs much faster if the data it has to retrieve is stored orderly in a contiguous chunk of memory. To know in advance how much memory must be reserved, the compiler needs to know the type of the data that are going to be stored in it. For instance, if it has to store a thousand 64-bit floats it needs more space than if it has to store a thousand 32-bit integers.

In Julia, we access specific entries in an array, or slices of it, by indexing, much in the way in which this is done in R, Python, and many other languages.[3] Suppose that we have a two-dimensional array

```
julia> ar = rand(3, 4)
3×4 Array{Float64,2}:
 0.160707  0.569687  0.797983  0.760631
 0.922007  0.57953   0.358656  0.0114223
 0.286155  0.796654  0.456668  0.698085
```

Then

```
julia> ar[2, 3]
0.3586560380995427
```

returns the entry in the second row, third column, and

3. However, Python users should keep in mind that Julia is not 0-indexed but (like R) 1-indexed.

```
julia> ar[:, 4]
3-element Array{Float64,1}:
 0.7606313907554907
 0.011422317555960682
 0.6980853689654172
```

returns the whole fourth column of the array, and

```
julia> ar[1, :]
4-element Array{Float64,1}:
 0.1607070692312922
 0.569687258606496
 0.7979833103099228
 0.7606313907554907
```

returns the first row in its entirety. Similarly, we can get chunks of `ar` by using, for example,

```
julia> ar[2:3, 1:3]
2×3 Array{Float64,2}:
 0.922007  0.57953   0.358656
 0.286155  0.796654  0.456668
```

which returns from rows 2 and 3 the numbers that are in columns 1 to 3. The same principles apply no matter the dimensionality of the array. To work with arrays, you have to know a bit about how to permute their dimensions (e.g., via the `permutedims` function), how to reshape them more generally (e.g., via the `reshape` function), how to apply functions to specific dimensions (row-wise or column-wise, e.g., via the `mapslices` function), how to concatenate two or more arrays (e.g., using `hcat` or `vcat`), how to find or select specific values or values meeting some criterion, in an array (e.g., using `findall` or `findfirst` or `filter`), and so on. I will not cover these issues here. In general, with the help of the internet, you should have no difficulty figuring out how to perform whichever operation you would like to perform on an array.

Computers can repeat an operation over and over again, thousands of times, hundred thousands of times, millions of times, and more, and they can do so *fast*, or in any event much, much faster than we ever could. In most simulations, and certainly in the ones that we reported, that is exactly what a computer is needed for. There is no step in any of those simulations that we

could not have calculated by hand. But if we had had to do that for each step, this book would never have come into existence. We could not have calculated all the steps of our simulations in our life time, no matter the effort we would have been willing to make. To define such repetition procedures, we need control flow elements.

An especially important automated repetition procedure, often used in the simulations, is the **`for`** loop. In such a loop, we go through a range of values, and for each value, run whichever procedure in which we are interested. There are variants, such as **`while`** loops, that repeat a procedure while some prespecified condition holds. If you come from R, you may be inclined to think that **`for`** loops are to be avoided for being relatively slow, and that instead one should use so-called vectorized code, where, roughly, a function is mapped over a vector, or an array, or another structure. That is not true in Julia, however: **`for`** loops are in general fast, and faster than vectorized code (although almost anything in Julia is fast, including vectorized code; but **`for`** loops are normally still a bit faster).

In the following code, we first pre-allocate a vector (i.e., a one-dimensional array) that can contain 1,000 integers and then use a **`for`** loop to fill it with the squares of the integers from 1 to 1,000. We time the code via the `@time` macro.

```
julia> v = Vector{Int64}(undef, 1000);
julia> @time for i in 1:1000; v[i] = i^2; end
  0.000025 seconds (1.47 k allocations: 22.922 KiB)
```

Using vectorized code, we can create the same vector as follows:

```
julia> @time v = [ i^2 for i in 1:1000 ];
  0.019676 seconds (41.84 k allocations: 2.221 MiB)
```

But this is clearly slower and also allocates more memory.

