# A Nontrivial Introduction to Julia
# For Data Science and Scientific Computing

Chris Rackauckas

University of California, Irvine

May 22, 2016

# Quick Introduction

- Third Year Math Ph.D student from MCSB.
- Main interest: stochastic analysis in theoretical (evo-) developmental biology.
    - Stochastic analysis: stochastic/nonautonomous dynamical systems and numerical methods
    - Focus on developmental signaling and cell lineages
- High-Performance Computing and Stats / Machine Learning.
- I run a blog on mathematics and scientific computing:
    - `http://www.stochasticlifestyle.com/`

# Notes About the Workshop

- I expect that you have knowledge of some programming or scripting language.
- Advanced topics will be colored blue
  - I do not expect everyone to get everything
  - I think it's important to expose the "lingo"
- Julia can get very low level and "CSy"
  - I will focus on the path from scripting downwards
- The projects give help for how to get started, but you will need to use the documentation
  - The ultimate goal is that you will learn how to use the documentation / Github / mailing lists / Gitter to get things done in Julia
  - Since Julia is so young, you cannot expect to get "free code" from Stack Exchange all the time!
- Do not be afraid to ask questions
- Take your time with the projects, help each other

# Outline

1. Getting Started in Julia
   1. Why Julia? When to choose Julia?
   2. Installing/Building Julia and setting up an IDE
   3. The package management system / Github
   4. Basic usage: control statements, types, functions
   5. Where to get help: documentation, message boards, etc.
2. Syntax / terminology plus some innards of Julia
   1. Differences from other common languages
   2. Unique Programming Paradigms of Julia
      1. Linear Algebra
      2. Data-Oriented Programming
      3. Macro Programming
      4. Levels of Parallelization
      5. De-vectorization, SIMD, threading, parallelization
      6. Named Functions
      7. Subscoping
      8. LLVM

# Additional Topics

1. Extra Projects
    1. Mathematical Modeling / Dynamical Systems
    2. Data Visualization
    3. Multi-Node HPC (Julia's "MPI")
    4. Interal Investigations
    5. Modules and Packages
    6. Language Bindings
    7. Data Saving and Serialization
2. Additional Topics
    1. Generated Functions
    2. Optimization / Machine Learning
    3. GPGPU / Xeon Phi computing

# Why Julia?

- The purpose of Julia is to solve the "multiple-language problem of scientific computing"
  - MATLAB -> MEX, Python -> Cython, R -> ???
  - These languages require vectorization or C
- Another way to think about Julia: It's the 2015 idea of a good programming language:
  - MATLAB (1984), Python (1991), R (1993), Java (1995), C (1973), Lisp (1958)
  - Scripting and vectorization is succinct for scientific programs
  - JIT compilation works really well (Javascript)
  - Object-oriented programming has some good ideas
  - Functional/Meta programming is very useful for some tasks
  - All the other things you learned in a CS course
- Result: Julia is an LLVM-based JIT compiled language that looks like MATLAB/Python/R, but contains all the fancy guts you learned in computer science

# When to Use Julia?

- Everyday scientific computing / data science scripting
  - The best language for scripts which will need to be optimized
- Library creation for scientific computing
  - Numerical / High-performance computing libraries
  - Exploit "Vectorization" style of scientific scripting
- To speed production of things you would have done in C
- Best optimization, machine learning, automatic differentiation, stochastic differential equations and more packages
- "Data-oriented programming" structure for readability
- Unicode symbols like $\alpha$ to enhance readability
- Replace MATLAB codes for HPCs due to licensing issues
- Use R / Python / C / MATLAB packages in one language
- The Cons of Julia
  - It's still new and rapidly evolving
  - Because of the advanced features, it's seems like it's not as beginner friendly as other scripting languages

# The Users of Julia

1. People who want scripts to be easily optimized to run fast
2. People who want to use fast packages/libraries written in Julia
3. People that need to bind together many different languages
4. People who want to use HPC/multi-node parallelism/GPGPU/Xeon Phi w/o OpenMP, MPI, CUDA...
    1. People who want to use "Big Data" (distributed/shared arrays)
5. People who want to write optimized packages/libraries but want the fast development time of a scripting language
    1. Making "libraries for scripters" that overwrite base function for automatic speedups (i.e. VML.jl, Devec.jl, ParallelAccelerator.jl)
6. People who want to mix procedural and functional programming styles
    1. Writing compilers in Julia (i.e. ParallelAccelerator.jl)
7. Everybody!

