Introduction to Julia

Lecture 11 & 12: Getting started

Diego de Sousa Rodrigues
diego.desousarodrigues@sciencespo.fr

SciencesPo Paris

Preliminary stuff

Julia, VSCode and ATOM

- Install `Julia` : link here
- Install `VSCode` : link link
- Link `Julia` to `VSCode` : tutorial here
- Install `Atom` : link here
- Link `Julia` to `Atom` : link here

Julia

What and why ?

Computation has become and important tool in Economics :

1. Macro : solution of DSGE models, forecasting models, etc.
2. Micro : agent-based models, life-cycle models, etc.
3. Econometrics, trade and spacial econ, finance

Julia

What and why ?

Computation has become and important tool in Economics :

1. Macro : solution of DSGE models, forecasting models, etc.
2. Micro : agent-based models, life-cycle models, etc.
3. Econometrics, trade and spacial econ, finance

Computation often **complements**, rather that substitutes, theory :

- If theory shows that some partial derivative of interest is positive, computation can tell us how positive
- Importance of computation is likely to increase

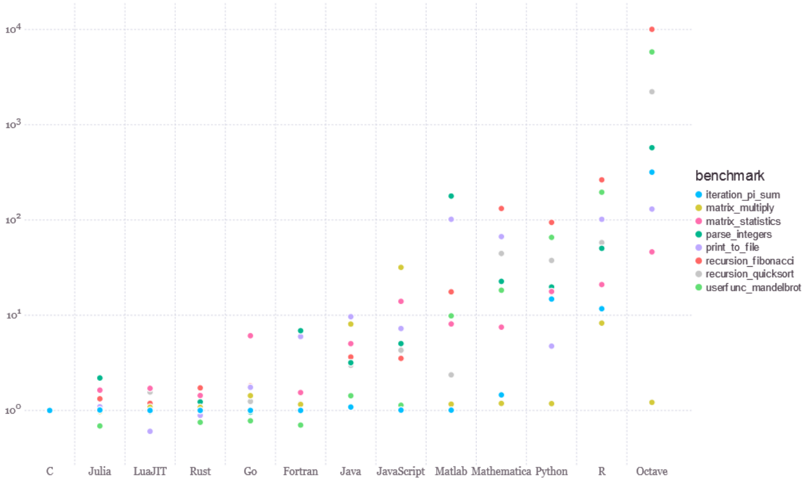
Julia

What and why ?

1. Modern language
2. Build for high-performance and parallel computing
3. Dynamically typed
4. Open source (!)
5. Many packages

Julia

What and why ?



VSCode

What and why ?

- Like Python and R, and unlike Matlab and Stata, there is **a looser connection** between Julia as a programming language and Julia as a development environment.

VSCode

What and why ?

- Like Python and R, and unlike Matlab and Stata, there is **a looser connection** between Julia as a programming language and Julia as a development environment.
- Because of this, you have much more flexibility in how you write and edit your code, whether that be locally or on the cloud, in a text-editor or IDE, etc.

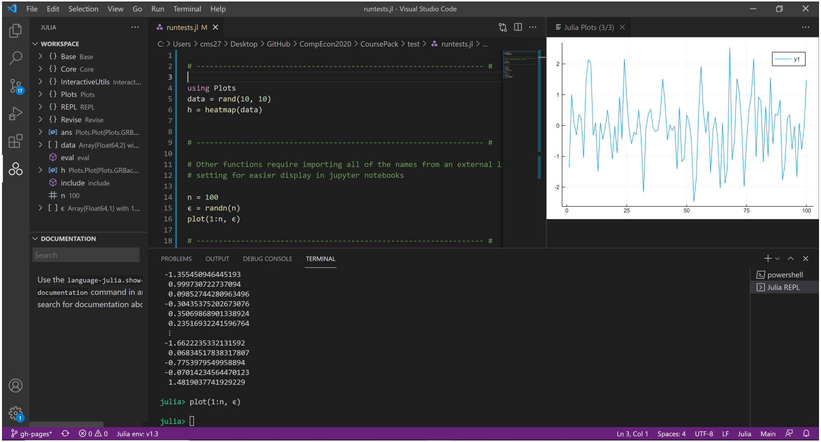
VSCode

What and why ?