If one needs very precise measurements, one should use the `@btime` macro, or for more informative output the `@benchmark` macro, from the package `BenchmarkTools.jl`.[4] (See the subsequent explanation on how to install and load packages.) To see this in action, we show an even faster way to square the

4. Using the `@time` macro (but not the `@btime` or `@benchmark` macro), one can verify that the first call to a function always takes a bit longer than subsequent calls. The reason is that although Julia *feels* like an interactive language (you type some code, e.g., in a Julia session running in the terminal, or in a Jupyter notebook, and it will immediately execute), it really is a compiled language, like C, C++, FORTRAN, and many others. However, whereas these

first thousand positive integers. This defines a vector containing just these integers and then squares each of them in place, meaning that the vector gets replaced by another vector containing the squares of the first thousand positive integers. That does not require any moving around of data in memory, which makes this a very fast operation:

```
julia> @benchmark v = (1:1000).^2
BenchmarkTools.Trial:
  memory estimate:  7.94 KiB
  allocs estimate:  1
  --------------
  minimum time:     738.293 ns (0.00% GC)
  median time:      810.732 ns (0.00% GC)
  mean time:        1.055 μs (14.74% GC)
  maximum time:     7.697 μs (87.11% GC)
  --------------
  samples:          10000
  evals/sample:     164
```

We see that, on my computer, this takes just about one microsecond, on average. That makes this way of creating v about eighteen times faster than the **for** loop that we used above.[5] (For details on how to use this package, see the information at https://github.com/JuliaCI/BenchmarkTools.jl.)

Here is an example of a **while** loop:

```
julia> i = 1; while i < 5; println(i); global i += 1; end
1
2
3
4
```

We first set *i* equal to 1 outside the loop and then increment it by 1 each time we go through the loop. In each iteration, we want the value of *i* in that iteration to be printed.

more traditional languages require the user first to compile an executable of his or her code, Julia does what is called "just-in-time compilation," meaning that, for instance, a function gets compiled just before it is needed for the first time. All further calls to the function make use of the compiled code that was created on the first call and that also got stored then.

5. This was actually determined using the `@btime` macro, not via the `@time` macro reported in the text above.

Another important class of control flow elements is conditional statements. Suppose we want a program to output the value of some variable x if that value is greater than 0.95 and else want it to output π. One way that we can get this done in Julia uses the so-called ternary operator:

```
julia> x = rand()
0.5020104104255427
julia> x > 0.95 ? x : π
π = 3.1415926535897...
```

(Note, incidentally, that Julia understands Unicode!) Another way to achieve the same result uses the `ifelse` operator:

```
julia> ifelse(x > 0.95, x, π)
π = 3.1415926535897...
```

There are still further conditional operators in Julia; see subsequent discussion and the Supplementary Materials.

At least in the simulations, what we want the control flow elements to help us do is perform mathematical operations on synthetic data. Julia's notation of mathematical operations is fairly standard, for instance:

```
julia> 2 + 3
5
julia> 2 * 3
6
julia> 2^3
8
```

Boolean evaluations and set-theoretic operations are also fairly standard:[6]

```
julia> a = 12 < 3
false
julia> b = 3 + 4 ≤ 7
true
julia> a && b  # conjunction operator
false
```

6. Julia interprets anything following the # symbol as a comment and does not try to evaluate it.

```
julia> a || b  # disjunction operator
true
julia> !a      # negation operator
true
julia> [2, 3] ∩ [2, 5, 6]
1-element Array{Int64,1}:
 2
julia> [2, 3] ∪ [2, 5, 6]
4-element Array{Int64,1}:
 2
 3
 5
 6
julia> [2, 3] ⊆ [2, 5, 6]
false
julia> 3 ∈ [2, 5, 6]
false
julia> 3 ∉ [2, 5, 6]
true
```

