A Very Short Introduction to Reproducible Research Using Julia for Economists

Benedikt Heid

April 21, 2022

1 What this document does

Julia is a multi-purpose programming language geared towards scientific computing. It is increasingly becoming an alternative to expensive proprietary software packages like Matlab. This is due to its many advantages (see Section 3). So why is not everyone using Julia already? One reason, particularly for longtime Matlab users may be that the entry cost for Julia, at least from my perspective, are higher than for Matlab. Installing Julia and the necessary editor to Get to run a simple "Hello world." script needs more setup effort than just opening up Matlab. One reason is that programming scripts in Julia is mainly done using a separate editor, typically VS Code, similar to how one uses a separate editor like, e.g. TeXnicCenter to write LaTeX files. VS Code is an excellent editor but it is quite complex in itself, so things can be a bit overwhelming for completely new users of VS Code and Julia (such as myself). Setting up reproducible research which includes package dependencies also is different from Matlab or Stata, so there are high entry costs which may deter new users.

While there are many websites available which explain many Julia details and intricacies, my own experience was that I did not find a **simple** tutorial that would enable me to just run a simple script file and which would highlight its differences to the software I was used to (mainly Matlab and Stata). Most introductions are too over the top or geared towards

actual programmers, while I was looking for an introduction for an empirical or quantitative economist. Many introductions also introduce Julia, Jupyter, and git at the same time.[1] While Jupyter is an excellent platform, it is not necessarily what people are looking for when they want to learn how to program in Julia and to create publication-ready replication files for their research projects. Finally, I do not think it is pedagogically the best solution to introduce people to interactive programming, version control and a new programming language at the same time. I prefer a one step at a time approach. There simply was no place which used such an approach to explain how VS Code and Julia work together and how to reproduce the behavior of Matlab.

This document tries to provide this introduction. **I am by no means an expert in Julia**, I am an applied economist, and I have written this document mainly to help myself. It therefore reflects my **incomplete** understanding of Julia. It is an evolving document, and I will update it as my own learning progresses. If you find this document helpful, too, all the better.

I provide `playground.zip` which contains a simple script file `plot_random_walk.jl` and the according environment `toml` files (see Section 8.2) on my website `https://benediktheid.weebly.com/` which illustrates the content of this document. It contains a script that plots a random walk and illustrates some basic functionality of Julia. Please note that this file is not optimized for speed and is not written in Julia style. It is written like a typical Matlab user would probably write the script file, and this is on purpose, as my goal is to replace Matlab by Julia. In the future, I may post more Julian examples.

If you find any typos or have any comments or questions, do not hesitate to contact me at heid@uji.es.

[1] Jupyter is a platform for interactive programming which lets you create notebooks which contain exectuable code as well as formatted text. git is an open-source version control system. I highly recommend its use for managing research projects, using a simple user interface like Sourcetree.

2 What this document does not do

This document cannot and does not want to replace a thorough, detailed introduction to Julia. Instead, this document should be seen as a starting point for previous users of Matlab (and Stata) who would like to transition from Matlab to Julia.

There are lots of helpful websites which give a much more detailed introduction to Julia:

- An excellent in depth introduction geared towards economists by the QuantEcon project can be found at `https://julia.quantecon.org/intro.html`.
- Julia is typically much faster than Matlab. Having said that, using programming style from Matlab may lead to code which is much slower than what Julia could achieve if you know a little bit more about issues such as types, type inference, and type propagation. Once you have been able to implement a first version of your research project, and you realize that you would benefit from speeding up calculations, have a look at `https://docs.julialang.org/en/v1/manual/performance-tips/` first and then at `https://julia.quantecon.org/software_engineering/need_for_speed.html`.
- If you do not use version control for your empirical research projects in economics, please do. For an introduction to open-source version control using git and Sourcetree, see, e.g., `http://economistry.com/2013/05/sourcetree-git-research/`.