- Like Python and R, and unlike Matlab and Stata, there is **a looser connection** between Julia as a programming language and Julia as a development environment.
- Because of this, you have much more flexibility in how you write and edit your code, whether that be locally or on the cloud, in a text-editor or IDE, etc.
- One example is **VSCode**, which provides a great interface to execute code in different languages (e.g. Julia, Python, R, etc.). Also : Jupyter notebook.

VSCode

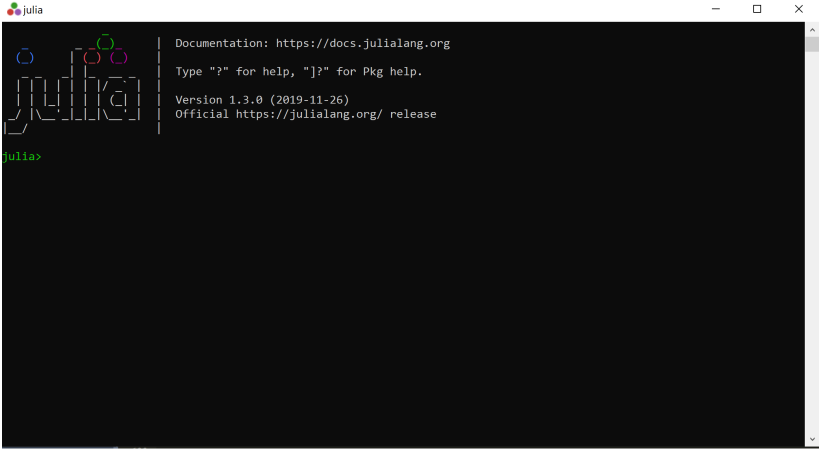
What and why ?



First steps

Julia REPL

The Julia REPL (Read-Evaluate-Print Loop, remember ?) is the first interface to deal with Julia



First steps

Julia REPL : **shell** mode

The Julia REPL (Read-Evaluate-Print Loop, remember ?) is the first interface to deal with Julia

- Hitting `;` brings you into shell mode, which lets you run bash commands (PowerShell on Windows)
- Example : try to type `; cd`
- To go back to standard mode, type `CTRL + c`

First steps

Julia REPL : **package** mode

The Julia REPL (Read-Evaluate-Print Loop, remember ?) is the first interface to deal with Julia

- Hitting `]` brings you into package mode
- `] add Expectations` will add a package (here, `Expectations.jl`)
- Likewise, `] rm Expectations` will remove that package
- `] st` will show you a snapshot of what you have installed
- `] up` will (intelligently) upgrade versions of your packages
- `] precompile` will precompile everytihng possible
- Everything you dream to know about the package mode : `] ?`

First steps

Julia REPL : **hlep** mode

The Julia REPL (Read-Evaluate-Print Loop, remember ?) is the first interface to deal with Julia

- Hitting `?` brings you into help mode
- The key use case is to find docstrings for functions and macros, e.g. `? print`
- Note that objects must be loaded for Julia to return their documentation, e.g. `? @test`

First steps

Package Environments

- Julia's package manager lets you set up Python-style "virtualenvs," or subsets of packages that draw from an underlying pool of assets on the machine = an **environment**
- This way, you can work with (and specify) the dependencies (i.e., required packages) for one project without worrying about impacts on other projects.
- E.g., project A may depend on packages `xyz` while project B may depend on an older version of `xyz`
- An **environment** is a set of packages specified by a `Project.toml` (and optionally, a `Manifest.toml`)

First steps

Package Environments

- You can check the status of your current (default) **environment** by typing : `] st`

First steps

Package Environments

- You can also create a new **environment** by typing : `] generate ExEnv`
- Then go to it `; cd ExEnv`
- Finally, activate the new **environment** by typing : `] activate .`
- You can now add and remove packages that will be unique for this new **environment**

First steps

Package Environments

- To go back to your default environment, simply type `] activate`
- Remove the example environment by typing `; cd ..` and then `rm -rf ExEnv` (only on Linux / MacOS)

Learning Julia !

- We are now ready to start learning **Julia** !
- Our approach is aimed at those who already have at least some knowledge of programming (R, Python, MATLAB, etc.)
- I assume you have some familiarity with fundamental programming concepts such as `variables, arrays or vectors, loops, conditionals (if/else)`

Learning Julia !

Approach

- In this lecture we will write and then pick apart small Julia programs
- At this stage the objective is to introduce you to basic syntax and data structures
- Deeper concepts—how things work—will be covered in later lectures
- Since we are looking for simplicity the examples are a little contrived

Learning Julia !

