Why your next project should use Julia:

A high-level dynamic programming language designed for high-performance scientific computing

Eric Ford
Department of Astronomy & Astrophysics
Center for Exoplanets & Habitable Worlds
Center for Astrostatistics
Institute for CyberScience
Penn State
October 6, 2015
What Limits our Science?

• Quality of Data
  – Precision, accuracy, what’s being measured
  – Size of Dataset

• Quality of Models
  – Resolution/number of particles in simulation
  – How much physics is included in model
  – Assumptions/approximations adopted

• Quality of Analysis and Interpretation
  – Person-hours devoted to analysis
  – Efficiency of time devoted to analysis
How can a Programming Language Improve Quality of Science?

Improving computing speed enables:

• Analysis of larger datasets
• Increased accuracy & realism of simulations
• Increased sophistication of statistical models
• More rapid interaction with data/models
  – Build intuition for what’s going on
  – Choose next computation wisely
How can a Programming Language Improve Quality of Science?

Increase time (brain cycles) for science by reducing development time (complexity)

- Accelerate initial implementation (High-level language)
- Accelerate exploratory data analysis & debugging (Interactive environment)
- Minimize rewriting of code to do similar things (Generic programming, natural syntax/patterns lead to efficient execution, require minimal changes to parallelize)
- Ease sharing of code (package management system)
- Promote reproducibility via simplified workflow (“good enough” for general purpose programming)
The problem with most programming languages...

“...They're designed by [computer scientists], who tend to worry about things that I don't much care for. Safety, type systems, homoiconicity, and so forth.” - Evan Miller
What were languages designed for?

- Fortran: Technical computing (~1950s)
- C: Writing Unix (~1970s)
- IDL: Science w/o programmers (~1970s)
- R: Statistical analyses (based on S ~ 1976)
- Perl: Scripting to generate reports (1987)
- Java: Hand-held devices, cross-platform (1991)
- Python: System administration (1991)
- Julia: optimized for scientific computing, effective for general-purpose (2012)
What do we need?

A computing environment that is:

- Efficient for scientific computing (like C/Fortran)
- Useful for general purpose computing (like Python)
- Interactive exploratory data analysis (like IDL, python, R, Matlab, Mathematica…)
- Able to run on highly-parallel hardware (like C/C++/Fortran plus MPI)
- Open source (unlike IDL, Matlab, Mathematica…)
- Able to call existing libraries easily
“Traditional “Solution”  (1/3)

Use best-of-class language for each task:
• Script in tcsh
• Parse input with perl
• Perform main computation in C/Fortran
• Compute statistics in R/matlab
• Plotting in IDL
• Tweak something, repeat (many times)
  a) By hand (slow, error prone)
  b) Write code to automate (build tool or scripts)
Problems with “Traditional Solution”

• Learning/keeping up with several languages

• Interfaces between languages
  – Files/pipes (slow)
  – Direct access (matching memory layouts)

• Complex workflow
  – Many small steps distract from science
  – Hard to remember/reproduce/pass on
  – Error prone
“Contemporary Solution”  (2/3)

• Write program in high-level language (e.g., IDL, Python, R, Ruby…)
• Figure out which parts are slow
• Rewrite slow parts using funky syntax
• Figure out which parts are still slow
• Rewrite still slow parts in C/C++/Fortran
• If want even better performance
  – Repeat, repeat,…
  – Identify opportunities for parallelization
  – Parallelize (likely two different ways depending on number of cores to be used)
Problems with “Contemporary Solution”

• Learning/keeping up with a few languages
• Rewriting code twice, thrice, or more
• Interfaces between languages
  – Matching data structures
  – Complex function calls between languages
  – Non-standardized language features
  – Error prone
• Large-scale parallelization
  – Requires more coding/libraries
  – Another level of complexity
Modern Solution: Julia  (3/3)