# Example of a Julia Success: ParallelAccelerator.jl

- ParallelAccelerator.jl started in June 2015
- It is made and maintained by 6 people from IntelLabs
  - It is a compiler written in Julia
  - It takes standard scripting code ("vectorized") and "does the optimizations an expert C++ coder would do"
    - Eliminates array bounds checking
    - Vectorizes to use SIMD / threading
    - Controls the caching
    - Many other things I may not understand
- To use it, just add "@acc" to the front of a function
  - Users don't have to know why it works to get a MASSIVE (20x+) speedup
  - Can automatically compile code for Xeon Phi. May target OpenACC in the future for GPGPU

Summary: Written in Julia to speed up Julia code. Users don't have to change their code to use it.

# Building / Installing Julia

- The easiest way to get started is to use the Current Release from julialang.org
- Note: for heavy language users you "may" want to get the nightly builds
  - As of May 2016, the nightly build is V0.5 which includes threading
- The other option is to build Julia:
  - GCC version 4.7 or later is required to build Julia
  - The default compiler on CentOS / RHEL (gcc 4.4) is too old to build Julia
  - For large systems (HPC nodes) you will want to increase the number of BLAS threads
  - Instructions for linking with MKL are found on the Julia Github

# Integrated Development Environments

- As of May 2016, the recommended IDE is Atom
  - The Juno team is now actively developing Atom.jl, Blink.jl, ink, atom-julia-client
  - Julia with Atom (Juno) is similar to Rstudio / MATLAB
- Other popular IDEs:
  - IJulia (browser), can work with Atom via Hydrogen
  - Sublime Text with IJulia
  - vim, emacs, etc.

# Example Julia Code

```julia
matrix = [1 2 3 4
          3 4 5 6
          4 4 4 6
          3 3 3 3] #Define the matrix

f(x,y) = 2x+y #Create an inline function
@time for i=1:4,j=1:4 #Two loops in one command
  #update matrix[i,j] = matrix[i,j] + f(i,j)
  matrix[i,j] += f(i,j)
end #end statement is required for control statements like for

Pkg.add("ASCIIPlots") #Only use the first time!
using ASCIIPlots #Load the package
scatterplot(1:4,matrix[end,:],sym='*') #Plot the last row
scatterplot(1:4,matrix[2,:],sym='^') #Plot the second row
package = "ASCIIPlots" #Define a string
println("$package is a super cool package") #Print a string
```

# Example Output

```
0.002328 seconds (82 allocations: 5.281 KB) #Timing result

julia> scatterplot(1:4,matrix[2,:],sym='^') #Plot the second row

        --------------------------------------------------------------
        |                                                           ^ | 14.00
        |                                                             |
        |                                                             |
        |                                                             |
        |                                                             |
        |                                                             |
        |                                                             |
        |                                        ^                    |
        |                                                             |
        |                                                             |
        |                                                             |
        |                                                             |
        |                                                             |
        |                    ^                                        |
        |                                                             |
        |                                                             |
        |                                                             |
        |                                                             |
        |                                                             |
        |^                                                            | 8.00
        --------------------------------------------------------------
        1.00                                                     4.00

julia> println("$package is a super cool package") #Print a string
ASCIIPlots is a super cool package
```

# Basic Types of Julia

- A Vector is a one-dimensional array
- An Array{Type,2}(n,m) is an $n \times m$ array of types
- $1 : n$ forms a “Range type” which can be thought of as the vector $[1, 2, \ldots, n]$.
- A dictionary Dict is a key-value structure
- A tuple is an immutable structure
  - Used for multiple return values
  - Very computationally efficient!
- A function is also a type

