About these Lectures

Jesse Perla, Thomas J. Sargent and John Stachurski

December 4, 2020

1 Overview

Programming, mathematics and statistics are powerful tools for analyzing the functioning of economies.

This lecture series provides a hands-on instruction manual.

Topics include

related mathematical and statistical concepts, and

basics of coding skills and software engineering.

The intended audience is undergraduate students, graduate students and researchers in economics, finance and related fields.

2 Julia

The coding language for this lecture series is Julia.

Note that there's also a related set of Python lectures.

In terms of the differences,

- Python is a general purpose language featuring a huge user community in the sciences and an outstanding scientific and general ecosystem.
- Julia is a more focused language primarily used in technical and scientific computing, with an outstanding ecosystem for cutting-edge methods and algorithms.

Both are modern, open source, high productivity languages with all the key features needed for high performance computing.

While Julia has many features of a general purpose language, its specialization makes it much closer to using Matlab or Fortran than using a general purpose language - giving it an advantage in being closer to both mathematical notation and direct implementation of mathematical abstractions.

2.1 A Word of Caution

The disadvantage of specialization is that Julia tends to be used by domain experts, and consequently the ecosystem and language for non-mathematical/non-scientfic computing tasks is inferior to Python.

Another disadvantage is that, since it tends to be used by experts and is on the cutting edge, the tooling is much more fragile and rudimentary than Matlab.

Thankfully, with the v1.x release of Julia the fragility no longer applies to the language itself - now stable and carefully managed for version compatibility. However, casual users should not expect the development tools to quite as stable, or to be comparable to Matlab.

Nevertheless, the end-result will always be elegant and grounded in mathematical notation and abstractions.

For these reasons, Julia is most appropriate at this time for researchers who want to:

1. invest in a language likely to mature in the 3-5 year timeline
2. use one of the many amazing packages that Julia makes possible (and are frequently impossible in other languages)
3. write sufficiently specialized algorithms that the quirks of the environment are much less important than the end-result

3 Advantages of Julia

Despite the short-term cautions, Julia has both immediate and long-run advantages.

The advantages of the language itself show clearly in the high quality packages, such as

- Differential Equations: DifferentialEquations.jl
- Function approximation and manipulation: ApproxFun.jl
- Interval Constraint Programming and rigorous root finding: IntervalRootFinding.jl
- GPUs: CuArrays.jl
- Linear algebra for large-systems (e.g. structured matrices, matrix-free methods, etc.): IterativeSolvers.jl, BlockBandedMatrices.jl, InfiniteLinearAlgebra.jl, and many others
- Automatic differentiation: Zygote.jl and ForwardDiff.jl

These are in addition to the many mundane but essential packages available. While there are examples of these packages in other languages, no other language can achieve the combination of performance, mathematical notation, and composition that Julia provides.

The composition of packages is especially important, and is made possible through Julia's use of something called multiple-dispatch.

The promise of Julia is that you write clean mathematical code, and have the same code automatically work with automatic-differentiation, interval arithmetic, and GPU arrays–all of which may be used in cutting edge algorithms in packages and combined seamlessly.

4 Open Source

All the computing environments we work with are free and open source.

This means that you, your coauthors and your students can install them and their libraries on all of your computers without cost or concern about licenses.

Another advantage of open source libraries is that you can read them and learn how they work.

For example, let's say you want to know exactly how Distributions.jl implements mean of the exponential function

No problem: You can go ahead and read the code.

This goes even further since most of Julia is written in Julia. For example, you could see the code for how the standard library calculates the eigenvalues of a triangular matrix or calculates the mean of a range

Those two examples also provide examples where the "multiple dispatch" allows exploitation of the structure of a problem, leading to specialized algorithms (e.g. if the user calls `eigvals` on a matrix that happens to be triangular, it just needs to return the diagonal).

While dipping into external code libraries takes a bit of coding maturity, it's very useful for

1. helping you understand the details of a particular implementation, and
2. building your programming skills by showing you code written by first rate programmers.

