Analyzing Large Datasets with the Julia Language

Matt Bhagat-Conway

Odum Institute for Research in Social Science

THE UNIVERSITY
of NORTH CAROLINA
at CHAPEL HILL

About me

- Assistant professor in City and Regional Planning and Odum Institute
- PhD and MA in Geography from Arizona State, BA from UC Santa Barbara
- Three years as professional software developer before graduate school

THE UNIVERSITY
of NORTH CAROLINA
at CHAPEL HILL

Outline

- Intro and conceptual discussion
- Hands on demos of
  - Reading and manipulating tabular data
  - Plotting
  - Performance improvements
  - Statistical analysis
- Brief discussion of more advanced topics

What is Julia?

- Relatively young programming language (just turned 10)
- Designed for speed and ease of use
- Geared for science and math
- Large community writing packages and providing support

Solving the two-language problem

- Scientists working with data have long needed two types of language
- Easy to use, simple, high-level languages
  - Prototyping and development very efficient
  - Poor performance at scale
  - e.g. Python, R
- More difficult, low-level languages
  - Can be very fast
  - Much higher learning curve
  - e.g. C, C++, Fortran, Java

What makes Julia fast?

- Just in time compilation
- Code optimization

Concepts: compile time and run time

- Compile time: when the Julia compiler examines your code and generated optimized low-level code for your processor to run
- Run time: when your code is actually executed by the computer
- Compile time happens once, run time as many times as you do something
- Most high-level languages do a lot at run time, Julia is fast because it moves more things to compile time

What makes Julia fast?

- The *type system*
  - Julia can usually *infer* the types of values used in Julia programs (e.g. integer, real, string, etc.)
  - When types can be inferred, specialized algorithms can be used for those types
  - Julia relies heavily on *multiple dispatch*, where different versions of a function can be selected based on the types of its arguments
  - Numeric types are fixed-width (same number of bits), so can be *stack-allocated*

Using Julia for social science

- Julia is a general-purpose language
- Data analysis functionality is provided by packages
  - DataFrames.jl for tabular data manipulation
  - CSV.jl to read CSV files
  - StatsBase.jl for descriptive stats
  - Plots.jl, Gadfly.jl, Makie.jl for plotting
  - GLM.jl for regression
  - MLJ.jl for machine learning
  - Graphs.jl for network analysis
  - etc...

Using Julia for social science

- Julia packages are usually written in Julia, so you can make your own or contribute to existing ones

THE UNIVERSITY
of NORTH CAROLINA
at CHAPEL HILL

Reasons not to use Julia

- Package ecosystem for social sciece is much more developed in R, Python, Stata, etc.
- More educational and troubleshooting resources available for other tools
  - But, the Julia community is very responsive (discourse.julialang.org and Julia Slack)
- More likely that collaborators are familiar with other tools

Reasons to use Julia

- For large datasets
  - Start considering at around 500mb
- For computationally intensive algorithms that aren't available in packages
- For distributed computing

The Julia community

- The Julia community is very friendly to newcomers
- discourse.julialang.org has many questions already answered, and community members are quite responsive if you post a new question
- Most Julia core developers are in the Julia Slack or Zulip channels, and will respond to questions there as well
- Julia and package documentation tends to be extensive as well

THE UNIVERSITY
of NORTH CAROLINA
at CHAPEL HILL

Recap: loading and cleaning data

(after `1 Reading Data into Julia.ipynb`)

- Julia's DataFrames package has tools to work with tabular data
- You need a separate library to read data (CSV, most often)
- Any Julia function can be applied to a vector by adding a `.`
- Missing data is common and represented by a special value `missing`
- `ismissing` will identify missing data, `skipmissing` skips them for a particular operation, and `coalesce` will return first non-missing argument

Recap: plotting

(after `2 Plotting.ipynb`)

- Plots.jl is a mature and high-performance plotting library
- The basic format of a plot is `plot(x, y)` or `scatter(x, y)`
- Appending `!` will add to existing plot
- `xlabel!` and `ylabel!` label axes
- `fmt=:png` speeds up very large plots
- `histogram(data)` will make a histogram

Julia performance

- It is easy to write high-performance code in Julia
- But it is *also* easy to write slow code in Julia
- Differences between fast and slow code can be very subtle

Julia performance

- The usual culprit in performance problems is *type instability*
  - ...when Julia can't figure out the type of a variable at compile-time, or the type of the variable changes
- When Julia doesn't know the type of the variable, it can't use efficient algorithms for that particular variable type
- Julia also cannot know the size of a variable of unknown type, so it must store it on the *heap* rather than the *stack*
- Heap memory must be garbage-collected, adding overhead, and is not in the same part of memory as other stack-allocated functions

Julia performance

- Put performance-critical code in functions. Julia can't specialize top-level code for specific types.
- Don't refer to global variables in functions; pass variables as arguments

Julia performance: the time-to-first-plot problem (TTFP)

- Unlike other compiled languages, Julia doesn't have a separate compilation phase
- The first time you use a function with particular types, it is compiled
- This can be quite slow, especially when compiling large libraries (e.g. Plots)
- When benchmarking, always look at the performance of the *second* function call

Julia performance tools

- Fortunately, Julia has a lot of tools to help with finding/fixing performance problems
- We'll cover two
- `@time` will time a function call and report heap memory allocations
- `@code_warntype` will highlight areas where Julia couldn't figure out the type of variables

Package management in Julia

(after `3 Performance.ipynb`)

