Julia Tutorial for optimization and operations research

Part 01 - Brief introduction of Julia

Prof. Luiz-Rafael Santos - https://lrsantos11.github.io

UFSC, Brazil and MS&E/Stanford, US

How this tutorial will work

Julia is an *open-source* computational language that has proven to be a great asset in the field of scientific computing.

- Hands on using Julia through Pluto notebooks and/or VSCode) (possibly using terminal too)
- Some theory and a lot of practice (not a modelling or optimization course)
- Coding best practices (in Julia) will be encouraged
- **Focus:** Tools for learning and research with a focus on optimization as well as operations research
    - Problem modeling and solving (practical problems)
    - Development of optimization algorithms
    - Testing and comparison of methods

Presenting Julia

Some advantages for scientific computing and research

- Open-source code
- Dynamic language with support for iterativity (Jupyter and Pluto Notebooks, REPL)
- Julia uses *multiple dispatch* as a paradigm, making it easy to express functional programming patterns and object-oriented programming.
- **Fast** - developed for high performance
- Syntax similar to MATLAB/Octave and Python
- Easy importing of libraries and interface with C, Fortran, C++, Python, R, Java, among other languages
- Huge number of developers and packages for various fields of Scientific Computing
    - In addition to optimization (we will see this throughout the course), there are great packages for Differential Equations (DifferentialEquations.jl), Statistics and Data Science (DataFrames.jl and JuliaStat), Machine Learning (MLJ.jl), Images (JuliaImages), Parallel Computing (DistributedArrays.jl and GPUs), Economics (QuantEcon.jl), Bioinformatics (BioJulia), Dynamical Systems (JuliaDynamics), among other things;
- Vast number of documentation and tutorials available:
    - Start with the Julia Cheat Sheet and use Google!
    - Easy adaptation for MATLAB/Octave and Python users, very common programs in the implementation of optimization algorithms.

Advantages for Optimization and OR

Specific ecosystems for modelling

Those that I use or like the most, without any preference order:

- JuMP: algebraic modeling language for linear, quadratic, and nonlinear optimization (with or without constraints)
    - JuMP makes it easy to formulate and solve a range of problem classes, including linear programs, integer programs, conic programs, semidefinite programs, and constrained nonlinear programs.
    - You can use it to
        - route school buses,
        - schedule trains,
        - plan power grid expansion, or
        - optimize milk output.
- Convex.jl: algebraic modeling language for disciplined convex programming
- MathOptInterface: an abstraction layer for using optimization solvers. Many solvers are available in a very simple way using MOI:
    - HiGHS for linear, mixed integer and quadratic optimization
    - IPOpt for nonlinear optimization
    - *Comercial solvers*: Gurobi, KNitro, CPLEX, Xpress, Mosek

For continuous optimization

- JuliaSmoothOptimizers (JSO): collection of Julia packages for development, testing, and benchmarking of (nonlinear) optimization algorithms.
    - Modeling
        - NLPModels: API to represent optimization problems `min f(x) s.t. l <= c(x) <= u`
        - CUtEst.jl: interface to CUTEst, a repository of optimization problems for testing and comparing optimization algorithms.
        - BenchmarkProfiles: Julia package for generating the widely used Dolan-Moré-Wild performance profiles to compare optimization algorithms.
- ProximalOperators.jl: implements first-order primitives (in particular, prox operators) that facilitate convex optimization.
- Ease of generating graphics and visualizations using the Plots ecosystem.

How to use `Julia` ?

We can use `Julia` :

- in a notebook (like Pluto.jl or Jupyter)
- In IDE like: *terminal + editor* or *VSCode*

- In Pluto just add package in cell

```
using NameOfPackage
```

- In terminal, one have to install packages first

```
julia> ]
(@v1.8) pkg> activate .
(NameOfFolder) pkg> instantiate
(NameOfFolder) pkg> add Plots # if necessary, add other packages
```

A beatiful plot of the Lorenz system

In 1963, Lorenz and co-autors developed a simplified mathematical model for atmospheric convection. The model is a system of three ordinary differential equations now known as the Lorenz equations:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \sigma(y - x),$$
$$\frac{\mathrm{d}y}{\mathrm{d}t} = x(\rho - z) - y,$$
$$\frac{\mathrm{d}z}{\mathrm{d}t} = xy - \beta z.$$

The solution is a beautiful plot called Lorenz Attractor (or Lorenz Butterfly)

Plotting the Lorenz Attractor in `Julia`

```
Define a point in Lorenz atractor as a strucuture
```

Lorenz

```
Base.@kwdef mutable struct Lorenz
        dt::Float64 = 0.02 #step size
        σ::Float64 = 10 # Prandtl number
        ρ::Float64 = 28 # Rayleigh bymber  (you can choose other values like 13, 14, 15)
        β::Float64 = 8/3
        x::Float64 = 1 # Initial point
        y::Float64 = 1 # Initial point
        z::Float64 = 1 # Initial point
    end
```