Also, you can modify the library to suit your needs: if the functionality provided is not exactly what you want, you are free to change it.

Another, more philosophical advantage of open source software is that it conforms to the scientific ideal of reproducibility.

5 How about Other Languages?

But why don't you use language XYZ?

5.1 MATLAB

While MATLAB has many nice features, it's starting to show its age.

It can no longer match Python or Julia in terms of performance and design.

MATLAB is also proprietary, which comes with its own set of disadvantages.

In particular, the Achilles Heel of Matlab is its lack of a package management system, which means that either you need to (1) rely on Matlab's own packages, which are mostly written for engineers and not economists; (2) write the code yourself; or (3) use unreliable and manual ways to share code (e.g. email or downloading a zip).

If you are a structural engineer or designing a microcontroller, then Matlab provides a coherent set of packages that takes care of all of your needs.

For economists, on the other hand, the expansion in complexity of numerical methods, the need for researchers to collaborate on code, fix bugs, deploy improvements, and have dependencies (i.e. packages relying on other packges) has increased past what Matlab can handle.

Given what's available now, it's hard to find any good reasons to invest in MATLAB.

Incidentally, if you decide to jump from MATLAB to Julia, this cheat-sheet will be useful.

5.2 R

R is a very useful open source statistical environment and programming language

Its primary strength is its vast collection of extension packages

Julia is more general purpose than R and hence a better fit for this course

5.3 C / C++ / Fortran?

Isn't Fortran / C / C++ faster than Julia? In which case it must be better, right?

For the same algorithms, as a compiled language Julia can often achieve a similar level of performance to those languages.

But even when it doesn't, keep in mind that the correct objective function to minimize: total time = development time + execution time

In assessing this trade off, it's necessary to bear in mind that

- Your time is a far more valuable resource than the computer's time.
- Languages like Julia are much faster to write and debug in.
- In any one program, the vast majority of CPU time will be spent iterating over just a few lines of your code.

The other issue with all three languages, as with Matlab, is the lack of a package management system. Collaborating on C++ or Fortran packages and distributing code between researchers is difficult, and many of the criticisms of matlab equally apply.

Finally, the first-order question of performance is which algorithm you are using, and whether it exploits the structure of the problem. The right algorithm in Matlab or Python is typically faster than the wrong algorithm in Fortran - and the right algorithm in Fortran and Julia can be made roughly comparable.

When considering the performance advantages, remember that the design and package system of Julia make it easy to try out algorithms that exploit structure of the problem. We will investigate this more in Introductory Examples

5.3.1 Last Word

Writing your entire program in Fortran / C / C++ is best thought of as "premature optimization"

On this topic we quote the godfather:

> We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. – Donald Knuth

But, to put the final part of the quote

> … Yet we should not pass up our opportunities in that critical 3%. – Donald Knuth

Julia is an excellent language to attain those last few percent, without having to resort to C or Fortran code and mix languages in your project

6 Credits

These lectures have benefited greatly from comments and suggestions from our colleagues, students and friends. Special thanks are due to our sponsoring organization the Alfred P. Sloan Foundation and our research assistants Chase Coleman, Spencer Lyon and Matthew McKay for innumerable contributions to the code library and functioning of the website.

We also thank Andrij Stachurski for his great web skills, and the many others who have contributed suggestions, bug fixes or improvements. They include but are not limited to Anmol Bhandari, Long Bui, Jeong-Hun Choi, David Evans, Xiaojun Guan, Shunsuke Hori, Chenghan Hou, Doc-Jin Jang, Adam Jozefiak, Qingyin Ma, Akira Matsushita, Tomohito Okabe, Daisuke Oyama, David Pugh, Alex Olssen, Nathan Palmer, Pooya Rashidi Ravari, Arnav Sood, Bill Tubbs, Natasha Watkins, Pablo Winant, Kaan Yolsever and Yixiao Zhou.