Julia for Machine Learning

Jake Snell
University of Toronto

Machine Learning Group Tutorial
May 2, 2014
language

from Latin “lingua” (tongue)

- a system for the expression of thoughts, feelings, etc, by the use of spoken sounds or conventional symbols
Desiderata

- Syntax looks like pseudocode
- Vocabulary to talk about data & operations
- Large standard library
- Good performance
• High-level
• Dynamic type system
• Performance approaching statically-compiled languages
• Metaprogramming
• Parallelism
• Good interop with other languages
• MIT licensed
Outline

• Motivation
• Background
• Syntax
• Type system
• Speed
• Features
• Community
• Conclusion
Background
# Technical Computing Landscape

<table>
<thead>
<tr>
<th></th>
<th>Matlab</th>
<th>Python</th>
<th>Julia</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Introduced</strong></td>
<td>1984</td>
<td>1991*</td>
<td>2012</td>
</tr>
<tr>
<td><strong>Creators</strong></td>
<td>MathWorks</td>
<td>Guido van Rossum</td>
<td>Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman</td>
</tr>
<tr>
<td><strong>License</strong></td>
<td>Proprietary</td>
<td>BSD-style</td>
<td>MIT</td>
</tr>
<tr>
<td><strong>Name</strong></td>
<td>“Matrix Laboratory”</td>
<td>Monty Python</td>
<td>?</td>
</tr>
</tbody>
</table>

*NumPy introduced in 1995 as Numeric and 2006 as NumPy*
Syntax
Mandelbrot

function mandel(z)
    c = z
    maxiter = 80
    for n = 1:maxiter
        if abs(z) > 2
            return n-1
        end
        z = z^2 + c
    end
end
return maxiter
end
function randmatstat(t)
    n = 5
    v = zeros(t)
    w = zeros(t)
    for i = 1:t
        a = randn(n,n)
        b = randn(n,n)
        c = randn(n,n)
        d = randn(n,n)
        P = [a b c d]
        Q = [a b; c d]
        v[i] = trace((P'*P)^4)
        w[i] = trace((Q'*Q)^4)
    end
    std(v)/mean(v), std(w)/mean(w)
end
Fibonacci

\[ \text{fib}(n) = n < 2 \ ? n : \text{fib}(n-1) + \text{fib}(n-2) \]

\[ \text{fib}(20) \ # \Rightarrow 6765 \]
Data Structures

- Vectors
- Matrices
- Strings
- Tuples
- Dictionaries
- Sets
- …
b = [4, 5, 6]
b[1] # => 4
b[end] # => 6

matrix = [1 2; 3 4]

tup = (1, 2, 3)
tup[1] # => 1
tup[1] = 3 # => ERROR

dict = ["one"=> 1, "two"=> 2, "three"=> 3]
dict["one"] # => 1

filled_set = Set(1,2,2,3,4) # => Set{Int64}(1,2,3,4)
try
  error("help")
catch e
  println("caught it $e")
end
# => caught itErrorException("help")
Strings

split("wow look at these words")
# => 5-element Array{SubString{ASCIIString},1}:
  "wow"
  "look"
  "at"
  "these"
  "words"

join("
  "We invited the rhinoceri", "Washington",
  "and Lincoln"], ",", ")
# => "We invited the rhinoceri, Washington, and Lincoln"
RegExes

```ruby
ismatch(r"^\s*(?:#|$)" , "# a comment")
# => true

m = match(r"(a|b)(c)?(d)" , "acd")
# => RegExMatch("acd", 1="a", 2="c", 3="d")

m.captures
# => 3-element Array{Union(SubString{UTF8String},Nothing),1}: "a" "c" "d"
```
Comprehensions

\[ [i + j \text{ for } i = 1:3, j = 1:5] \]
# => 3x5 Array\{Int64,2\}:

2 3 4 5 6
3 4 5 6 7
4 5 6 7 8

\[ [i \Rightarrow \text{char(64 + i) for } i = 1:5] \]
# => [5=>'E',4=>'D',2=>'B',3=>'C',1=>'A']
FP-style shenanigans

map(x -> x * 2, [1, 2, 3])
# => [2, 4, 6]

reduce(*, 1, [1, 2, 3])
# => 6

map(s -> (s, length(s)), subsets([1, 2, 3]))
# => [([], 0), ([1], 1), ([2], 1), ([1, 2], 2), ([3], 1), ([1, 3], 2), ([2, 3], 2), ([1, 2, 3], 3)]
Matrix Operations

trace
det
eigfact
sparse
inv
kron
...