Step of a discrete time ODE solver (Euler method)

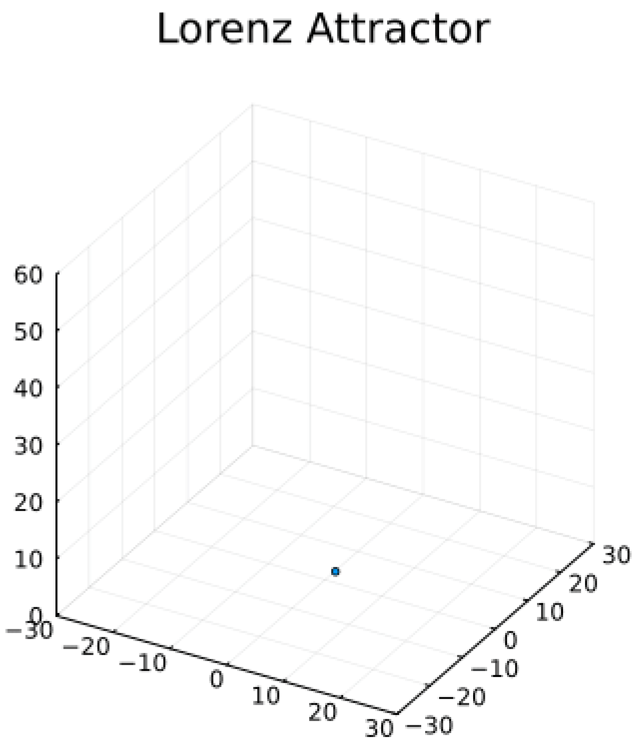
step! (generic function with 1 method)

```
function step!(l::Lorenz)
    dx = l.σ * (l.y - l.x);         l.x += l.dt * dx
    dy = l.x * (l.ρ - l.z) - l.y;   l.y += l.dt * dy
    dz = l.x * l.y - l.β * l.z;     l.z += l.dt * dz
end
```

Initialize Lorezn attractor

attractor = Lorenz(0.02, 10.0, 28.0, 2.6666666666666665, 1.0, 1.0, 1.0)

```
attractor = Lorenz()
```



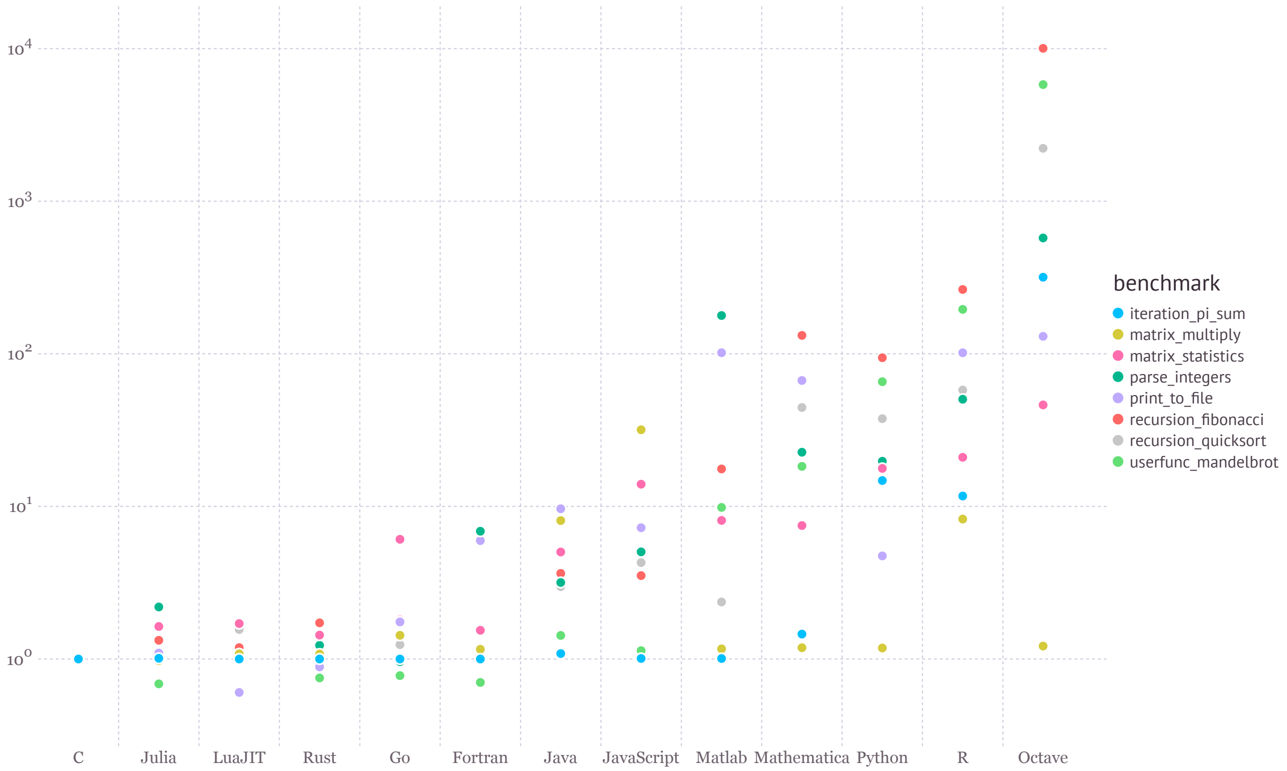

```julia
begin
	# Initialize the plot with the first point
	plt = plot3d(
		1,1,1,
		xlim = (-30, 30),
		ylim = (-30, 30),
		zlim = (0, 60),
		title = "Lorenz Attractor",
		marker = 2,
		leg = false,
	)


	# Building a gif adding points to the plot, saving at each 10 frames
	plt = @gif for i=1:1500
		step!(attractor)
		push!(plt, attractor.x, attractor.y, attractor.z)
	end every 10
end
```

```
Saved animation to /var/folders/_b/3kv60l2n3xz0kgr__6tfxd_00000gn/T/jl_UbMX2B8n
Mt.gif
```

