BETA Bureau d'économie théorique et appliquée

AgroParisTech
Talents d'une planète soutenable

Scientific Programming ~~and Machine Learning~~ with Julia

Kick-off meeting

Antonello Lobianco

"A language that doesn't affect the way you think about programming is not worth knowing."

-- Alan Perlis(American computer scientist)

BETA Bureau d'économie théorique et appliquée
AgroParisTech Talents d'une planète soutenable

Today's objectives

- **What this course is about**
- **How this course is organised**
- **Who am I ? (Who are you ?)**
- **What is Julia ? (Why to bother with yet-an-other language ?)**
- **Let's set up the environment for the course !**
  - **diff, git, github**
  - **computational environment**
- **Let's start by learning to work with packages**

This course

- A brief but solid introduction to the Julia Programming Language
  - General syntax and programming constructs (packages, data types, control flows, macros, I/O,...˚
  - "scientific" programming: wrangling and visualising data, optimisation problems, statistics, ...
- The course follows the following references :
  - The book "Julia Quick Syntax Reference" (Apress 2019) for the Julia part (PDF available on the course site – to not redistribute, only for you !)
- Course on Arche: https://arche.univ-lorraine.fr/course/view.php?id=53796
- Course on GitHub: https://github.com/sylvaticus/SPMLJ

Course organisation

- This kick-off meeting
- 2 additional recorded video "lessons":
  - Basic Julia programming
  - Scientific programming with Julia
- ~~One Lesson == One week~~ One Lesson != One week
  - Videos and related exercises open each week and are due for the ending of the week (see calendar for details)
  - You can either watch the video or just play by yourself with the associated Julia script or study them in the "compiled form" online
- The computational environment
  - The best option: you install the computational environment on your own PC (one reason why we have a kick-off meeting)
  - The fallback: a server is available for you for this course
    - could be slow
    - documents will not remain available after the course end
  - Other online computational environments with Julia support: https://cocalc.com, https://nextjournal.com, https://notebooks.gesis.org, https://codeocean.com, https://visualstudio.microsoft.com/services/github-codespaces , https://replit.com/languages/julia

BETA Bureau d'économie théorique et appliquée

AgroParisTech
Talents d'une planète soutenable

Let's present us

- I'm a research engineer in Forest and Environmental Economics at BETA (Bureau d'économie théorique et appliquée) Nancy (UMR AgroParisTech, INRAE, University of Lorraine, University of Strasbourg, CNRS)
- My work involves a lot of modelling and simulations
  - agent-based farm-level to regional simulations → RegMAS (Regional Multi Agent Simulator)
  - bio-economic models of the forest sector (forest dynamics, timber markets equilibrium, carbon cycle) → FFSM (French Forest Sector Model)
- My pathway with programming:
  - Commodore 16, Commodore64 Basic, DOS, VBA, Pascal, Perl, PHP, C++, Python, Julia
  - I have experience also with setting LAMP servers/middleware
  - What I used this for? Everything from management applications (including the students management office of my university), web sites, numerical utilities.. research
- And you ?
  - Which programming languages do you use (let me guess.. R ? Python? Maybe javascript or Matlab ?)
  - What's your background ?
  - Have you already used ML algorithms ?
  - What are you looking / expect from this course ?

JULIA

BETA Bureau d'économie théorique et appliquée
AgroParisTech Talents d'une planète soutenable

A comparative timeline of R, Python, Julia, LLVM

- 1991: Python announced
- 1993: R announced
- 1994: Python reaches v1.0
- 2000: R reaches v1.0
- 2003: LLVM announced (v1.0)
- 2012: Julia announced by a MIT team to solve the "two languages problem"
- 2018: Julia reaches v1.0

JuPyteR notebooks

JIT compiled
- R: `compiler` library (JIT default in 3.4)
- Python: Numba, PyPy, ...
- Julia: natively, with type inference

BETA Bureau d'économie théorique et appliquée
AgroParisTech
Talents d'une planète soutenable