3 Advantages of Julia

- It is free, hence will literally save you thousands of € every year. Your programming skills will also not depend on your employer financing your, e.g., annual Matlab subscription.

- According to several people who have done comparisons, it is much faster than software like MATLAB. In my own experience, JULIA is several orders of magnitude faster for solving, e.g., quantitative international trade models.
- It has been developed for scientific computing, so it is ideal for research projects.
- It allows parallel computing (even without specific programming and knowledge in parallel computing).
- It allows non-linear optimization, again for free.
- It allows the creation of replication packages for publication of research articles, including documenting the versions of the third-party packages you used for creating your results.

4 Differences to Matlab

- MATLAB comes with an integrated editor for `m` files which probably most users use to write their script files. MATLAB allows you to inspect the content of variables created by a script file in its workspace. It also has a built-in debugger. JULIA works differently. JULIA and VS CODE have the same relationship as the LATEX language (and a particular distribution such as MIKTEX) and a particular editor like TEXNICCENTER. To write `jl` files, you can use any editor you want, but for several reasons, I recommend VS CODE as editor for writing your Julia scripts.
- In MATLAB, you run a script by opening MATLAB, opening the script file and then clicking on `Run` (or pressing `F5`). The variables the script has created will appear in the workspace, and you can click on them and inspect them. You can also use the command prompt in Matlab to create variables on the fly, print variables, etc. This interactive use is an example of REPL (read-eval-print loop). The script is read, evaluated,

and its results are printed at the command prompt. JULIA provides the same REPL feature, see Section 6. In short, REPL is JULIA's name for its command prompt.

- Running a MATLAB `m` file will print out all lines of code which do not end with `;`. JULIA does not print variables; there is no need to use `;` to suppress output. To see the output of `x`, you have to use `show(x)`. To see figures created by `plot`, use `display(figurename)`.

- After running a file in MATLAB, you can click on created variables in the workspace within MATLAB. Double clicking `julia.exe` will give you access to the command line of JULIA for interactive use in a REPL fashion, but it looks basically like an old DOS prompt and there is no graphical user interface which would show you an editor, a variable editor, etc., so that is not how you would typically use JULIA. To get the graphical user interface, you need VS CODE. So to run a JULIA script, you typically run it from within VS CODE, similar to how you compile a `tex` file from within your text editor (even though you can, in principle, compile your tex file at the command prompt). VS CODE in combination with its JULIA extension provides the variable browser and the "look and feel" of what you are used from MATLAB.

- The built-in MATLAB editor also allows you to set break points to debug your code. In Julia, the same functionality is available via VS CODE.

- Not a difference, but worth pointing out: The elements in JULIA's arrays are indexed starting with 1, just like in MATLAB (but different to, e.g., PYTHON, where the first element is index 0, the second 1, etc. (so-called "0-indexed").

- A short overview of the differences between MATLAB and JULIA can be found at `https://docs.julialang.org/en/v1/manual/noteworthy-differences/#Noteworthy-differences-from-MATLAB`.

5 What to install to actually use Julia

1. Install the latest Julia version.
2. Install Visual Studio Code (VS Code).
3. Open VS Code, click on the Extensions button and install the Julia extension.

6 How to run your first script

Let us run a "Hello world." script.

- In VSCode, open a new file. Type

  ```
  "Hello world."
  ```

- Save the file under a name, using `jl` as the file extension.
- Click on the ▷ symbol in the top right corner of VS Code. This will "Julia: Execute active file in REPL". This is the equivalent to Matlab's `Run`.
- If you do not need to see any printed output or use the REPL but just want to run the script (for example because your script creates output files on your hard disk), you can also run the script by `Ctrl+F5`.

7 Debugging your code

One of the best features of Matlab is its debugger, which allows you to inspect the content of the workspace within functions to debug your code. Julia in combination with VS Code delivers the same functionality. To activate the debugger, click on the "Run and debug" button on the left in VS Code (a button which consists of the run symbol ▷ and a little bug) You can set breakpoints by clicking on the line number in your script.