You can also define your own functions:

```
julia> f(x) = sin(x) + sqrt(x)
f (generic function with 1 method)
julia> f(4)
1.2431975046920718
```

Multiline function definitions must be prepended with **`function`** and must end with **`end`**. For instance, the following function defines Bayes's rule specifically for the kind of data that we use in our simulations:

```
function bayes_rule(probs, toss)
    likelihood_heads = 0:1/(length(probs) - 1):1
    likelihood_tails = reverse(likelihood_heads)
    if toss == 1
        (probs .* likelihood_heads) ./ dot(probs, likelihood_heads)
    else
        (probs .* likelihood_tails) ./ dot(probs, likelihood_tails)
    end
end
```

Suppose that `probs` is your current probabilities for a set of jointly exhaustive and mutually exclusive hypotheses about the bias of a given coin and that `toss` is the outcome of a given toss. Then the function looks at whether the outcome was heads (coded as 1) or tails (coded as 0 in our simulations, though any value other than 1 would have done) and then, if the former, first multiplies your probabilities with the likelihoods bestowed on heads by the various hypotheses and then renormalizes them (by dividing each by the dot product of your probabilities and the likelihoods for heads) or, if the latter, does the same but now with likelihoods for tails replacing likelihoods for heads.[7]

Note the use of the period before the multiplication and division operators: that ensures that those operators are *broadcasted*, meaning that they apply element-wise (e.g., your probability for the hypothesis that the coin has a bias of x gets multiplied with the likelihood of the outcome on *that* bias hypothesis) and do not try to multiply, respectively divide, two whole *vectors* (which in the first case would throw an error and in the second would give the wrong outcome).

Broadcasting is also very convenient for selecting and replacing elements that have certain properties. For example, the following shows where in the array `ar` defined previously we find the elements greater than .5:

```
julia> ar .> .5
3×4 BitArray{2}:
 0  1  1  1
 1  1  0  0
 0  1  0  1
```

We can select those,

```
julia> ar[ar .> .5]
7-element Array{Float64,1}:
```

7. To make the dot product available, we must load the `LinearAlgebra.jl` package. It is also to be noted that the definition of Bayes's rule in the Jupyter notebooks contained in the Supplementary Materials looks a bit more complicated, but that is just because, thus defined, it is more convenient for how we set up the simulations. (The definition in the notebooks is also type annotated, as are all other definitions used for the simulations. Sometimes, using type annotations makes your code more performant, but more often, it has no influence on performance. Even then, I like to use type annotations, simply because they help me remember what kind of input or inputs a function expects.)

```
 0.9220074819252688
 0.569687258606496
 0.5795303164581178
 0.796653818790771
 0.7979833103099228
 0.7606313907554907
 0.6980853689654172
```

which we may for instance want to do if we need to add them up:

```
julia> sum(ar[ar .> .5])
5.124578945811484
```

Or perhaps we want to replace them all by some other value:

```
julia> ar[ar .> .5] .= 5;

julia> ar
3×4 Array{Float64,2}:
 0.160707  5.0  5.0       5.0
 5.0       5.0  0.358656  0.0114223
 0.286155  5.0  0.456668  5.0
```

And broadcasting together with the negation operator makes it easy to omit particular elements of a vector, or to omit rows or columns of an array. For instance,

```
julia> ar[1:end .!=2, 1:end .!=3]
2×3 Array{Float64,2}:
 0.160707  5.0  5.0
 0.286155  5.0  5.0
```

tells Julia to return `ar`, but with the second row and the third column removed.

Using the feature that Julia is a typed language, we can overload functions:

```
julia> g(x::Int64) = x^3
g (generic function with 1 method)
julia> g(x::Float64) = sqrt(x)
g (generic function with 2 methods)
julia> g(3)
```

```
27
julia> g(16.0)
4.0
```

In Julia jargon, there are now two *methods* associated with `g`. Which method is called depends on the type of the input argument.