Julia: multiple dispatch

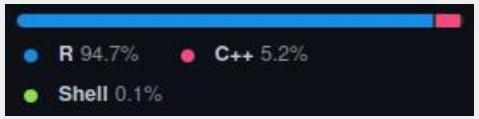
- dispatch the call to a function to the most specific version ("method") based on the types of all the arguments
- is a generalisation of OO single argument dispatch obj.foo(arg2, arg3) → foo(obj,arg2,arg3)
- not to be confused with C++ -style polymorphism (at compile time)
- simplify code composition: as soon new types know how to handle low level functionalities, they can reuse high level one

- Julia libraries
  - DataFrames:

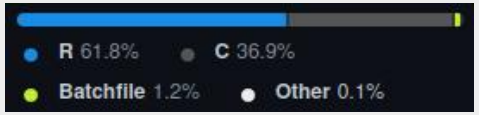
  - Flux:

  - BetaML:

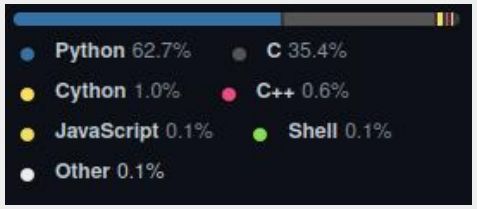
- R libraries
  - Dplyr:



  - data.table:

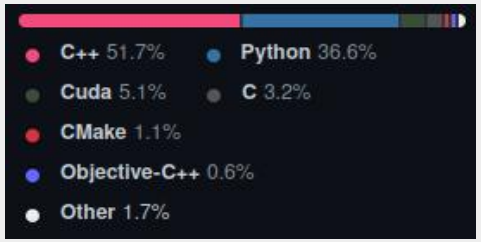


- Python libraries:
  - Numpy:



  - Pandas:

  - Pytorch:



Multiple kinds of parallel computation

Hardware level SIMD (single instruction multiple data), GPU

- `@simd for i in …`
- CUDA.jl

Multithreading

- `Threads.@threads for i in …`

Multiprocessing

- `addprocs(n, ssh parameters)` same machine or other HPC nodes using SSH tunnelling
- `@everywhere function foo(arg) …`
- `pmap(foo,data)`

Other characteristics of Julia for Scientific Programming

- Macro and meta-programming: at the level of the Abstract Syntax Tree
  - allow more flexible APIs (compare those of JuMP with the equivalent of PYOMO)

- Extended Unicode support on any identifier: `julia> σ̃² = 1` `julia> x̄₂ ∈ [1,2,π,e]`

- Support to user-defined primitive type (need a Int24 type ??), even basic operators are extendible: `+(x,y::Int24) = ….`

- Linear algebra built in: `x' * Y^(-1)`

- Broadcasting support to any function (including user-defined): foo.(args) (note the dot)

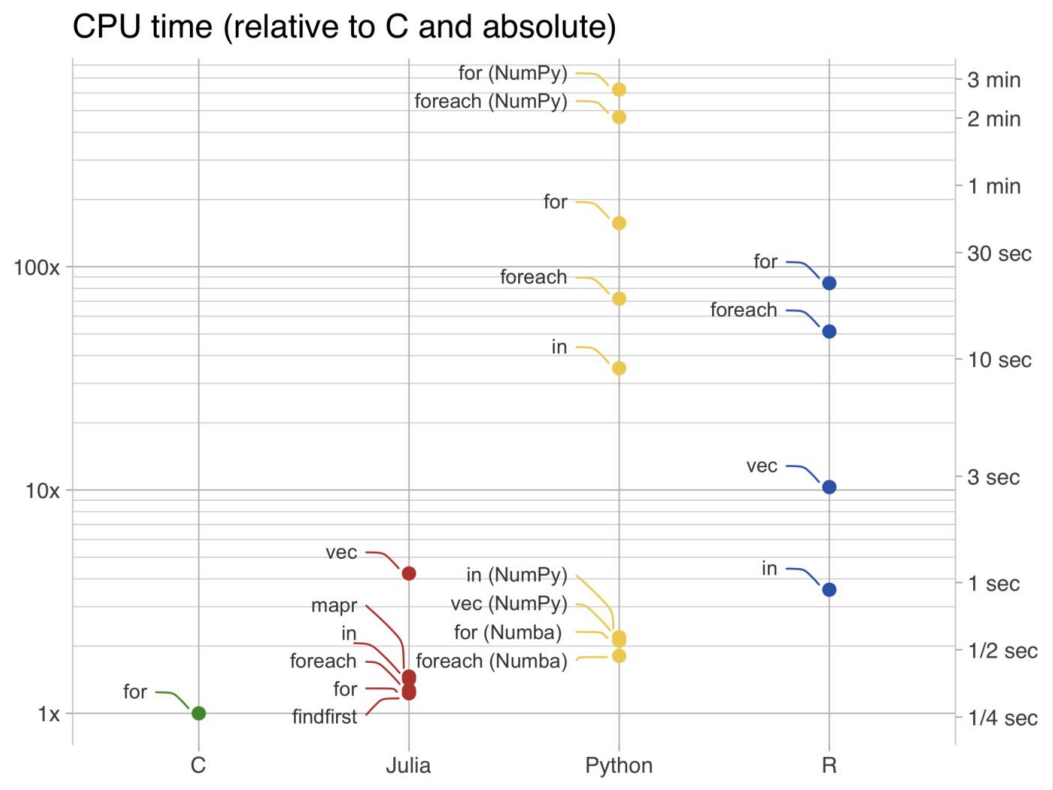
- Direct interoperability with C/Fortran libraries ccall(:foo,foolib,...) and VERY easy integration with R and Python