# Example Usage of Julia Types

```julia
ran = 1:5 #Define a range
vec = collect(ran) #Turn the range into a vector
vec2=Vector{Float64}(2) #Define a size 2 vector of 64-bit floats
vec2[1]=2; vec2[2]=3   # vec2 = [2,3], ; allows multiple commands
push!(vec2,5) # vec2 = [2,5]
mat = Array{Int64,2}(5,5) # Create an empty 5x5 matrix
mat2 = ones(5,5) #Create a 5x5 matrix of ones
Q,R = qr(mat2) #QR decompose mat2 (returns a tuple) into Q and R
tup = ran,vec,vec2 #Define a tuple of many different objects
tup[2] #This returns vec
dict = Dict("a" => 1, "b" => 2, "c" => 3) #Define a dictionary
dict["a"] #Returns 1
```

# Defining Functions

```
function testFunction(x,y;z=0) #z is an optional argument
  if z==0
    return x+y,x*y #Return a tuple
  else
  return x*y*z,x+y+z #Return a different tuple
  #whitespace is optional
  end #End if statement
end #End function definition
a,b = testFunction(2,2) #Returns 4,4
a2,b2 = testFunction(2,3,z=3) #Returns 18,8
```

# The Package Management System

- Julia's package management system is Github
  - The packages are hosted as Github repositories
  - Julia packages are normally referred to with the ending ".jl"
  - Repositories register to become part of the central package management by sending a pull request to METADATA.jl
- The packages can be found / investigated at Github.com
  - Julia's error messages are hyperlinks to the page in Github

# A Very Quick Introduction to Git/Github

- Git is a common Version Control System (VCS)
- A project is a **repository (repos)**
- After one makes changes to a project, **commit** the changes
- Changes are **pulled** to the main repository hosted online
- To download the code, you **clone** the repository
- Instead of editing the main repository, one edits a **branch**
  - To get the changes of the main branch in yours, you **fetch**
  - One asks the owner of the repository to add their changes via a **pull request**
- Stable versions are cut to **releases**
- The major online server for git repositories is Github
- Github is a free service
- Anyone can get a Github account
- The code is hosted online, free for everyone to view
- Users can open **Issues** to ask for features and give bug reports to developers

# Using Julia's Package Management System

- Standard Usage
  - Pkg.add("Package") clones the repo of the package to your system
  - "using Package" imports the package for use in your code
  - Pkg.update() fetches releases from METADATA.jl
- Advanced Usage
  - Pkg.build("Package") re-compiles a package (if compilation failed)
  - Pkg.checkout("Package",branch) checks out the branch (default: main)
  - Pkg.pin("Package") keeps the package at a specific version / commit
  - Pkg.test("Package") runs the Travis CL unit tests
- The tools for generating, tagging, and publishing a package exist within Julia

# Julia's Documentation

- Julia has a standard inline documentation system which is supported by Read the Docs
    - Inline docstrings can be accessed via ?functionname
- Functions (not methods) are documented in this API by their input/output
- Optional function arguments are denoted by (...)

# Project 1: Setup and Using Packages/Documentation

- Get an IDE up and running
- Install the package Gadfly
- Go to the Github page and find the documentation
- Plot the example Stem plot
- Find the code where Stems is defined
- Search the issues for "Error when using Gadfly"
- Change to JeffBezanson's Working branch
- Now look at the other plotting package Gadfly:
    - Find the Travis CI test coverage percentage
    - Take a look at the Gitter chat
- Go to Google Groups-Julia-users, search for posts about Gadfly
    - What are the names of the people who seem to answer all of the questions?

# Main Differences from Other Languages

`http://docs.julialang.org/en/release-0.4/manual/noteworthy-differences/`

- Julia uses 1-based indexing
- A vector of 1 slot is not a scalar
  - Vector{Type}(size), 1:N, Float64, etc. are **Types**
  - User types and functions are **first-class**
- A variable can hold a function (named functions)
- 2-dimensional arrays are treated as matrices
  - * is matrix multiplication, A\b solves Ax=b
    - These operations are implemented in BLAS, Linpack
  - .*, .+, .>, etc. perform element-wise operations
- Arrays are dereferenced and constructed with [] (i.e. A[i])

