JULIAN ECONOMICS

Spencer Lyon

April 23, 2015

OUTLINE

Intro

Quant-Econ

Examples

Pitfalls

Final thoughts

INTRO

ABOUT ME

- Personal
  - Economics PhD student at NYU.
  - Physics and econ undergrad
  - I have a wife and two kids
- Programming
  - Started on Mathematica
  - First love was Python
  - Dabbled with C, C++, R, Scala, Haskell, MATLAB
  - New favorite for many tasks is Julia

WHY I LIKE JULIA

- Fast
- Functional
- Flexible
- Clean
- Open source

JULIA IS ... FAST

- We've seen the benchmarks
- This matters for economics because problems often have:
  - Many states
  - Solving functional equations on state space
  - Many algorithms require explicit looping over matrices that represent these functions
  - Typical model is stochastic ⇒ requires approximation of expectation ⇒ hard (impossible?) to parallelize *across* iterations
- So, fast iterations are crucial

JULIA IS ... FUNCTIONAL

- Proper support of basic functional programming makes code readable and concise:
  - `do` notation
  - `map`, `fold(l|r)`, `reduce`, `pmap`, comprehensions, ...
- "Litetweight" types make it natural to have very small types (can be treated like a Dict in python or a list in R, with the additional ability to specify how functions operate on it, even relative to types of neighboring arguments)
- Multiple dispatch lets you combine two previous points in unique and powerful ways (e.g. type-based API – not kwarg. Example to come)

JULIA IS ... FLEXIBLE

- Most of the standard library written in Julia itself – and it is fast

JULIA IS ... FLEXIBLE

- Most of the standard library written in Julia itself – and it is fast
    - Means other code written in Julia has *potential* to perform at the same level as the standard library (if written well)

JULIA IS ... FLEXIBLE

- Most of the standard library written in Julia itself – and it is fast
  - Means other code written in Julia has *potential* to perform at the same level as the standard library (if written well)
  - Not true of most high level languages (e.g. Python, R, Matlab) where standard library or core numeric tools written in lower level language

JULIA IS ... FLEXIBLE

- Most of the standard library written in Julia itself – and it is fast
  - Means other code written in Julia has *potential* to perform at the same level as the standard library (if written well)
  - Not true of most high level languages (e.g. Python, R, Matlab) where standard library or core numeric tools written in lower level language
  - To achieve standard library performance in those languages you usually need to write routines in C/C++/Fortran and wrap it for use in high-level language

JULIA IS ... FLEXIBLE

- Most of the standard library written in Julia itself – and it is fast
  - Means other code written in Julia has *potential* to perform at the same level as the standard library (if written well)
  - Not true of most high level languages (e.g. Python, R, Matlab) where standard library or core numeric tools written in lower level language
  - To achieve standard library performance in those languages you usually need to write routines in C/C++/Fortran and wrap it for use in high-level language
- Can call C, natively with no overhead and no wrappers.

JULIA IS ... FLEXIBLE

- Most of the standard library written in Julia itself – and it is fast
  - Means other code written in Julia has *potential* to perform at the same level as the standard library (if written well)
  - Not true of most high level languages (e.g. Python, R, Matlab) where standard library or core numeric tools written in lower level language
  - To achieve standard library performance in those languages you usually need to write routines in C/C++/Fortran and wrap it for use in high-level language
- Can call C, natively with no overhead and no wrappers.
  - Opens door to call Python (PyCall.jl), R (RCall.jl), MATLAB (MATLAB.jl), Java (JavaCall.jl), ect.

JULIA IS ... FLEXIBLE

- Most of the standard library written in Julia itself – and it is fast
    - Means other code written in Julia has *potential* to perform at the same level as the standard library (if written well)
    - Not true of most high level languages (e.g. Python, R, Matlab) where standard library or core numeric tools written in lower level language
    - To achieve standard library performance in those languages you usually need to write routines in C/C++/Fortran and wrap it for use in high-level language
- Can call C, natively with no overhead and no wrappers.
    - Opens door to call Python (PyCall.jl), R (RCall.jl), MATLAB (MATLAB.jl), Java (JavaCall.jl), ect.
    - Easily write Julia interface into mature C libraries (LAPACK and BLAS in standard library, NLopt, Sundials, many more...)

JULIA IS ... CLEAN AND OPEN SOURCE

- Syntax is powerful and concise
  - Convenient linear algebra syntax (`A * B` instead of `A %*% B` or `np.dot(A, B)`)
  - Matlab-esque matrix construction
  - Minor points, but make the experience better
- Open source
  - Learn how (and sometimes why) standard library functions are implemented
  - Gihub issue list or the google group great ways to watch progress

QUANT-ECON

WEBSITE

QuantEcon is an organization run by economists for economists with the aim of coordinating distributed development of high quality open source code for all forms of quantitative economic modeling.

- Two fold:
  1. Website with over almost 40 teaching modules (textbook chapters) that teach programming and economics
  2. Code libraries in Python and Julia

LIBRARIES

- Started as teaching tools – implementations of routines in chapters
- Transitioning into performance-oriented set of tools
- Julia and Python versions, both first class members
- Open source, community developed, on github

EXAMPLES

PLAYING NICELY

- Many potential users may be worried about “abandoning” code they have written or rely on from other languages
- Julia’s ability to naturally call R and Python might alleviate these concerns
- NOTE: show RCall and PyCall examples

FUNCTIONAL STYLE

- As mentioned before; Julia’s functional style, lightweight types, and multiple dispatch open the door for unique API design opportunities
- NOTE: show IterationManagers and CompEcon examples

PITFALLS

SPAGHETTI

- Lightweight types and multiple dispatch offer power
- …sometimes too much power
- It is tempting to make a type or new function for every operation "just in case" you will dispatch on it in the future
- You might end up with complicated spaghetti code composed of many `type`s of noodles
- Personal rule(s) of thumb
    - **Don't** break out parts of a function if I will only ever call them from one place
    - **Do** break out parts of a function if I can dispatch on them

YOUNG LANGUAGE

- Community is relatively small (compared to Python, R, MATLAB)
  - Cons
    - Not many mature learning materials
    - Less collective man power writing packages, tutorials, ect.
  - Pros
    - Interact with “big” names (“Hi Stefan!” :))
    - Opportunity for users to help form community, culture, even the language itself
- Not to version 1.0 yet
  - (quickly) Moving target to develop against
  - Code that ran a few months ago might not run today
  - Usually very easy to fix these issues
- Conventions still in flux (docs, testing, style-guide not quite PEP8)
- Package ecosystem not as rich as Python or R (or MATLABs toolboxes for specific functionality)
  - But it is growing… fast

CONVEX LEARNING CURVE

- Easy to learn, hard to master
- MATLAB or NumPy users immediately comfortable writing functions and using Arrays
- Unlocking full Julia potential requires
    - Learning to think functionally (not traditional OOP, or even procedural)
    - Understanding type system (abstract, composite, parametric...) can be intimidating
    - Advanced features like meta-programming are powerful and seductive, but often improperly used

FINAL THOUGHTS

TIPS

- `@less`, `@which`, `JULIA_EDITOR` + `@edit`
- Get involved - follow mailing list or issue list on github