You can then inspect the content of the variables, both local variables within functions as well as global variables. The debugger also comes with its own REPL, i.e., you can interactively use the content of the variables at the breakpoint. You can access this REPL by clicking on "Debug Console" on the bottom of VS Code.

8 Reproducible research in Julia

To publish in top journals, your research findings have to be reproducible by providing a "replication package". This means that all the data manipulation, starting from the raw data until the final data set, have to be traceable via the collection of script files, as well as all the statistical and econometric analysis. Also, your script files should be able to be executed on any computer, not only your own.

Backward induction implies that you should adjust your behavior to fulfill this requirement at the beginning of your research project, and not try to adjust your code after acceptance, only to realize that you cannot replicate your own research.

8.1 Making your research project reproducible: directories

- All data and code necessary for your research project should be located within a single project folder (with suitable subfolders). This folder and your code should be portable, i.e., you should be able to run your script to reproduce your research on different computers, independent on the specific directory structure on the machine you are running the code, just by handing over the folder to another user.

- To achieve this, you should not use hardcoded file paths (such as `C:/Users/mycomputer/myfiles/mynewproject`).[2] Instead, use relative file paths, i.e., only indicate files relative to the location of the

[2]By the way, when describing file paths, always use /, not \. \ is only used in Windows and will create problems when running your code on a different operating system. Always write filepaths with / only. Windows will also understand /.

current script you are running.

- To get the directory of the current script, use `@__DIR__` .
- You can then set this directory as the current directory:

```
cd(@__DIR__)
```

- You can then, e.g., write the array `x` into a file `newfile.csv` in the folder `data` which is contained in the current directory. (`.` indicates the current directory as a relative path):

```
open("./data/newfile.csv", "w") do io
   writedlm(io, x)
end
```

- Applying these recommendations will avoid drawing the ire of the AEA Data Editor, see the tweet by @AeaData on February 22, 2022, or `https://economistwritingeveryday.com/2022/02/26/how-to-set-working-directory-in-r-for-replication-packages/#more-6018`.

8.2 Making your research project reproducible: managing package dependencies via environments

In almost all cases, your research will rely on packages programmed by other users (e.g., packages which simulate random numbers, implement econometric estimators, etc.). As these packages get updated frequently, and they themselves rely on other packages (so-called dependencies), a key component of reproducible research is to include the packages and all their dependencies in a replication package.

In Stata, you can do this by installing packages into your project folder. By including this project-specific folder in a replication package, you can ensure that someone else can use the packages in the version you were using. This leads to duplicating the files of packages you use in separate projects.

Julia handles things differently. Instead of duplicating package files, Julia's package manager `pkg` is used. The collection of packages needed for

your research is called "environment". For each environment, the package manager creates two files, `Project.toml` and `Manifest.toml`. These two files contain a unique list of all the packages you use as well as the specific version numbers you are using, as well as all the packages they depend on, i.e., they contain a complete description of all dependencies. So with these two files, you get a complete description of the environment which created your results. So instead of including the files of the packages, you simply include the `toml` files in addition to your `jl` files in a replication package. The user of the replication package then can use `Pkg.instantiate()` to get all the necessary packages in the correct versions you used from the internet. This makes the replication package smaller. This means that, in addition to all your `jl` files, `Project.toml` and `Manifest.toml` should be included in a commit of a version control software.

To avoid inclusion of packages you do not really use, each of your separate research projects (papers) should have its own environment. You can create a new environment in the current directory by

```
using Pkg
Pkg.activate(".")
```

This activates the environment described in the files `Project.toml` and `Manifest.toml` in the current directory (or creates these files when called the first time). So your research project not only consists of all your `jl` files but also of `Project.toml` and `Manifest.toml`.

