An Introduction to julia

Madeleine Udell, ORIE
Scientific Software Club
October 24 2016

Why I like Julia

- It's like MATLAB in that
  - Basic syntax is similar
  - Great for linear algebra: BLAS libraries in base namespace
- It's like Python in that
  - It's interoperable powerful glue
  - Good interaction with Jupyter
  - No need for explicit memory management
  - It's open source
- It's like C in that
  - It's pretty fast
  - (Julia is JIT-compiled)
- Very strong user community
  - Interaction takes place on Github

What’s not to like?

- Debugging is wanting
- Error messages can be hard to interpret
- Parallelism has yet to fulfill its promise
- Not yet 1.0
    - I’ve been using Julia since 2012
    - For me, “breaking changes” have not been very breaking

Features

- Multiple dispatch: providing ability to define function behavior across many combinations of argument types
- Dynamic type system: types for documentation, optimization, and dispatch
- Good performance, approaching that of statically-compiled languages like C
- Built-in package manager
- Lisp-like macros and other metaprogramming facilities
- Call Python functions: use the PyCall package
- Call C functions directly: no wrappers or special APIs
- Powerful shell-like capabilities for managing other processes
- Designed for parallelism and distributed computation
- Coroutines: lightweight “green” threading
- User-defined types are as fast and compact as built-ins
- Automatic generation of efficient, specialized code for different argument types
- Elegant and extensible conversions and promotions for numeric and other types
- Efficient support for Unicode, including but not limited to UTF-8
- MIT licensed: free and open source

Installing and using Julia

- Install current version from julialang.org
- Many ways to invoke:
  - command line interface
  - Juno IDE
  - Jupyter notebooks
  - Atom text editor can evaluate chunks
  - ...
- Try it out on JuliaBox.com
  - Sync with https://github.com/madeleineudell/intro-to-julia

On the command line interface: try typing “;” or “?”

Basic syntax

- Demo

The Two Language Problem

1) Begin project writing simple, easy code in a prototyping language
    - eg, Python, MATLAB, R
2) Realize you need the code to be fast
3) Rewrite code in a fast language
    - C, Scala, Fortran
4) Wrap the code
    - Cython, Mex, ...

Julia’s solution to the two-language problem

- Start with basic MATLAB-like syntax
- Speed it up: take advantage of JIT compilation
  - Add type annotations
  - Devectorize (!)

The Many-Language Problem

Problem: want to use code written in a bunch of different languages

Julia is a good **glue language**:

- Native support for Fortran and C
- Fast in-memory interfaces for Python, R, MATLAB, C++, Java, …
- Can also call Julia from C!

Advanced feature demo

- Demo:
  - JIT
  - Introspection
  - Packages
  - Calling out

Parallel Computing

- Embarrassingly parallel operations are embarrassingly easy

- Fine-grained control over data movement and interprocess communication
  - RemoteRefs, Channels, ...
- What about multithreading?
  - There’s no GIL, so it’s possible
  - In the bad old days: I’ve done it via SharedArrays
  - Now: experimental multithreading implementation…!

Optimization in Julia

- Julia is currently the best programming language for mathematical optimization
- Why?
  - Convex.jl
  - JuMP
  - MathProgBase
  - Solver interfaces

Optimization: from math to solution

1) Write down a mathematical description
2) Identify structure in objective/constraints
3) Perform transformations to change structure
4) Identify a solution method (algorithm, solver)
5) Massage transformed problem into something the algorithm can work with
6) Code it up and pass it along

Optimization schematic in Julia

```julia
type OptProb{T}; blob; end

solve(x::OptProb{:Linear}) = simplex_method(x.blob)

solve(x::OptProb{:Semidefinite}) = interior_point(x.blob)

solve(x::OptProb{:Integer}) = branch_and_bound(x.blob)

solve(x::OptProb{:Lasso}) = glmnet(x.blob)
```

The details

A couple questions:

1. How do we recognize the problem class?
2. How do we translate `blob` into something useful?

The details

A couple questions:

How do you create `blob` in the first place?

1. How do we recognize the problem class?
2. How do we translate `blob` into something useful?

The details

A couple questions:

How do you create `blob` in the first place?