- Write entire program/workflow in Julia
- If want to improve performance
  - Profile to find which parts are slow
  - Optimize Julia code
- If want even better performance
  - Identify opportunities for parallelization
  - Parallelize (still in Julia): Often can reuse serial code and one programming pattern whether for multi-core, cluster or cloud
  - Harnessing GPU/Phi still requires recoding
Arithmetic & Linear Algebra

julia> 1*2
2
julia> 1/2
0.5
julia> x = [3,3]
2-element Array{Int64,1}:
 3
 3
julia> A = [1.0 2.0; 2.0 1.0]
2x2 Array{Float64,2}:
 1.0  2.0
 2.0  1.0
julia> A * x
2x2 Array{Float64,2}:
 9.0  9.0
julia> A \\ x
2-element Array{Float64,1}:
 1.0
 1.0
julia> svd(A)
(2x2 Array{Float64,2}:
 -0.707107  -0.707107
 -0.707107  0.707107,
 [2.9999999999999996  1.0],
 2x2 Array{Float64,2}:
 -0.707107  0.707107
 -0.707107 -0.707107)
julia> eigvals(A)
2-element Array{Float64,1}:
 1.0
 3.0
Simple Generic Function

```
julia> ephemeris(transit_num, param::Vector) =
       param[1]+param[2]*transit_num
ephemeris (generic function with 1 method)

julia> ephemeris(3,[1,2])    # arguments are integers
7

julia> ephemeris([1.,2.,3.],[1.,2.])  # args are arrays
3-element Array{Float64,1}:
3.0 5.0 7.0

julia> ephemeris(1.0:3.0,[1.,2.])    # now arg is a range
3.0:2.0:7.0
```
function chisq_model_vs_data(param::Vector, model::Function, x::Vector, y::Vector, sigma::Vector)
    @assert( length(x) == length(y) == length(sigma) > 0)
    chisq = zero(eltype(param))
    for i in 1:length(x)
        predict = model(x[i],param)
        chisq += ((predict-y[i])/sigma[i])^2
    end
    chisq
end

- OR -

function chisq_model_vs_data_python_lovers(param::Vector, model::Function, x::Vector, y::Vector, sigma::Vector)
    chisq = sum(((model(x,param)-y)./sigma).^2)
end
Julia lets you implement algorithm however most natural

```julia
function chisq_model_vs_data(param::Vector,
    model::Function, x::Vector, y::Vector, sigma::Vector)
    @assert( length(x) == length(y) == length(sigma) > 0)
    chisq = zero(eltype(param))
    for i in 1:length(x)
        predict = model(x[i],param)
        chisq += ((predict - y[i]/sigma[i])^2)
    end
    chisq
end
```

- Code runs quickly with either syntax
- For loops often faster than “vectorized” expressions
- Devectorize package can optimize “vectorized” expressions
- Fastest to use expressions that result in calling BLAS

```julia
function chisq_model_vs_data_python_lovers(param::Vector,
    model::Function, x::Vector, y::Vector, sigma::Vector)
    chisq = sum(((model(x,param)-y) ./sigma).^2)
end
```
> P_b_true = 5.729;
> t0_b_true = 781.99;
> param_true = [t0_b_true,P_b_true];
> const sim_trid_list_b = collect(linspace(-125,129,255));
> const true_tt_list_b = ephemeris(sim_trid_list_b,param_true);
> const sigma_tt_b = 0.005*ones(length(true_tt_list_b));
> const sim_tt_list_b = true_tt_list_b +
>    sigma_tt_b.*randn(length(true_tt_list_b));

> @time chisq_model_vs_data(param_true,
>    ephemeris,sim_trid_list_b,sim_tt_list_b,sigma_tt_b)
> 0.016961 seconds (7.60 k allocations: 298.285 KB)
240.02078111666512
> @time chisq_model_vs_data(param_guess,
>    ephemeris,sim_trid_list_b,sim_tt_list_b,sigma_tt_b)
> 0.000060 seconds (2.04 k allocations: 32.031 KB)
240.02078111666512
Using a Julia Package

```julia
> chisq_linear_b(param::Vector) = chisq_model_vs_data(param, ephemeris,sim_trid_list_b,sim_tt_list_b,sigma_tt_b)

> P_b_guess = P_b_true+0.000001*randn()
> t0_b_guess = t0_b_true+0.00001*randn()
> pl_b_guess = [t0_b_guess,P_b_guess]
> chisq_linear_b(pl_b_guess)
240.1443838614597

> Pkg.add("Optim")
> using Optim
> fit_b_output = optimize(chisq_linear_b,pl_b_guess)
```

Results of Optimization Algorithm
* Algorithm: Nelder-Mead
* Starting Point: [781.9899914090659,5.728999443174088]
* Minimum: [781.9898596530603,5.729002151809993]
* Value of Function at Minimum: 239.576124
* Iterations: 61
* Convergence: true
  * |x - x'| < NaN: false
  * |f(x) - f(x')| / |f(x)| < 1.0e-08: true
  * |g(x)| < NaN: false
Julia has potential to be faster than C