- Modern testing and documentation facilities (Test module – in the st.lib -, Literate.jl, Documenter.jl, integration with GitHub actions and CodeCov)

- Light-wise git-based environments for easy replicability

When Julia is no (computational) advantage over R, Python, Matlab...

- when the bottleneck is given my some matrix operation (multiplication, inverse,...)
  - it resolves to the underlying BLAS/LAPACK library emploied
  - still in Julia >= 1.7 very easy to switch to MLK if needed
- when the bottleneck of your code is in using some highly-optimised C/Fortran libraries with a R/Python wrapper (including vectorised ops with numpy)

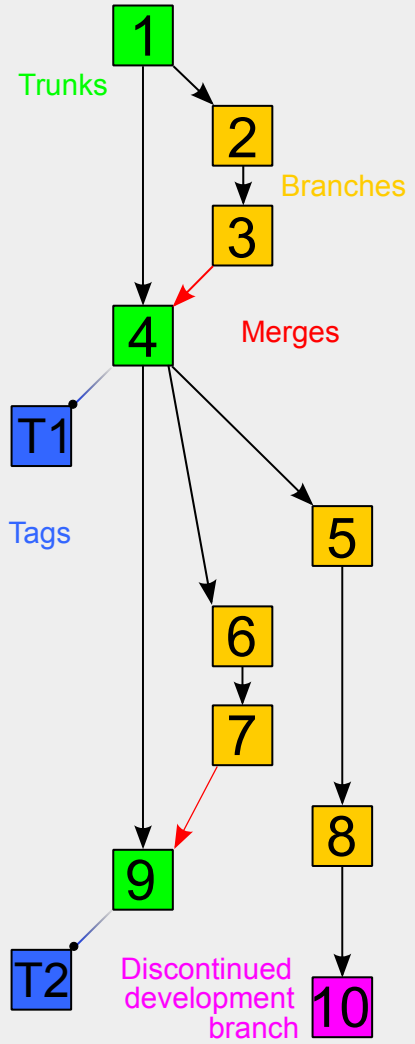


https://towardsdatascience.com/r-vs-python-vs-julia-90456a2bcbab

HANDS ON

Version control software

- Used to keep track of the development of a project and allow easy integration of multiple sub-projects or team contributions
- Typically they allow assigning a name/log entry to a given "screenshot in time" of project development, visualise diffs between them, create and merge branches of different development pathways, automatically manage or facilitate resolution of conflicts between versions,...
- First generation (CVS, SVN,...):
  - client-server model → the full project (history) is kept in a centralised server and individual users connect and keep in their pc one specific version
- Second generation (Git, Mercurial, Bazaar,...):
  - distributed model → each user has the whole copy (including history) of the project and exchanges its contribution with the other "remote" nodes
  - increases redundancy
  - decreases network load
  - faster operations (many operations – e.g. diff – can be fully done in local)