How fast is Julia?

- According to the official site, very **FAST**



Simple example: accessing matrices compontes row-wise or column-wise

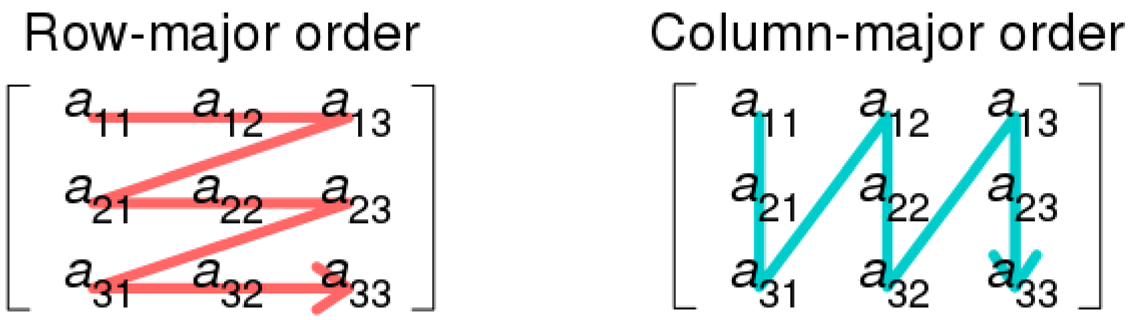
- Ordering types: [Row-major or Column-major order]()

Example

Matrix $A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$ can be possible stored in the computer memory in (possible) two-ways:

Address	Row-major	Column-major
1	a_{11}	a_{11}
2	a_{12}	a_{21}
3	a_{13}	a_{12}
4	a_{21}	a_{22}
5	a_{22}	a_{13}
6	a_{23}	a_{23}

For $A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$ we get



Programming language specific

- **Row-major**: `numpy (python)`, `C/C++/Objective-C`, `Pascal`, SAS.
- **Column-major**: `Fortran`, MATLAB, GNU Octave, `R`, `Julia`, e Scilab.

Let us run some code

- The CPU time of the algorithm that allocates value $10i + j$ for component a_{ij} of matrx $A \in \mathbb{R}^{m \times n}$ in:
  - `C`
  - Python, using `numpy`
  - `Julia`

Remark: In `Julia`, we use package [BenchmarkTools.jl]() to compute CPU time.

```
begin

    using LinearAlgebra
    using StatsBase
    using PlutoUI
    using PlutoReport
    using Plots
    using HypertextLiteral: @htl, @htl_str

    struct Foldable{C}
        title::String
        content::C
    end

    function Base.show(io, mime::MIME"text/html", fld::Foldable)
        write(io,"<details><summary>$(fld.title)</summary><p>")
        show(io, mime, fld.content)
        write(io,"</p></details>")
    end

    struct TwoColumn{L, R}
        left::L
        right::R
    end

    function Base.show(io, mime::MIME"text/html", tc::TwoColumn)
        write(io, """<div style="display: flex;"><div style="flex: 50%;">""")
        show(io, mime, tc.left)
        write(io, """</div><div style="flex: 50%;">""")
        show(io, mime, tc.right)
        write(io, """</div></div>""")
    end
    # apply_css_fixes()
    # @bind _pcon presentation_controls(aside=true)
end
```