- Introspection:
  - Julia code is just like any other variable
  - Enables compilers to make complex optimizations

```
Pkg.add("ForwardDiff")
using ForwardDiff
grad_chisq = ForwardDiff.gradient(chisq_linear_b);
grad_chisq(param_true)
2-element Array{Float64,1}:
 2775.27
-2.32294e5
```

```
fit_b_output = optimize(chisq_linear_b,pl_b_guess,method = :bfgs,
                        autodiff= true);
```

Technically, this uses generic programming and dual numbers, not code introspection, but someday (see ReverseDiffSource.jl).
“I have lots of code written in C/Fortran”

- Julia was designed to interface with C/Fortran without additional overhead or writing wrapper functions.

```julia
julia> ccall( (:clock, "libc"), Int32, ())
3240000
julia> ccall( (:erf,"libm"),Float64, (Float64, ), 1.0) 0.8427007929497149
julia> erf(1.0) 0.8427007929497149
```

But I have lots of code written in...

<table>
<thead>
<tr>
<th>Language</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++:</td>
<td>Cxx.jl</td>
</tr>
<tr>
<td>Python:</td>
<td>PyCall.jl</td>
</tr>
<tr>
<td>R:</td>
<td>RCall.jl</td>
</tr>
<tr>
<td>Java:</td>
<td>JavaCall.jl</td>
</tr>
<tr>
<td>Mathematica:</td>
<td>Mathematica.jl</td>
</tr>
<tr>
<td>Shell:</td>
<td><code>run(</code>&quot;cmd&quot;<code>)</code></td>
</tr>
</tbody>
</table>

More for Javascript, Perl, PHP,...
"I have lots of code written in Python"

julia> Pkg.add("PyCall")  # once to install package
julia> using PyCall    # each time you start julia
julia> @pyimport kplr  # import kplr
julia> client = kplr.API()  # client = kplr.API()
PyObject <API(data_root="/astro/faculty/ebf11/.kplr")>
julia> koi = client[:koi](952.01)  # koi = client.koi(952.01)
PyObject <KOI("K00952.01")>
julia> lcs = koi[:get_light_curves]();
julia> [ lcs[i][:filename] for i in 1:length(lcs) ]
37-element Array{Any,1}:
"/astro/faculty/ebf11/.kplr/data/lightcurves/009787239/kplr09787239-2009166043257_llc.fits"
"/astro/faculty/ebf11/.kplr/data/lightcurves/009787239/kplr09787239-2009259160929_llc.fits"
"/astro/faculty/ebf11/.kplr/data/lightcurves/009787239/kplr09787239-2009350155506_llc.fits"
...
Integrates Visualization into Workflow

• Several Plotting packages, including:
  – Gadfly (native Julia)
  – PyPlot (interface to Python’s matplotlib)
  – Gaston (GnuPlot), Winston (like matlab),...
  – Web interfaces: Bokeh, GoogleCharts, Plotly, Vega

```python
using PyPlot
resid = sim_tt_list_b - ephemeris(sim_trid_list_b, fit_b_output.minimum)
pplt = PyPlot
pplt.clf()                         # Clear the current figure
pplt.errorbar(sim_tt_list_b, resid, yerr=sigma_tt_b, fmt="o")
pplt.title("Sample graph")
pplt.xlabel("time")
pplt.ylabel("residual")
pplt.figure(1)
```
Parallel Julia

- Start Julia with 4 worker processes
  ```bash
  shell> julia -p 4
  julia> addprocs(4)  # add 4 more worker processes
  ```
- Start Julia with worker processes on each of the machines assigned by PBS schedule
  ```bash
  shell> julia --machinefile $PBS_NODEFILE ~/demo.jl
  ```
- Can easily run same code on single workstation, cluster (e.g., ICS-ACI) or cloud environment (e.g., Amazon EC2, Domino)
Simple Parallelization

• Parallel for loops
  ```julia
  julia> @parallel for i in 1:n
          y[i] = f(x[i])
  end
  ```

• Map to Parallel map
  ```julia
  julia> y = map(f, x)  julia> y = pmap(f, x)
  ```

• Data Structures to minimize data movement
  – DistributedArrays
  – SharedArrays
Example: Linear Least Squares Regression
50 measurements for $4 \times 1024^2$ pixels