1. How do we recognize the problem class?
2. How do we translate `blob` into something useful?

JuMP and Convex are tools to help you build `blob` (and then do 1 and 2

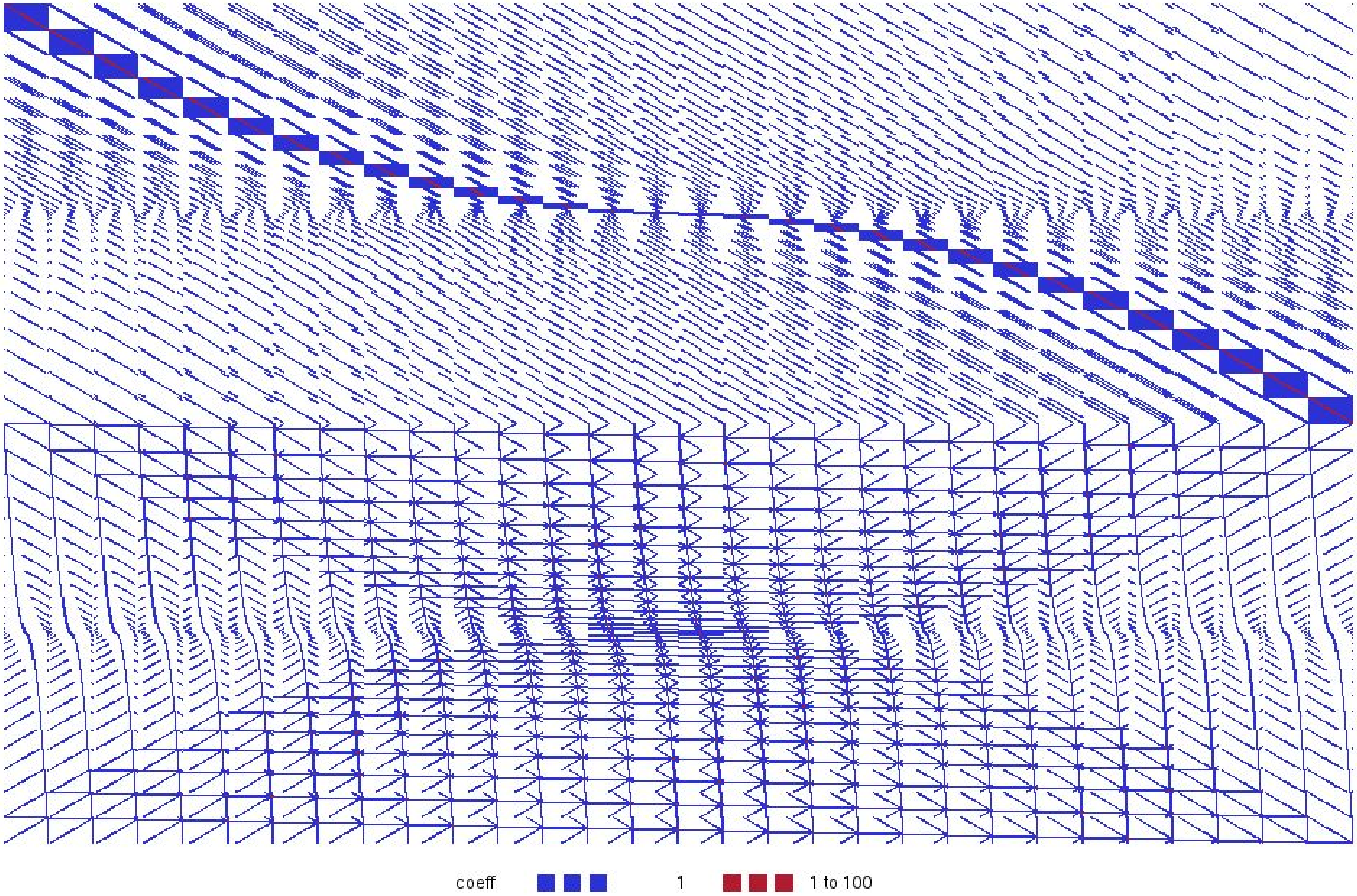
Algebraic modeling languages

- JuMP and Convex are *algebraic modeling languages* (AMLs)
- AM

$$
\begin{aligned}
\min_{\mathbf{u},\mathbf{y}} \quad & \frac{1}{4}\Delta_x \left(\left(y_{m,0} - y_0^t\right)^2 + 2\sum_{j=1}^{n-1} \left(y_{m,j} - y_j^t\right)^2 + \left(y_{m,n} - y_n^t\right)^2 \right) + \\
& \qquad \frac{1}{4} a \Delta_t \left(2 \sum_{i=1}^{m-1} u_i^2 + u_m^2 \right) \\
\text{s.t.} \quad & {}^1\!/_{\Delta_t} \left(y_{i+1,j} - y_{i,j} \right) = \\
& \qquad \frac{1}{2h_2} \left(y_{i,j-1} - 2y_{i,j} + y_{i,j+1} + y_{i+1,j-1} - 2y_{i+1,j} + y_{i+1,j+1} \right) && \forall i \in I', j \in J' \\
& y_{0,j} = 0 && \forall j \in J \\
& y_{i,2} - 4y_{i,1} + 3y_{i,0} = 0 && \forall i \in I \\
& {}^1\!/_{2\Delta_x} \left(y_{i,n-2} - 4y_{i,n-1} + 3y_{i,n} \right) = u_i - y_{i,n} && \forall i \in I \\
& -1 \leq u_i \leq 1 && \forall i \in I \\
& 0 \leq y_{i,j} \leq 1 && \forall i \in I, j \in J
\end{aligned}
$$

...into this:

```
type QuadProgData
    Q::SparseMatrixCSC{Float64,Int}
    c::Vector{Float64}
    A::SparseMatrixCSC{Float64,Int}
    lb::Vector{Float64}
    ub::Vector{Float64}
    l::Vector{Float64}
    u::Vector{Float64}
end
```



http://blogs.sas.com/content/operations/2015/06/25/finding-the-beauty-in-optimization-models-visualizing-mps-data-sets/

Algebraic modeling

- Very useful for medium-to-large scale constrained optimization
- Automates a process that is
  - Tedious
  - Error-prone
  - Opaque
  - Not portable
- The goal is a direct translation of math to valid code

JuMP vs. Convex

JuMP

- Larger-scale
- Complex indexing
- Advanced solver features
- Autodiff for exact derivatives

Convex

- Reformulates complex nonlinear expressions
- Automatic convexity proof

Why Julia?

- C FFI lets us work with solvers in memory
- Macros let us build up models efficiently (avoid operator overloading and copying)
- Multiple dispatch allows an easy, generic interface layer (MathProgBase)
- Performance (duh)
- A real programming language for data wrangling, post-processing, etc.

More information and examples

- JuliaOpt
- Intro to Convex.jl
- Convex.jl example
- JuMP example