Example 1 : Plotting a White Noise Process

To begin, let's suppose that we want to simulate and plot the white noise process, where each draw is independent standard normal

1. activate a project environment, which is encapsulated by `Project.toml` and `Manifest.toml` files by typing : `] activate`
2. add packages you need with `] add` e.g. `] add MyPackage` (which is similar to `Pkg.add("MyPackage")`)
3. after the installation and activation, `using` provides a way to say that a particular code or notebook will use the package :

   ```
   using LinearAlgebra, Statistics, Compat
   ```

Learning Julia !

Example 1 : Plotting a White Noise Process

Some functions are built into the base Julia, such as `randn`, which returns a single draw from a normal distibution with mean 0 and variance 1 if given no parameters

- E.g. type `rand()`, what do you get ?
- Now use the package `Plots` to plot the result of 100 realizations of `randn()`

Learning Julia !

Example 1 : Plotting a White Noise Process

Some functions are built into the base Julia, such as `randn`, which returns a single draw from a normal distibution with mean 0 and variance 1 if given no parameters

- E.g. type `rand()`, what do you get ?
- Now use the package `Plots` to plot the result of 100 realizations of `randn()`
- Solution :

```
using Plots
n = 100
ϵ = randn(n)
plot(1:n, ϵ)
```

Learning Julia !

Example 1 : Plotting a White Noise Process

Let's break this down and see how it works :

- The effect of the statement `using Plots` is to make all the names exported by the Plots module available
- Because we used `] activate` previously, it will use whatever version of `Plots.jl` that was specified in the `Project.toml` and `Manifest.toml` files
- The other packages `LinearAlgebra` and `Statistics` are base Julia libraries, but require an explicit using
- The arguments to plot are the numbers `1,2, ..., n` for the x-axis, a `vector` ϵ for the y-axis, and (optional) settings
- The function `randn(n)` returns a column vector n random draws from a normal distribution with mean 0 and variance 1

Learning Julia !

Example 1 : Plotting a White Noise Process

- As a language intended for mathematical and scientific computing, Julia has strong support for using unicode characters
- In the above case, the ϵ and many other symbols can be typed in most Julia editor by providing the LaTeX and `<TAB>`, i.e. `\epsilon<TAB>`
- The return type is one of the most fundamental Julia data types : an array. Try `typeof(ϵ)`, what do you get ? And `ϵ[1:5]` ?

Learning Julia !

Example 1 : Plotting a White Noise Process

- The information from `typeof()` tells us that ϵ is an array of 64 bit floating point values, of dimension 1
- In Julia, one-dimensional arrays are interpreted as column vectors for purposes of linear algebra
- Notice from the above that (i) array indices start at 1 (like MATLAB, but unlike Python and C) and (ii) array elements are referenced using square brackets
- Don't forget you can always get help by typing `?` before a function name of synthax

Learning Julia !

Example 1 : Plotting a White Noise Process

Exercise 1 : your first loop !

- Although there's no need in terms of what we wanted to achieve with our program, for the sake of learning syntax let's rewrite our program to use a for loop for generating the data
- Starting with the most direct version, and pretending we are in a world where `randn()` can only return a single value
- Use the syntax `for i in 1:X` to write a loop that iteratively fill a vector ϵ with values

Learning Julia !

Example 1 : Plotting a White Noise Process

Exercise 1 : solution

```julia
# poor style
n = 100
ϵ = zeros(n)

for i in 1:1000
    ϵ[i] = randn()
end
plot(1:n, ϵ)

# better style
n = 10
ϵ = zeros(n)
for i in eachindex(ϵ)
    ϵ[i] = randn()
end
```

Learning Julia !

Example 1 : Plotting a White Noise Process

Exercise 1 : solution (explained)

- Here we first declared ϵ to be a vector of n numbers, initialized by the floating point 0.0
- The for loop then populates this array by successive calls to `randn()`
- Like all code blocks in Julia, the end of the for loop code block (which is just one line here) is indicated by the keyword end
- The word `in` from the `for` loop can be replaced by either $\in$ or `=`
- The index variable is looped over for all integers from `1 : n` – but this does not actually create a vector of those indices
- Instead, it creates an iterator that is looped over – in this case the range of integers from 1 to n
- While this example successfully fills in ϵ with the correct values, it is very indirect as the connection between the index i and the ϵ vector is unclear
- To fix this, use `eachindex`

Learning Julia !