In Julia, you can even define your own types, which are then on a par with Julia's built-in types. Suppose that we define

```
julia> struct my_type; a::String; end
```

which introduces a new type, `my_type`, which has one field that can contain a string.[8] Then we could add a further method to the ones already defined for `g`, for instance,

```
julia> g(x::my_type) = "Hello, " * x.a
g (generic function with 3 methods)
```

Then if

```
julia> s = my_type("Alice and Bob");
```

the function `g` allows us to greet our friends Alice and Bob:

```
julia> g(s)
"Hello, Alice and Bob"
```

(The type and method serve merely illustrative purposes here: I cannot think of any reason why one would want to introduce either.) This feature also enables multiple dispatch, which we briefly discussed in chapter 1.

It was already mentioned that Julia uses packages (or libraries). These contain functions that serve special purposes. Some of those packages are preinstalled, but most are not and must be installed via the package manager. For instance, to install the `Distributions.jl` package, press the] key and enter (and execute) `add Distributions`. (To leave the package manager, press the backspace key.) To use a package in a session, you must explicitly load it. For instance, to load the `Distributions.jl` package, enter `using Distributions`.

8. While type annotations are often optional in function definitions, as mentioned in footnote 7 in this appendix, in type definitions it is strongly recommended to annotate the field or fields, as this will help the Julia compiler to optimize the assembly code.

To repeat, at the time of this writing, a number of packages are still in beta, which means that later versions may introduce breaking changes. Often, package developers then include deprecation warnings, telling you how to update your old code—often, but not always, which can lead to frustrating experiences, as can (at the moment) dependency issues and some other quirks.[9]

Two packages that deserve special mention are `RCall.jl` and `PyCall.jl`. The first allows you to call R from within Julia, the second, from within Python. (They require R and, respectively, Python to be installed on your computer.) For instance, the following lines of code perform an ANOVA test using R's `aov` function:

```
julia> using RCall
julia> u = rand(100);
julia> v = rand(100) .+ 0.1;
julia> w = rand(100) .- 0.1;
julia> x = repeat(1:3, inner = 100);
julia> @rput u; @rput v; @rput w; @rput x;
julia> R"summary(aov(c(u, v, w) ~ as.factor(x)))"
RCall.RObject{RCall.VecSxp}
              Df Sum Sq Mean Sq F value   Pr(>F)
as.factor(x)   2  2.369  1.1844    15.9 2.76e-07 ***
Residuals    297 22.127  0.0745
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Here, the calls to `rand(100)` create vectors of one hundred random floats between 0 and 1 inclusive. To create `v`, we add 0.1 to *each* random value in the generated vector, by broadcasting the `+` operator (if we used `+` instead of `.+`, we would be trying to add 0.1 to a vector, which makes no sense and accordingly would throw an error), and similarly for `w`. The vector `x` is to serve as our predictor. Then the `@rput` macro is used to make `u`, `v`, `w`, and `x` available to R. The use of `PyCall.jl` is similar, for instance,

```
julia> using PyCall
julia> math = pyimport("math");
```

9. You can ignore warnings because they will not prevent your code from running. Errors *cannot* be ignored; they have to be fixed, or your code will not run.