Set up the environment

- Create our first git repository
  - register or sign in on github.com
  - add new repository "testGit" (attention capital letters! no spaces!)
  - add readme, gitignore (Julia) and licence (MIT? GPL?)
- Install Julia (terminal): https://julialang.org/ → download → 1.7 64 bit installer → run with the option "add julia to path"
- Install Git for Windows: https://git-scm.com/download/win Check the options !
- Install Visual Studio Code: https://code.visualstudio.com/ → download → run with the options additional icons/other
- Install Julia extension:
  - search in extensions: "julia" (not "julia insider")
  - try it is working: new file -> select a language -> julia -> println("Hello World!") SHIFT+ENTER

In case of impossibility to set up the environment on your own pc an account on the lab server at https://lef-nancy.org has been set for you (username: your surname all in lower letters, psw [XXXXXX]

Help: this course forum (for those with access to it), https://discourse.julialang.org/ , StackOverflow (using the "julia" tag)

Essential git workflow

- Clone an existing repositories or start a new one: `git clone [url]` or `git init`
- ..work on your project..
- Add files: `git add [file]`
- Commit your work: `git commit -a -m ‘‘[message]’’`
- See branches: `git branch`
- Create branch: `git branch [branchname]`
- Switch to branch: `git checkout [branchname]`
- Merge a branch: `git merge [branchname]`
- Delete a local branch: `git branch -d [branchname]`
- Fetch a remote repository (without attempt to update the local repository): `git fetch`
- Fetch and try to merge from a remote repository: `git pull`
- Push your data to a remote repository: `git push`
- List all the remote repositories: `git remote`
- Get current repository status: `git status`
- Get git logs: `git log`
- See diff with HEAD: `git diff`
- See diff between any commit(s): `git diff [commithash1] [commithash2]`

Further resources
Git tutorials (shorts):
- https://git-scm.com/docs/gittutorial
- https://thenewstack.io/tutorial-git-for-absolutely-everyone

Git book (long):
- https://git-scm.com/book

A first look on git(hub)
- retrieve the repository locally

- Clone the repository locally using Visual Code Studio
  - Automatic way
    - CTRL+SHIFT+P to open the command palette and search for git:clone
    - Clone from github
    - Authorise
    - Choose the repository to clone
    - Open in new file
  - Manual way
    - open a new "git bash" terminal using "new terminal" and then the "plus"
    - git clone https://github.com/[YOUR GITHUB USERNAME]/testGit.git
    - open folder This PC > OS (C:) > Users > Your name > testGit

A first look on git(hub)

- work, commit & push

- Add a file "test.jl" → `println("Hello World!")` → CTRL+ENTER → save
- Edit the README.md file
- `git status`
- `git add test.jl`
- `git status`
- `git config --global user.email "you@example.com"`
- `git config --global user.name "Your Name"`
- `git commit -a` → add a message → CTR+X → Y
- `git push` → follow online instruction to authorise
- => changes are back to github

BETA Bureau d'économie théorique et appliquée
AgroParisTech Talents d'une planète soutenable

A first look on git(hub)

- working with branches

- Edit file "test.jl" to add text on lines 4, 7 and 9, save and commit
- Open a git bash terminal
- git branch temp
- git checkout temp
- Edit test.jl on line 9, save and commit
- git checkout main
- Edit test.jl on line 5, save and commit
- git diff main temp
- git branch
- git merge temp
- git log
- git push
- git branch -d temp

Modules and packages

Structure of a module:

```
module ModuleName
export myObjects # functions, structs, and other objects that will be directly available once `using ModuleName` is typed
[...module code...]
end
```

- `using pkgOrModule`
- `import pkgOrModule: X,Y,Z`
- Module names are customary starting with a capital letter and the module content is usually not indented. Modules can be entered in the REPL as normal Julia code or in a script that is imported with `include("file.jl")`.
- There is no connection between a given file and a given module as in other languages, so the logical structure of a program can be decoupled from its actual division in files. For example, one file could contain multiple modules.
- A more common way to use modules is by loading a package that will consists, at least, of a module with the same name of the package.
- All modules are children of the module `Main`, the default module for global objects in Julia, and each module defines its own set of global names.
- Note: using and import, when they are followed with either Main.x or .x, look for a module already loaded and bring it and its exported objects into scope (for `import` only those explicitly specified). Otherwise, they do a completely different job: they expects a *package*, and the package system lookups the correct version of the module x embedded inside package x, it loads it, and it bring it and its exported objects into scope (again, for `import x` only those explicitly specified).