Example 1 : Plotting a White Noise Process

Exercise 1 : solution (explained)

- Here, `eachindex(`ϵ`)` returns an iterator of indices which can be used to access ϵ
- While iterators are memory efficient because the elements are generated on the fly rather than stored in memory, the main benefit is (1) it can lead to code which is clearer and less prone to typos ; and (2) it allows the compiler flexibility to creatively generate fast code
- In Julia you can also loop directly over arrays themselves !

Learning Julia !

Example 1 : Plotting a White Noise Process

Exercise 2 : Loops over an array

- Define the variable `ϵ_sum` equals to 0
- We want to compute the cumulative sum of the 5 first elements of the array ϵ using a loop and store it into `ϵ_sum`
- Compute `ϵ_mean`, the average value of those 5 first elements

$\Rightarrow$ Let's try (5')

Learning Julia !

Example 1 : Plotting a White Noise Process

Exercise 2 : solution on Slack !

Learning Julia !

Example 1 : Plotting a White Noise Process

Exercise 2 : solution (easier)

- Easier solution (built-in functions) : `ϵ_mean = mean(ϵ[1:m])`
- You can directly test proposition with the symbol : ≈ (\approx<TAB>)

  `ϵ_mean ≈ sum(ϵ[1:m])/m`
- In this example, note the use of ≈ to test equality, rather than ==, which is appropriate for integers and other types

Learning Julia !

Example 2 : User define function

- For the sake of the exercise, let's go back to the for loop but restructure our program so that generation of random variables takes place within a user-defined function
- To make things more interesting, instead of directly plotting the draws from the distribution, let's plot the squares of these draws

Learning Julia !

Example 2 : User define function

Exercise 3 : your first function

- use the keywords `function`, `return` and `end` to encapsulate the `for` loop
- the name of the function will be `generatedata()` and takes one argument : `n` which is the size of the array ϵ
- Remember : we want to plot the squares of these draws

Learning Julia !

Example 2 : User define function

Exercise 3 : solution

- `function` is a Julia keyword that indicates the start of a function definition
- `generatedata()` is an arbitrary name for the function ϵ
- `return` is a keyword indicating the return value, as is often unnecessary

Learning Julia !

Example 2 : User define function with broadcasting

- The looping over the i index to square the results may be difficult to read
- Instead of looping, we can broadcast the `^2` square function over a vector using a `.`
- To be clear, unlike Python, R, and MATLAB, the reason to drop the `for` is not for performance reasons, but rather because of code clarity
- Loops of this sort are at least as efficient as vectorized approach in compiled languages like Julia, so use a for loop if you think it makes the code more clear

Learning Julia !

Example 2 : User define function with broadcasting

- We can even drop the `function` if we define it on a single line

```
generatedata(n) = randn(n).^2
```

Learning Julia !

Example 2 : User define function with broadcasting

- We can broadcast any function (not only ^)
- We can broadcast your own user-defined functions as well !

Learning Julia !

Example 2 : User define function with broadcasting

Exercise 4 : broadcast your own function

- Create your own function `f(x)` that returns the square value of `x`
- Modify `generatedata()` to broadcast `f()` (instead of ^)

Learning Julia !

Example 3 : A slightly more useful function

- This function will be passed in a choice of probability distribution and respond by plotting a histogram of observations
- In doing so we'll make use of the `Distributions` package, which we assume was initiated above with the project

Learning Julia !

Example 3 : A slightly more useful function

- This function will be passed in a choice of probability distribution and respond by plotting a histogram of observations
- In doing so we'll make use of the `Distributions` package, which we assume was initiated above with the project
- Here's the code :

```julia
# Exemple 3
using Distributions

function plothistogram(distribution, n)
    ϵ = rand(distribution, n)  # n draws from distribution
    histogram(ϵ)
end

lp = Laplace()  # an "instance" of a Distribution type. More on this later!
plothistogram(lp, 500)
```

Learning Julia !

Example 3 : A slightly more useful function

- `lp = Laplace()` creates an instance of a data type defined in the Distributions module that represents the Laplace distribution
- When we make the function call `plothistogram(lp, 500)` the code in the body of the function `plothistogram` is run with (i) the name `distribution` bound to the same value as `lp` and (ii) the name `n` bound to the integer `500`

Learning Julia !