# Programming Paradigms of Julia 1: Linear Algebra

- Linear algebra is central to scientific computing and Julia
- Julia's linear algebra syntax follows MATLAB's closely.
  - For lots of linear algebra you can use MATLAB's documentation
- Julia's linear algebra documentation is split between two pages:
  - Multi-dimensional arrays: `http://docs.julialang.org/en/release-0.4/manual/arrays/`
  - Linear algebra: `http://docs.julialang.org/en/release-0.4/manual/linear-algebra/`

# Project 2: Linear Algebra

Use the following documentation page:

- Take two $N \times N$ random matrices, $A$ and $B$.
- Compute $C = AB$ and $D$, the Hadamard product of $A$ and $B$ $(A. * B)$
- Compute the maximum and 2-norm of the last column of $C - D$.
- Make the $2N \times N$ matrix which is $C$ concatenated on top of $D$.
- Find the SVD-decomposition of $D$ and plot the singular-value matrix
- Set blas_num_threads to different values and check the performance (parallelization) of $A * B$.
- Create a 100x100 sparse diagonal matrix with $-2, 1, -2, 1 \ldots$ on the diagonal

# Programming Paradigms of Julia 2: "Data-Oriented" Programming

- Julia does not implement a "full object-oriented" paradigm
  - There is no inheritance, encapsolation (except with Modules), and "data-owned methods"
  - **This is intentional and it will stay this way**
- However, types and multiple dispatch allow for a "Data-Oriented" structure
  - You can think of types as being objects
  - **Functions** call different **methods** depending on the type given
    - This is called multiple dispatch
    - Functions are a type whose fields are methods
  - User types are first class
    - Types can have a field which is another type, allowing for a form of inheritance
  - Standard types include Float64, Vector{Type}, Array{dim,Type}, **Any,** Dataframe ...
    - Functions which take in Any will compile at runtime for the type that it is given

# Project 3: "Data-Oriented" Programming

- Construct a function **solve** which takes in a matrix $A$ and $x_0$ and iterates $x_{n+1} = Ax_n + \epsilon_n$, $\epsilon \sim N(0,1)$.
- Construct a type called **model** which holds a matrix $A$ and a vector $x_0$.
- Create a new method for the solve function which takes in a model and solves it 100 times and returns the mean and variance
- Create a new type called **ParameterSweep** which holds a vector of models
- Create a new method for the solve function which takes in a parameter sweep and returns a vector of tuples with $A$, $x_0$, mean, and variance

# Programming Paradigms of Julia 3: "Macro" Programming

- Macros are implemented by the parser pre-compilation
  - Macros are methods for metaprogramming: programming the program
- Macros can be used to "expand" simple code into more complex code, while maintaining the simple code for readability
- Common macros: @time, @elapsed, @parallel, @devec, @simd, @nobounds, @thread
- Macros can block via: @macro begin #codecodecode end
- Users can write their own macros to avoid having to do repetative programming tasks.

# Project 4: Using Macros

- Write functions which take $N$, generates uniformly distributed $A$, $B$, $C$, and $D$ and solves $A.*B.*C.*D$ in three ways:
    - $A.*B.*C.*D$
    - A for loop
    - A for loop with @simd, @nobounds
    - $A.*B.*C.*D$ with @devec
- Time the functions using tic() and toc()
- Time the functions with @time
- Use @elapsed to time the functions and save the values to a variable
    - Some of your first runs will be weird (the longest!). How can you fix this? Can you find out why this happened?
- Use @benchmark / @benchmarks (Extra: Try ParallelAccelerator.jl)
- Find out what the native Julia implementation is for .*
    - Can you explain why devectorization is fastest?

# Levels of Parallelization

- SIMD parallelization is "processor-level parallelization"
  - The package is developed by Intel
  - It uses the processor's AVX (Advanced Vector Extensions)
    - The compute cores can process more than 1 number at a time!
  - Add @simd to the beginning of an inner loop
    - The values must be able to be re-ordered
- Thread parallelization is shared-memory parallelization
  - Threads share the memory on the same computer
  - Use @thread at the beginning of a threadable loop
- Task parallelization is a general form of parallelization
  - Much easier than MPI, though the same capabilities
    - Can be used on multiple compute nodes on an HPC
  - Memory is not shared, items have to be distributed and passed
  - Either open julia with -p numberOfProcs or use addprocs(n)
    - One process is the control process, the others are workers
  - Use @parallel, pmap, etc. to distribute work to the workers