https://syl1.gitbook.io/julia-language-a-concise-tutorial/language-core/11-developing-julia-packages

The integrated Pkg manager

```
julia> using Pkg

julia> Pkg.add("DataFrames")
   Resolving package versions...
  No Changes to `~/.julia/environments/v1.6/Project.toml`
  No Changes to `~/.julia/environments/v1.6/Manifest.toml`

julia>
```

```
(@v1.6) pkg> add DataFrames
    Updating registry at `~/.julia/registries/General`
   Resolving package versions...
  No Changes to `~/.julia/environments/v1.6/Project.toml`
  No Changes to `~/.julia/environments/v1.6/Manifest.toml`
  Progress [========================================>]  3/3
3 dependencies successfully precompiled in 44 seconds (366 already precompiled)

(@v1.6) pkg>
```

Some useful package commands:

- `status` Retrieves a list with name and versions of locally installed packages
- `update` Updates your local index of packages and all your local packages to the latest version
- `add myPkg` Automatically downloads and installs a package
- `rm myPkg` Removes a package and all its dependent packages that has been installed automatically only for it
- `add pkgName#master` Checkouts the master branch of a package (and `free pkgName` returns to the released version)
- `add pkgName#branchName` Checkout a specific branch
- `add git@github.com:userName/pkgName.jl.git` Checkout a non registered pkg
- `free pkgName` Return to the "standard" latest compatible released version of a package

https://syl1.gitbook.io/julia-language-a-concise-tutorial/language-core/11-developing-julia-packages

Julia environments

- Packages themselves are downloaded, "installed" (precompiled) and stored in a user common directory (e.g. `/home/[username]/.julia` in Linux)
- Each project can (should!) have its own "environment"
- This is really simple and "light", as it is a simple, automatically managed file ("Manifest.toml") that list all the packages and sub-packages used in the project and their versions
  - providing this file to the "customer"/book, together with the source code, the set of inputs and using a fixed seed for the random numbers at the beginning of the script(s) guarantee reproducible analysis/research

- `cd(@__DIR__)`
- `using Pkg`
- `Pkg.activate(".")`
- `Pkg.instantiate()`

An Example of Manifest file

```
# This file is machine-generated - editing it directly is not advised

[[ArgTools]]
uuid = "0dad84c5-d112-42e6-8d28-ef12dabb789f"

[[Artifacts]]
uuid = "56f22d72-fd6d-98f1-02f0-08ddc0907c33"

[[Base64]]
uuid = "2a0f44e3-6c83-55bd-87e4-b1978d98bd5f"

[[Compat]]
deps = ["Base64", "Dates", "DelimitedFiles", "Distributed", "InteractiveUtils",
"LibGit2", "Libdl", "LinearAlgebra", "Markdown", "Mmap", "Pkg", "Printf", "REPL",
"Random", "SHA", "Serialization", "SharedArrays", "Sockets", "SparseArrays",
"Statistics", "Test", "UUIDs", "Unicode"]
git-tree-sha1 = "727e463cfebd0c7b999bbf3e9e7e16f254b94193"
uuid = "34da2185-b29b-5c13-b0c7-acf172513d20"
version = "3.34.0"

[[Crayons]]
git-tree-sha1 = "3f71217b538d7aaee0b69ab47d9b7724ca8afa0d"
uuid = "a8cc5b0e-0ffa-5ad4-8c14-923d3ee1735f"
version = "4.0.4"

[[DataAPI]]
git-tree-sha1 = "ee400abb2298bd13bfc3df1c412ed228061a2385"
uuid = "9a962f9c-6df0-11e9-0e5d-c546b8b5ee8a"
version = "1.7.0"

[[DataFrames]]
deps = ["Compat", "DataAPI", "Future", "InvertedIndices",
"IteratorInterfaceExtensions", "LinearAlgebra", "Markdown", "Missings", "PooledArrays",
"PrettyTables", "Printf", "REPL", "Reexport", "SortingAlgorithms", "Statistics",
"TableTraits", "Tables", "Unicode"]
git-tree-sha1 = "d785f42445b63fc86caa08bb9a9351008be9b765"
uuid = "a93c6f00-e57d-5684-b7b6-d8193f3e46c0"
version = "1.2.2"
```