If you want to use a package for the first time, you have to download it from the internet and add it to Julia (this is similar to installing a package in Stata). You do this by

```
Pkg.add("packagename")
```

Similar as installing a package in Stata, you only have to add a package once. This means that to speed up the execution of your code, you can (and should) comment out this part of the file after having it run once.[3]

[3]You can also add packages interactively using the pkg's REPL. You can access this

There is a difference to packages in Stata. In Stata, once the package is installed, you can use all its commands in any do file. In Julia, if you want to use a command from a package, you have to "load" this package in your script file by

```
using packagename
```

Finally, there is conceptual difference between `Pkg.add` and `Pkg.instantiate`. `Pkg.add` adds the package for the first time, so you always have to do this once when you are writing new code for a new research project. This will then lead to a change in the `Project.toml` and `Manifest.toml` files. However, if you want to share your code with someone so that they can reproduce exactly the same results as you, you send them your `jl` files as well as the `toml` files. Then, if they `Pkg.activate(.)` and then `Pkg.instantiate()` the environment on their computer, Julia will automatically install (add) all necessary packages on their computer. So they do not have to run the `Pkg.add` commands, instantiating the project=environment on their computer is enough. The developer of the code, however, at one point, has to `Pkg.add` a package. Therefore, you will see many descriptions of how to use Julia which use the REPL of the package manager `pkg` interactively (you can enter it by entering `]`) at the Julia REPL. The prompt will change from green, indicating Julia's REPL, to blue. If you use an environment, this interactive adding of packages is okay to ensure replicability by others, as you can simply make available the `toml` files but I was really confused by this when trying to use Julia for the first time.

8.3 Environments and VS Code

Another issue which I found confusing at first is how environments interact with VS Code: VS Code shows the currently active environment in the bottom left corner as "Julia env: *environmentname*". If you work on different projects and different environments, you have to make sure that you are using

by typing] at the Julia prompt. I would advise against this interactive use as it is not reproducible, while using `Pkg.add` is. However, this can be circumvented by instantiating a project, see explanations below.

the correct environment. You do this in your `jl` file by `Pkg.activate(.)` it as explained above. Now, VS Code's environment is different from the environment of Julia's REPL.[4] Why does VS Code need or have a separate environment? It actually makes it more user-friendly by providing features such as code completion (IntelliSense in VS Code jargon). For this to work properly, VS Code has to know which packages are installed.

A simple way to ensure that both Julia and VS Code use the correct environment is that, in addition to activating it in your `jl` file, you should click on "Julia env: *environmentname*" and choose the folder which contains the environment.

As a side remark, note that different projects can use the same environment, there is nothing inherently wrong with this, simply activate the same environment. However, if you want to include in your replication package only references to the packages needed in your project (i.e., `Project.toml` and `Manifest.toml` only refer to these packages), you want to have seperate `Project.toml` and `Manifest.toml` files for each project, and hence a separate environment.

9 Useful commands

9.1 General commands

- To comment out a single line, use `#`. To comment out a block of text, enclose it in `#=...=#`.
- To plot a time series y, use `figurename = plot(y,label = "y")`. The label option labels the legend of the figure. *`figurename`* can be used to reference the figure, e.g., to save it as a pdf, see next bullet point.
- To save a figure as a pdf, use `savefig(`*`figurename`*`,"`*`filename`*`.pdf")`
- `include("`*`filename`*`.jl")` to include another script file and run it as if its code where copied into the script which contains the `include`

[4] See `https://discourse.julialang.org/t/how-to-properly-create-activate-and-resume-an-environment-in-vscode/48289/8` for a discussion.

statement. Helpful to split your larger projects into several smaller files.

- To see which version of Julia you are using, type `VERSION`.

9.2 Array manipulation

- Create a vector which contains a regular sequence of numbers:

  ```
  a=[1:1:4;]
  ```

  creates

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

9.3 Data input and output

- To write the content of two vectors vector1 and vector2 (of equal length) into a tab delimited file

  ```
  using DelimitedFiles
  open("filename.txt", "w") do io
      writedlm(io,[vector1 vector2])
  end
  ```