- Research projects in Julia often rely on dozens or hundreds of packages, directly or indirectly
- By default, packages are installed into a "global environment" - meaning all your Julia projects share the same set of packages
- Best practice is to use per-project environments - so each project has its own set of packages
- This makes it easy to share code with others, or come back to it in a few years, without version compatibility issues

Package management in Julia

- Julia has a built-in package manager accessible from REPL
  - REPL: read-evaluate-print loop, known as command prompt or console in other languages
- Any folder can be a Julia environment
- To activate a Julia environment, at Julia prompt type `]activate <env>`
  - To activate the current directory, `]activate .`
- An environment has files `Project.toml` and `Manifest.toml` which track installed packages
- If a directory isn't already an environment, activating it will create the environment

Before proceeding

- Add the `GLM`, `MLJ`, `StatsModels`, `DecisionTree`, and `MLJDecisionTreeInterface` packages for our statistics demo
  - Open julia, activate current directory, and press `]` to open package manager
  - type `add GLM MLJ StatsModels DecisionTree MLJDecisionTreeInterface`

Statistics in Julia

- There are lots of packages for statistics in Julia
- We'll be using four
  - `StatsBase` for descriptive statistics
  - `GLM` for linear regression and GLMs
  - `MLJ` and `DecisionTree` for random forests

THE UNIVERSITY
of NORTH CAROLINA
at CHAPEL HILL

Statistics in Julia: recap

(after `4 Statistics.ipynb`)

- The `GLM` library provides (generalized) linear regression functions
- `MLJ` is an interface to a lot of different machine learning packages

Project organization in Julia

- Notebooks are convenient, but can quickly get out of hand
- Most large projects will have many notebooks with some common functionality
- Having code duplicated between notebooks leads to errors and maintenance hassles
- Any common code should go in a `.jl` file which is referenced in the notebook interface

Project organization: recap

(after `5 Moving Functions to Files.ipynb`)

- It's a good idea to put common functionality in a separate file (or files)
- `Revise` will let you edit these files while your notebooks are running without restarting Julia

Using Julia without notebooks

- At some point you probably will want to run Julia in a non-interactive environment
  - *e.g.* a compute cluster
- Instead of using a notebook, you can just put your Julia code into a file
- Everything should be in a function for best performance, except for one line that calls the main function
- You will have to explicitly use `println` or `@info` (with the `Logging` package) to show results

Advanced topics: linear algebra

- Julia has built-in support for *n*-dimensional matrices
- Built-in library `LinearAlgebra` provides many matrix operations
- Many space efficient specialized matrix types such as symmetrical, tridiagonal, etc.

Advanced topics: multithreading and multiprocessing

Premature optimization is the root of all evil

—Sir Tony Hoare/Donald Knuth

Danger, Will Robinson!

—Robot

Advanced topics: multithreading vs multiprocessing

- Multithreading uses multiple *threads* within a single *process* on a single computer
  - Threads share memory/variables
  - Saves memory, but prone to *race conditions*
  - Low overhead
- Multiprocessing uses multiple processes on one or more computers
  - Process don't share memory but communicate by passing information between them
  - Uses more memory, but allows dividing of processing across machines
  - High overhead

Advanced topics: multithreading

- Julia has built-in support for multithreading
- Start Julia with additional threads with `-t <num_threads>`
- Make for loops parallel with `Threads.@threads for`
- Avoid data races!
  - Common pattern is to have an array of results, one per thread, indexed by `Threads.threadid()`, and combine at end

Advanced topics: multiprocessing

- Julia provides `Distributed` module to support multiprocessing
- One main process sends commands and data to be executed to other processes
- You can start Julia with multiple processes on one machine using `-p <num_processes>`
- If you want to use a cluster, you will need a *cluster manager*, of which there are many
    - UNC Dogwood cluster uses Slurm, which you can use from Julia with `SlurmClusterManager`

Advanced topics: GPU support

- Julia supports NVIDIA, AMD, and Intel GPUs through the `CUDA`, `AMDGPU`, and `oneAPI` packages
- Good for large array operations that can be run in parallel (thousands of simultaneous processes working on different parts of the array)
- Tutorial: https://www.youtube.com/watch?v=Hz9IMJuW5hU

What people are using Julia for

- Astrophysics (Celeste.jl)
- Weather and climate modeling (CliMA project)
- Financial analysis (many private organizations)
- Economic forecasting (Federal Reserve Bank of NY)
- Electricity demand forecasting (Électricité de France)
- Migration modeling (Jakub Bijak)

Recap and additional resources

- Julia provides a high-performance environment for data analysis
- Most functionality in Julia is provided by packages
- Julia is fast because it can infer the types of data and variables
- If you write your code such that it *can't* infer types, it will be slow
  - Most commonly due to iterating over a DataFrame, or not writing code in a function
  - Check with `@time` and `@code_warntype`

Recap and additional resources

- Environments or projects are a way to keep a consistent set of packages and versions across team members
- Many stastical functions are available, but most are in packages (even `mean`!)
- Larger projects with multiple notebooks should put common code into files
- Julia scripts can contain the same code as notebooks, but should have all code in functions

Recap and additional resources

- Julia documentation: docs.julialang.org
- Julia Discourse: discourse.julialang.org
- JuliaCon: juliacon.org
- Julia Slack
- Package documentation
- Demos from this course: github.com/mattwigway/odum-julia

Matt Bhagat-Conway
mwbc@unc.edu

THE UNIVERSITY
of NORTH CAROLINA
at CHAPEL HILL