```
julia> math.sqrt(2)
1.4142135623730951
```

To use R or Python packages from within Julia, these must first be installed via the package managers for those languages (so, using `install.packages` for R and `pip` or `conda` for Python).

As a final general comment, I should mention how easy parallel computing is in Julia. Parallel computing leverages that nowadays virtually all computers have more than one core. Even computers with twenty-four or more cores have become relatively affordable. For the simulations we ran, having multiple cores available helped to speed up computation tremendously. In these simulations, various procedures are repeated over and over again, each time starting from a somewhat different random configuration, the results from the various runs then typically being averaged in some way at the end. Julia makes it very easy to divide those runs over different cores, which can then run the requisite computations in parallel, so that, for instance, if your computer has twenty-four cores, running twenty-four simulations takes about as long as running just one simulation (the former will take a *bit* longer because of the overhead associated with parallel computation). There are different ways to set up parallel computations in Julia, and which is best depends on details of the computations to be executed. I refer here to the highly readable part on parallel computing in the Julia manual, at https://docs.julialang.org/en/v1/manual/parallel-computing/, which also contains all the information about the macros and functions for parallel computing used in the notebooks in the Supplementary Materials.

Statistics in Julia

R is a computer language specifically for statistical computing. Whatever type of advanced statistical analysis you may want to conduct, there will be at least one dedicated R package for it. Julia is still comparatively limited in this respect. However, for all the standard types of analysis, there are excellent packages available, including `Statistics.jl`, `StatsBase.jl`, `Bootstrap.jl`, `HypothesisTests.jl`, `GLM.jl`, `Lathe.jl`, and the previously mentioned package `Distributions.jl`. In fact, Douglas Bates, the main developer of the `lme4` package for R, has switched to Julia precisely because it is so much faster than R and has written the excellent `MixedModels.jl` package, which was

used for the analyses reported in section 3.4, which readers can try out in the corresponding notebook in the Supplementary Materials. `Turing.jl` is a probabilistic programming package currently under active development but already excellently suited for running a rich variety of Bayesian statistical analyses. For any kind of analysis that is not yet available in Julia, you can call the requisite R package via `RCall.jl`, as just explained. See Nazarathy and Klok (2021) for more on statistics in Julia.

Optimization

There are a number of packages for optimization. Probably the best-developed and best-maintained of these is `JuMP.jl`. This package is an interface to a variety of different solvers, both open-source and proprietary ones. It does not provide those solvers itself, so they have to be installed independently. Which solver is best to use depends on the optimization problem at hand. For instance, the optimization problem considered in section 5.3 was nonlinear, for which one can use the Ipopt optimizer, a free and open-source optimizer that is automatically installed when one installs the `Ipopt.jl` package in Julia.

The `JuMP.jl` package offers a highly intuitive language for defining optimization problems: one defines the variables, the objective function to be optimized, constraints (if there are any), and then simply solves the resulting model, indicating the type of optimization (minimization, in our case) one is after. We illustrate the use of this package by showing how we found the minimal VS rule penalty for David's update on evidence E in section 5.3:

```
using JuMP
using Ipopt

model = Model(Ipopt.Optimizer)
@variable(model, 0 ≤ x[1:3] ≤ 1)
@NLobjective(
    model,
    Min,
    .01(.1(1 - x[1])^2 + .3x[2]^2 + .6x[3]^2) +
    .25(.45x[1]^2 + .1(1 - x[2])^2 + .45x[3]^2) +
    .1(.6x[1]^2 + .3x[2]^2 + .1(1 - x[3])^2)
)
@constraint(model, sum(x) == 1)
optimize!(model)
```

This actually gives a lot of output, which is why it is suppressed here. You may want to retrieve the minimum of the objective function specifically by entering `objective_value(model)`, and similarly for the values of the variables for which that minimum is obtained, which can be accessed by entering `value.(x)` and so on.

To ascertain that Bayes's rule does minimize expected RPS penalty (as we saw in the same section), replace the specification of the objective function in the preceding by

```
@NLobjective(
    model,
    Min,
    .01(((x[1] - 1)^2 +
         (x[1] + x[2] - 1)^2 +
         (x[1] + x[2] + x[3] - 1)^2)/2) +
    .25(((x[1] - 0)^2 +
         (x[1] + x[2] - 1)^2 +
         (x[1] + x[2] + x[3] - 1)^2)/2) +
     .1(((x[1] - 0)^2 +
         (x[1] + x[2] - 0)^2 +
         (x[1] + x[2] + x[3] - 1)^2)/2)
)
```

One section earlier in the same chapter it was shown that Bayesian updates are not necessarily more accurate than EXPL updates if accuracy is measured by a VS rule. This involved finding weights that, given David's probabilities, made his VS score come out lower if he updated by EXPL than if he updated by Bayes's rule. Interested readers can reproduce the result by using the code below. We first define variables containing the post-update probabilities for the two rules, for which we load the `LinearAlgebra.jl` package:

```
using LinearAlgebra

bayes = normalize!([.01, .25, .1], 1)
expl = normalize!([.11, .25, .1], 1)
```

Next we define the corresponding scoring functions:

```
function bayes_scr(w1, w2, w3, w4, w5)
    x = .01(w1*(1 - bayes[1])^2 + w2*bayes[2]^2 + w3*bayes[3]^2)
    y = .25(w5*bayes[1]^2 + w4*(1 - bayes[2])^2 + w5*bayes[3]^2)
    z = .10(w3*bayes[1]^2 + w2*bayes[2]^2 + w1*(1 - bayes[3])^2)
    return x + y + z
end

function expl_scr(w1, w2, w3, w4, w5)
    x = .01(w1*(1 - expl[1])^2 + w2*expl[2]^2 + w3*expl[3]^2)
    y = .25(w5*expl[1]^2 + w4*(1 - expl[2])^2 + w5*expl[3]^2)
    z = .10(w3*expl[1]^2 + w2*expl[2]^2 + w1*(1 - expl[3])^2)
    return x + y + z
end
```

And finally we define a function to loop through combinations of weights to find ones (if there are any, and perhaps if we are lucky) that make David incur a lower VS penalty for his EXPL update than for his Bayesian update:

```
function find_weights(iter)
    i = 1
    res = ()
    weights = Vector{Float64}(undef, 5)
    while i ≤ iter
        v = sort(rand(Dirichlet([1, 1, 1])))
        w = sort(rand(Dirichlet([1, 1])))
        weights = vcat(v, w)
        res = bayes_scr(weights...), expl_scr(weights...)
        if first(res) > last(res)
            break
        end
        i += 1
    end
    if first(res) > last(res)
        return res, weights
    else
        println("No weights found. Try again!")
    end
end
```

In my experiments, running `find_weights(1000)` was typically enough to yield weights of the sought-after kind. In other words, such weights exist, and apparently it is not difficult to find them.

If you are curious by *how much* EXPL can outperform Bayes's rule, in principle, we can again use the `JuMP.jl` package to find the weights that will maximize the difference between the EXPL-update score and the Bayesian-update score, with the former being lower than the latter. Specifically, the following chunk of code defines a function that takes weights as input and then outputs David's EXPL-update VS score (given the input weights) minus his Bayesian-update VS score. It then lets the computer look for the weights that minimize that function:

```
model = Model(Ipopt.Optimizer)
@variables(model, begin
    0 ≤ w[1:3] ≤ 1
    0 ≤ v[1:2] ≤ 1
end)
@NLobjective(
    model,
    Min,
    (.01(w[1]*(1 - expl[1])^2 +
         w[2]*expl[2]^2 +
         w[3]*expl[3]^2) +
     .25(v[1]*expl[1]^2 +
         v[2]*(1 - expl[2])^2 +
         v[1]*expl[3]^2) +
      .1(w[3]*expl[1]^2 +
         w[2]*expl[2]^2 +
         w[1]*(1 - expl[3])^2)) -
    (.01(w[1]*(1 - bayes[1])^2 +
         w[2]*bayes[2]^2 +
         w[3]*bayes[3]^2) +
     .25(v[1]*bayes[1]^2 +
         v[2]*(1 - bayes[2])^2 +
         v[1]*bayes[3]^2) +
      .1(w[3]*bayes[1]^2 +
         w[2]*bayes[2]^2 +
         w[1]*(1 - bayes[3])^2))
)
@constraints(model, begin
    sum(w) == 1
    sum(v) == 1
    w[1] ≤ w[2]
    w[2] ≤ w[3]
    v[2] ≤ v[1]
end)
optimize!(model)
```

Running `objective_value(model)` on this model shows that the EXPL-update VS score can at most be 0.001 lower than the Bayesian-update VS score; again, run `value.(x)` to obtain the weights that realize that difference in favor of EXPL.

Agent-Based Modeling

The simulations reported in chapter 7 relied heavily on agent-based models. Such models have become increasingly popular in philosophy and, to some extent, in cognitive science. Most colleagues working in this field use NetLogo (Wilensky & Rand, 2015), a language originally designed to introduce children to computer programming. In this language, the programmer basically tells turtles how to move and behave. Like Julia, NetLogo is free and open-source. The language has some other virtues as well (e.g., it is easy to learn, and it offers an attractive user interface). Nevertheless, I strongly recommend the use of Julia for agent-based modeling.

That is not just because you might be embarrassed to tell your friends that you program turtles for a living and would much prefer to say that you use this cool new language for high-performance computing that has been adopted by Google, IBM, NASA, Pfizer, and other high-end users. Most importantly, NetLogo is *extremely* slow. (What else can one expect from turtles?)

Users of NetLogo may tell you that the lack of speed is not really a problem, given that one can always move performance-critical parts of the code to C++, or FORTRAN, or some other compiled language. While that is true, that one has to take such an extra step is exactly the problem, known as "the two-language problem," that motivated the Julia developers. Julia can be used both for prototyping—designing your program—and for running the simulations. To repeat, it does so by offering a language that is easy to work with—as easy as NetLogo and Python—and yet is about as fast as C++.

Getting started with agent-based modeling in Julia has become particularly easy with the advent of the `Agents.jl` package (Vahdati, 2019), which requires from the user no more than a specification of agent and model properties and then takes care of all the rest (in particular, the running of the simulations and the gathering of relevant data). The appendix of Douven and Hegselmann (2021) contains links to various tutorials that help readers to get started with the package. It is to be noted, however, that the package is geared primarily toward ease of use and generality, and only then toward performance. And while it still allows one to write fast code, I have found it easier to write highly performant code by starting from scratch, without using the package. In particular, the code used for chapter 7 does not rely on `Agents.jl`. Still, the package offers an excellent starting point for new users.

Graphics

There is a variety of high-quality graphics packages for Julia. The figures in this book have all been produced using `Gadfly.jl`, but I have also had good experiences with `PlotlyJS.jl`, and the standard package `Plots.jl` yields publication-quality output as well. Nevertheless, I know of users who came to Julia from R and prefer to use the `ggplot` package via `RCall.jl`, as explained previously, and also of users who came from Python and have stuck with `pyplot`, which can be used in Julia thanks to `PyCall.jl`. And users with previous experience with `gnuplot` may want to have a look at the `Gaston.jl` and `Gnuplot.jl` packages. Probably, the most powerful visualization package for Julia is `Makie.jl`, which is under rapid development at the time of this writing but which at least for now offers a less than intuitive user interface (in my admittedly limited experience with the package).

Running the Simulations

I am making the code for the simulations available, because, first, as mentioned, I am a strong believer in a learning-by-doing approach, which in the present case means that I believe experimenting with the code to be the best and fastest way to obtain a good understanding of the simulations that were reported. In fact, because Julia code is relatively easy to understand, just reading it will deepen one's understanding of the simulations. To run the code, and later possibly alter it, first download it from the following public GitHub repository: https://github.com/IgorDouven/ABDUCTION.git. There are four Jupyter notebooks in the repository, which correspond to different parts of the book (as indicated in the titles of the notebooks), as well as the data from Douven and Schupbach (2015a), which are necessary to run the analysis reported in section 3.4. For instructions on how to work with Jupyter notebooks, see the documentation on this website: https://jupyter.org. Since very recently, you can also open, and work with, Jupyter notebooks in Visual Studio Code, a combination that I have found to offer a wonderful user experience.