# Programming Paradigms of Julia 3: Named Functions

- User functions are first-class in Julia
  - This means that they can be modified like base functions
- Julia functions can be named, allowing inline function definitions
  - $\nabla f(x) = 5x$ is a valid function definition
- This can be useful for defining "subfunctions" which perform a repetative task.
- Base functions can be overloaded with new functionality
  - Packages can use this to make "fast" versions
  - Example: VML.jl

# Project 5: Named Functions

- Extend the code from project 4 using Vector{Float64}(0) and push! to get the timings of $A. * B. * C. * D$ for $N = 2^n$ with $n = 1 : 10$.
- Write a line of code which takes the array of times to produce a plot via Gadfly.
- Extend this to an inline function plotTimes(timeArr,titStr) which takes in a time array and a titStr and produces the appropriate plot.
- Use this function to plot the timings.
- Extend the plotting function to take in 4 arrays and plot them on the same graph using "layer()"

# Programming Paradigms of Julia 4: Subscoping

- Functions take the default values of their current scope.
- Variables defined in a higher scope "give their value" to lower scope by default
- **The highest scope is automatically global**
  - This is the single largest cause of slowdowns for first-time Julia users

```julia
x=5; y=7; #Defined globally
function scopeTest(z)
  x += z #Changes global value
  y = Vector{Float64}(1) #Declares a variable, local scope
  y[1] = 2
  return x + y + z
end
```

# Project 6: Scoping Caution

```
function f1()
  @parallel for i = 1:100
    var = 10
    if var < 100
      var = var + 1
    end
  end
  var = 100 + 10
end
f1()
function f2()
  @parallel for i = 1:100
    var = 10
    if var < 100
      var = var + 1
    end
  end
end
f2()
```

# Quick Introduction to LLVM

- LLVM is the compiler the Julia on top of
  - It is an open source project that many other languages use
- Many functionalities of Julia are developed via the LLVM
  - Naitive LLVM compilation of CUDA kernals and Xeon Phi code will soon be part of Julia base
- You can check your LLVM code via @code_llvm, @code_native
  - One thing to look for is automatic SIMD constructs
  - Also look out for "too many extra things"

# Modeling Project: Logistic Curve (Medium)

The logistic difference equation is defined by the recursion

$$b_{n+1} = r * b_n(1 - b_n)$$

where $b_n$ is the number of bunnies at time $n$. Starting with $b_0 = .25$, by around 400 iterations this will reach a steady state. This steady state (or steady periodic state) is dependent on $r$. Write a function which solves for the steady state(s) for each given $r$, and plot every state in the steady attractor for each $r$ ($x$-axis is $r$, $y$ =value seen in the attractor) using PyPlot. Take $r \in (2.9, 4)$ Optimize this function.

# Modeling Solution

```julia
function logisticPlot()
  tic()
  r = 2.9:.00005:4; numAttract = 100;
  steady = ones(length(r),1)*.25;
  for i=1:400 ## Get to steady state
    @devec steady = r.*steady.*(1-steady);
  end
  x = zeros(length(steady),numAttract);
  x[:,1] = steady;
  @simd for i=2:numAttract ## Grab values at the attractor
    @inbounds @fastmath x[:,i] = r.*x[:,i-1].*(1-x[:,i-1]);
  end
  toc()
  fig = figure(figsize=(20,10));
  plot(collect(r),x,"b.",markersize=.06)
  savefig("plot.png",dpi=300);
end
using PyPlot
@time logisticPlot()
```

# Data Visualization and DataFrames Project (Easy)

Follow along with the Gadfly + RDatasets demo at:
`https://github.com/timholy/gadfly/blob/master/demo.md`
Or try Plots.jl, PyPlot.jl, or Plotly.jl.