Final Example : Variations on Fixed Points

- Reminder : take a mapping $f : X \to X$ from some set X. If there exists $x^* \in X$ s.t. $f(x^*) = x^*$ then x^* is called a fixed point.
- In this example, we will try to determine fixed points of a function, going from a MATLAB style to a Julian style.
- Consider the simple equation $v = p + \beta v$. The problem rewrite $v = f(v)$ with $f(v) = p + \beta v$

Learning Julia !

Final Example : Variations on Fixed Points

- One approach to finding a fixed point is to start with an initial value, and iterate the map : $v^{n+1} = f(v^n)$ that converges for this exact function if $\beta < 1$

Learning Julia !

Final Example : Variations on Fixed Points

```julia
p = 1.0 # note 1.0 rather than 1
β = 0.9
maxiter = 1000
tolerance = 1.0E-7
v_iv = 0.8 # initial condition

# setup the algorithm
v_old = v_iv
normdiff = Inf
iter = 1
while normdiff > tolerance && iter <= maxiter
    v_new = p + β * v_old # the f(v) map
    normdiff = norm(v_new - v_old)  # "size" of a vector in some space

    # replace and continue
    global v_old = v_new
    global iter = iter + 1
end

println("Fixed point = $v_old, and |f(x) - x| = $normdiff in $iter iterations")
```

Learning Julia !

Final Example : Variations on Fixed Points

`LinearAlgebra.norm` — Function

```
norm(A, p::Real=2)
```

For any iterable container `A` (including arrays of any dimension) of numbers (or any element type for which `norm` is defined), compute the `p`-norm (defaulting to `p=2`) as if `A` were a vector of the corresponding length.

The `p`-norm is defined as

$$\|A\|_p = \left(\sum_{i=1}^{n} |a_i|^p \right)^{1/p}$$

with a_i the entries of A, $|a_i|$ the `norm` of a_i, and n the length of A. Since the `p`-norm is computed using the `norm`s of the entries of `A`, the `p`-norm of a vector of vectors is not compatible with the interpretation of it as a block vector in general if `p != 2`.

`p` can assume any numeric value (even though not all values produce a mathematically valid vector norm). In particular, `norm(A, Inf)` returns the largest value in `abs.(A)`, whereas `norm(A, -Inf)` returns the smallest. If `A` is a matrix and `p=2`, then this is equivalent to the Frobenius norm.

The second argument `p` is not necessarily a part of the interface for `norm`, i.e. a custom type may only implement `norm(A)` without second argument.

Learning Julia !

Final Example : Variations on Fixed Points

Exercise 5 : Improve this code !

1. use a `for` loop instead a `while`
2. create a user define `function` to encapsulate the loop

Learning Julia !

Final Example : Variations on Fixed Points

```julia
# With a loop:

p = 1.0 # note 1.0 rather than 1
β = 0.9
maxiter = 1000
tolerance = 1.0E-7
v_iv = 0.8 # initial condition

# setup the algorithm

v_old = v_iv
normdiff = Inf
iter = 1
for i in 1:maxiter
    v_new = p + β * v_old # the f(v) map
    normdiff = norm(v_new - v_old)
    if normdiff < tolerance # check convergence
        iter = i
        break # converged, exit loop
    end

    # replace and continue
    global iter = iter + 1
    global v_old = v_new
end
println("Fixed point = $v_old, and |f(x) - x| = $normdiff in $iter iterations")
```

Learning Julia !

Final Example : Variations on Fixed Points

```julia
# good style
function fixedpointmap(f; iv, tolerance=1E-7, maxiter=1000)
    # setup the algorithm
    x_old = iv
    normdiff = Inf
    iter = 1
    while normdiff > tolerance && iter <= maxiter
        x_new = f(x_old) # use the passed in map
        normdiff = norm(x_new - x_old)
        x_old = x_new
        iter = iter + 1
    end
    return (value = x_old, normdiff=normdiff, iter=iter) # A named tuple
end

# define a map and parameters
p = 1.0
β = 0.9
f(v) = p + β * v # note that p and β are used in the function!

sol = fixedpointmap(f, iv=0.8, tolerance=1.0E-8) # don't need to pass
println("Fixed point = $(sol.value), and |f(x) - x| = $(sol.normdiff) in $(sol.iter)"*" iterations")
```