Julia crash course

Mark Agerton

January 25, 2021

Julia

Why Julia? Multiple dispatch

1. *Multiple dispatch*: functions do different things depending on the *types* of variables
   - Explanations
     - https://www.juliaopt.org/meetings/santiago2019/slides/stefan_karpinski.pdf
     - https://arstechnica.com/science/2020/10/the-unreasonable-effectiveness-of-the-julia-programming-language/
   - Class-based OOP
     - All methods hard-coded in to class from outset
     - $\implies$ hard to extend packages
     - New function? $\implies$ rewrite data type
   - Functional language
     - Can add new functions... but new data types means rewriting existing functions
   - Multiple dispatch?
     - Old functions handle new ingredients
     - New functions handle old ingredients

Why Julia? (con'td)

2. Super fast without special tweaks (vs Python, R)
   - Don't have to vectorize for performance (looking at you, MATLAB!)
   - Don't have to use Numba/Cython whatever you do in Python
   - Thanks, multiple dispatch!
3. Programming syntax is close to the mathematics
4. Automatic differentiation
5. Parallelism is as straightforward as it can be
6. Open-source ecosystem
   - Free
   - Easy to contribute via Github
7. Codebase is in Julia

Why Julia (cont'd)

8. Can call R, Python, C, etc
9. Easy to do unit-testing, work with packages
10. Does functional programming (like Python or R's `purr`)
11. Programming language `JuMP` for LP/NLP/Integer Prog
    - Like GAMS/AAMPL
    - Many backend solvers (Knitro/Ipopt/Gurobi/etc)
    - Automatic differentiation

Why NOT Julia?

- Documentation is so-so
- Packages do get updated sometimes and break your code
- Slow to get to first plot b/c of compilation

When to use Julia?

- Julia was designed for *scientific computing* first and foremost
- Use it when you have to do some serious DIY computation
- If there is a canned stats routine in R/Python/MATLAB, you should prob use it instead

Things you need to know about Julia

- You can't delete variables from memory. Assign them a new value and let garbage collector take care of them
- Functions pass arguments by *reference*, not by *value*. (can't update scalars though)
- Naming convention: functions with names ending in `!` modify first argument(s)

```
x = [1.0, 2.0, 3.0,]
function f!(z::AbstractArray)
    z[1] = 0.0
end
f!(x)
x == [0.0, 2.0, 3.0]  # should be true
```

Things you need to know about Julia (cont'd)

- Every variable has a *type*. There are heirarchies of types (recall numbers taxonomy). Functions can do different things depending on the types you pass to them (*multiple dispatch*)
- Types have sub-types: `Vector{Int}` vs `Vector{Float64}`
- We can *broadcast* any function or operator element-by-element by prefixing with dots

```
A = [1.0  2.0; 3.0  4.0]
B = [5.0  6.0; 7.0  8.0]
all(A*B .== A .* B)   # broadcast vs matrix multiplication
```

- We load in libraries of code (*packages* or *modules*) by writing the following (similar to `library(dplyr)` in R or `import numpy` in Python)

```
using Plots
```

Julia references

- Julia documentation: Julia vs MATLAB, R Syntax
- QuantEcon Julia Lectures. Skim through lectures 1 & 2 from the QuantEcon Julia lectures to see how to install Julia and get started. Nice macro/DP lectures.
- Julia Cheat Sheet
  - See also MATLAB/Python/Julia translation: https://cheatsheets.quantecon.org/
- *Think Julia* book (also available via UCD Library after authentication) is good for getting started.
- *Hands-On Design Patterns and Best Practices with Julia* book has more advanced programming concepts
- Jesús Fernández-Villaverde's Chapter on Julia and list of Julia commands

Jupyter notebooks/Pluto

- Can use Jupyter notebooks like Python does
  - similar to browser-based Rmarkdown, but less klutzy with the cells
  - Handles `JuMP`
- I MUCH prefer Julia's `Pluto.jl`
  - Trackable in Git
  - No hidden state!
    - Create `x=1`
    - Run some code that knows about `z = f(x)`
    - And then rebind `x` to `x=2`
    - `z` gets update
    - Results NEVER depend on the order you ran each bit of code (as it does interactively)

Logit example

Constrained optimization with `JuMP`

Julia for package development

- Idea
  - put functions inside a *module* (package)
  - build up functions based on tests I work through
  - Have Julia watch for changes in module using `Revise.jl`
- Generate a new package
  - In REPL, change to `dev` (development) directory:

    ```
    ;cd C:\Users\<myusername>\.julia\dev
    ;cd /Users/<myusername>/.julia/dev
    ```

  - Generate a package

    ```
    ]generate Homework06
    ```

  - Make sure to `]dev` or `]add Homework06`
  - cd into `Homework06` and `]activate .`
  - add packages to update `Project.toml`

Then, in ~~Atom~~ VS Code

- Add `rcmnl` dir to project folder
- Develop code in `src/rcmnl.jl`
- Run tests in `test/runtests.jl`
- Can add `using Revise` before loading `rcmnl` to have Julia pick up updates
- Also, do automatic testing `]test rcmnl`
- See Revise.jl based workflows

Random coef multinomial logit

Have panel with individual i, time t, choice $k = 0, \ldots, 2$.

Utilities of choices are

$$u_{itk} = \begin{cases} & \text{if } k = 0 \\ x_{it}^\top \beta_k + e_{ik} & \text{if } k > 0 \end{cases}$$

where

$$e_i \sim N(0, \Sigma) \qquad\qquad \epsilon_{itk} \sim iid \text{ T1EV}$$

Individual solves $V_{it} = \max_k \{u_{itk} + \epsilon_{itk}\}$

Likelihood conditional on random effect

Likelihood conditional on X_i, e_i

$$\log L_i(y_i|X_i, v_i) = \sum_t \left(u_{itk} - \log \sum_k \exp\{u_{itk}\} \right)$$

Integrating out e_i

Integrate out e_i using Gauss-Hermite Quadrature

G-H designed for $\int_{-\infty}^{+\infty} e^{-x^2} G(x) dx$

Need to use Cholesky decomposition of $\Sigma = LL^\top$

If we use J integration points for $v_{ij} \in \mathbb{R}^k$ where $k = 2$, need to compute

$$\int Lik(e) dF(e; \Sigma) \approx \pi^{-k/2} \sum_{i=1}^{J} \sum_{j=1}^{J} \omega_i \omega_j Lik(\sqrt{2} L v_{ij} + \mu)$$

where $v_{ij} = [node_i, node_j]^\top$

Thus, simulated likelihood is

Compute simulated log likelihood as

$$\frac{1}{\pi}\log\left(\sum_j \omega_j Lik(y_i|X_i, v_j)\right) = \frac{1}{\pi}\log\sum_j \exp\left\{\log(\omega_j) + \log L(y_i|X_i, v_j)\right\}$$

All together

$$u_{itk} = \begin{cases} 0 & \text{if } k = 0 \\ x_{it}^\top \beta_k + e_{ik} & \text{if } k > 0 \end{cases}$$

Likelihood conditional on shock e_i

$$\log L_i(y_i | X_i, e_i) = \sum_t \left(u_{itk} - \log \sum_k \exp\{u_{itk}\} \right)$$

SLL is

$$\log L(data) = \sum_i \frac{1}{\pi} \log \sum_j \exp\left\{ \log \omega_j + \log L(y_i | X_i, \sqrt{2} L v_j) \right\}$$

where ω is quadrature weight and $v_j \in \mathbb{R}^2$ is the node