<table>
<thead>
<tr>
<th>Cores</th>
<th>Wall time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.19</td>
</tr>
<tr>
<td>2</td>
<td>0.91</td>
</tr>
<tr>
<td>4</td>
<td>0.49</td>
</tr>
<tr>
<td>8</td>
<td>0.24</td>
</tr>
<tr>
<td>12</td>
<td>0.18</td>
</tr>
<tr>
<td>16</td>
<td>0.17</td>
</tr>
<tr>
<td>20</td>
<td>0.19</td>
</tr>
</tbody>
</table>

Details: two-pass algorithm, double precision, using Distributed Arrays, benchmarked @ ICS ACI-b
My Julia code isn’t as fast as I hoped…

• Likely due to code that prevents Julia from deducing variable types at compile time.

• Check:
  – Is code broken in small functions?
  – Can function return type be determined at compile time?
  – Do container elements have concrete types?
  – When using non-const global variables, are variable types annotated?
Type Stability

• Unsafe:
  \[ \text{pos}(x) = x < 0 \ ? \ 0 : x \]

• Safe
  \[ \text{pos}(x) = x < 0 \ ? \ \text{zero}(x) : x \]

• In Julia v0.4, there’s a clever macro @code_warntype that highlights in red where variable types are ambiguous
Collections of Concrete Types

• Array of Abstract Types:
  > a = Real[]  
  # typeof(a) = Array{Real,1}
  > if (f = rand()) < .8 push!(a, f) end
  > a
  1-element Array{Real,1}: 0.0711957

• Array of Concrete Types
  > a = Float64[]  
  # typeof(a) = Array{Float64,1}
  > if (f = rand()) < .8 push!(a, f) end
  > a
  1-element Array{Float64,1}: 0.443518
Annotating types of global variables

• Global variable type unannotated:
  julia > global x = 1.0
  julia> @time [ sin(x) for i in 1:1000000 ];
  0.058961 seconds (1.00 M allocations: 22.891 MB, 31.97% gc time)

• Global variable with type annotated:
  julia > global x = 1.0
  julia> @time [ sin(x::Float64) for i in 1:1000000 ];
  0.014731 seconds (2 allocations: 7.629 MB)
Status of Julia

• v0.3.11 is latest stable release
• v0.4.0 is very close (on release candidate #4)
  – Pre-compiling of packages (faster startup)
  – SharedArrays (for multi-core machines)
  – Several behind the scenes improvements that will boost performance (e.g., garbage collection, Nullable arrays, generated functions...)
The Challenge for Julia

• Legacy code bases in other languages
  – Julia eases calling C, R, Python, soon C++
• Developers clinging to what they’re used to
  – Yet-another-language syndrome
• Likely to see improved patterns/packages in coming years for:
  – Parallelizing tightly coupled problems (PDEs)
  – Deploying to cloud
  – GPUs/Accelerators
The Promise of Julia

• Julia is the first viable modern language designed for scientific computing.
• Julia could unite science domain experts and computer scientists (“speed freaks”), so more CS research enhances science.
• Rapidly growing developer/code base
• Exponential growth in packages
• Well-positioned to harness increasing parallelism (both hardware & cloud)
I’m convinced… Now what?

• Download Julia Language: [http://julialang.org/](http://julialang.org/)
• Or try it out in browser: [https://www.juliabox.org/](https://www.juliabox.org/)
• Tutorials/videos: [http://julialang.org/learning/](http://julialang.org/learning/)
• Cheat sheets:
• Julia Language Users @PSU: [l-julia-users@lists.psu.edu](l-julia-users@lists.psu.edu)
• Julia Astro Google Group: [julia-astro@googlegroups.com](julia-astro@googlegroups.com)
Julia is the Future of Scientific HPC

- Speed (like C/Fortran)
- Interactive (like IDL, python, R)
- Familiar mathematical notation (like Matlab)
- Useful for general purpose programming (like Python)
- Natural string processing (like perl)
- Generic programming to write an algorithm just once and apply it to many types (like C++)
- Multiple dispatch to efficiently pick the best method for all of a function’s arguments
- Fast linear algebra (like Fortran)
- Easy for statistical analyses (like R)
- Designed for easy distributed computing (like Hadoop)
- Not required to specify types when we don’t feel like it
- Able to write simple scalar loops that compile to tight machine code (like C)
- Excellent package manager (better than Python)
- Simple to learn
- Open source

http://julialang.org/blog/2012/02/why-we-created-julia/