# HPC Multi-Node Parallelization (Easy)

- Use the job script to call a test.jl script on the cluster:

```
#!/bin/bash
#$ -N jbtest
#$ -q <CHOOSE A QUEUE>
#$ -pe mpich 128
#$ -cwd             # Run in current directory
module load julia/0.4.3
julia --machinefile jbtest-pe_hostfile_mpich.$JOB_ID test.jl
```

- Make sure that all of the compute nodes are being used

```
hosts = @parallel for i=1:120
        run(`hostname`) end
```

- Write a simple parallel loop (rand > .5 with (+) reduction), SSH into the nodes and use htop to check usage
- Information for other HPCs can be found here: http://www.stochasticlifestyle.com/multi-node-parallelism-in-julia-on-an-hpc/

# Interal Investigations Project (Quick, but Hard)

- Learn about the @code_native and @which macros here:

https://sinews.siam.org/DetailsPage/tabid/607/ArticleID/744/Julia-A-Fast-Language-for-Numerical-Computing.aspx

- Test the @code_llvm macro on some of your codes. What changes when you use @simd? Look for <Vector >s
- Try apropos, ?, @which to find functions and check dispatches
- On one of the functions previously written, use the fieldnames() function to find the function pointer

# Modules and Packages (Medium)

- Wrap the functions from Project 4/5 into a module
- Export the functions in the module
- Import the module and test the functionality
- Generate a package via Pkg.generate("Name")
- Put the module into the package, test the package
- Setup the package to pre-compile during the first build

For more information on how to "complete" a package, see `http://www.stochasticlifestyle.com/finalizing-julia-package-documentation-testing-coverage-pub`

# Language Bindings (Hard)

1. C
   1. Follow along here: `http://www.stochasticlifestyle.com/using-julias-c-interface-utilize-c-libraries/`
      1. If GSL is not available, Write a Hello World script in C which takes in an array of numbers and also prints the numbers.
      2. Write a Julia function which sorts the numbers, compile the function to a C function, and use the C calling interface to call the function from a C code

# Language Bindings Continued (Medium)

1. Python
   1. Use the following package to import Python packages: `https://github.com/stevengj/PyCall.jl`
2. MATLAB
   1. Check out the following: `https://github.com/JuliaLang/MATLAB.jl`
      1. Note that this requires you to NOT be on Windows
      2. Test some of the examples on the Github demo.
      3. For a more advanced introduction, try porting Finite Element Method code as described here: `http://www.stochasticlifestyle.com/julia-ifem-1-mesh-generation-and-julias-type-system/`
3. R
   1. To call R with Julia, test the following package: `https://github.com/lgautier/Rif.jl`
   2. To call Julia with R, test the following package: `https://github.com/armgong/RJulia`

# Data Saving and Serialization (Medium)

- Try the HDF5 package for .jld and .mat files: `https://github.com/JuliaLang/HDF5.jl`
- Try the serialize() and deserialize() functions for high-performance serialization (note: the official mantra for future compatability is "people will try not to break the serialization format, but you shouldn't depend upon on it." Use .jld and .mat for long-term data storage)

# Generated Functions (Hard)

- Generated functions are Macros which have type information.
  - More general than macros
- Information can be found on the metaprogramming page:
  - `http://docs.julialang.org/en/release-0.4/manual/metaprogramming/`

# Optimization / Machine Learning (Medium)

- An extensive modeling framework for optimization is available via JuMP
  - This package lets one easily write problems and switch out solvers to find the best solver method
  - There are currently 13 available solvers
  - `https://jump.readthedocs.org/en/latest/`
- Other interfaces are available via Ipopt, NLopt, etc.
- The homepage for the JuliaOpt group can be found here: `http://www.juliaopt.org/`
  - JuliaOpt has its own mailing list, Google Groups / Julia-opt

# GPGPU / Xeon Phi Computing (Medium)

- For tutorials on using the CUDA Runtime for GPGPU computing, see the following:
- `http://www.stochasticlifestyle.com/julia-on-the-hpc-with-gpus/`
- `http://www.stochasticlifestyle.com/multiple-gpu-on-the-hpc-with-julia/`
- The group JuliaGPU has a repository of the available packages:
- `https://github.com/JuliaGPU`
- Users can be found here: `https://gitter.im/JuliaGPU/meta`
- Xeon Phi linking can be found here: `http://www.stochasticlifestyle.com/interfacing-xeon-phi-via-julia/`
    - Native Xeon Phi linking can be found in the ParallelAccelerator.jl package (extremely new)