Shell Commands

```shell
a = readall(`echo hello`)  
# => "hello\n"

file = "/etc/passwd"  
# => "/etc/passwd"

`sort $file`  
# => `sort /etc/passwd`

run(`echo world` & `echo hello` | `sort`)  
# => hello world
```
Coroutines

```python
function producer()
    produce("start")
    for n=1:4
        produce(2n)
    end
    produce("stop")
end

for x in Task(producer)
    println(x)
end
```

# => start
2
4
6
8
stop
Type System
Julia’s Type System

- Dynamic
  - No “compile-time type”
  - Only values, not variables, have types
- Nominative
  - Relationships between types explicitly declared
  - Concrete types are final
- Parametric
  - Abstract and concrete types can be parameterized by other types & certain values
Multiple Dispatch

• All functions in Julia are generic
  • First-class objects
  • Can be passed around
  • Can be extended
• Function definition that is called depends on the types of all its argument
Multiple Dispatch Demo

- Notebook from Stefan Karpinski’s talk at Strange Loop 2013
Speed
Traditional Language Split

- Prototype in high-level language
- Write performance critical code in a low-level language e.g. C or Fortran
- Tie together with
  - Mex
  - Ctypes
  - Cython
  - SWIG
  - ...

...eating our cake too?

**Figure:** benchmark times relative to C (smaller is better, C performance = 1.0).
JIT

• Julia compiles the code it needs at run-time
  • JIT = “just in time”
  • Translates each method into machine code
• Utilizes LLVM for optimization and code generation.
• Originally “Low level virtual machine”, now LLVM is the full name of the project
• Collection of modular compiler and toolchain technologies
• Introduced by Vikram Adve and Chris Lattner at University of Illinois in 2003
• Used by Apple as part of dev tools for Mac OS X and iOS
Traditional Approaches

Three-phase compiler

Code → Bytecode Interpreter → Effect

Code → Compiler → Bytecode → Interpreter
LLVM Design

LLVM uses a common code representation
LLVM IR

- IR = intermediate representation

```c
unsigned add1(unsigned a, unsigned b) {
    return a+b;
}
```

```llvm
define i32 @add1(i32 %a, i32 %b) {
  entry:
    %tmp1 = add i32 %a, %b
    ret i32 %tmp1
}
```
Julia’s JIT Pipeline

Code

Parsed AST
:(), quote

Lowered AST
code_lowered()

Optimized AST
code_typed()

LLVM IR
code_llvm()

Assembly
code_native()
Types Help to Generate Efficient Code

- LLVM and types demo
Side benefit

• Since Julia is fast, most of Julia is written in itself
• You can learn by poking around source of Julia and its standard library
• Easy to contribute core components
Other JITted systems

• PyPy
  • Tracing JIT (vs method-at-a-time JIT)
  • No support for Numpy
• Numba
  • NumPy compatible, based on LLVM
  • Uses NumPy type information for inference
  • Doesn’t remove dynamic indirection for less well-typed ordinary Python code
• Pyston
  • Announced earlier this month by Dropbox
  • Still in early phases, far from release
Metaprogramming
Homoiconicity

- Code lives in data structures that can be manipulated by the language itself.
- In Julia’s case, Expr and Symbol types.
type Expr
    head::Symbol
    args::Array{Any, 1}
typ
end

ex = :(a+b*c+1)
# => :(+((a,*(b,c),1)))

typeof(ex)
# => Expr

ex.head
# => :call

ex.args
# => [:+, :a, :(*(b,c)), 1]
Macros

- Special functions to directly manipulate expressions

```julia
macro assert(ex)
    :($ex ? nothing : error("Assertion failed: ", $(string(ex))))
end

@assert 1==1.0
# =>

@assert 1 == 0
ERROR: assertion failed: 1 == 0
    in error at error.jl:21
```
Para elism
# parfor.jl

```
@time begin
    nheads = @parallel (+) for i=1:2000000000
        int(randbool())
    end
end
```

$ julia parfor.jl
elapsed time: 10.333040655 seconds (6323888 bytes allocated)

$ julia -p 8 parfor.jl
elapsed time: 2.505858567 seconds (13534036 bytes allocated)
# pmap.jl
M = \{rand(1000,1000) for i=1:10\}  
@time pmap(svd, M)

$ julia pmap.jl
elapsed time: 7.620465569 seconds (575974660 bytes allocated)

$ julia -p 8 pmap.jl
elapsed time: 4.206753903 seconds (524003124 bytes allocated)
Cluster Computing

• Workers don’t have to be on the local machine
  • Passwordless SSH
  • ClusterManagers.jl: Sun Grid Engine, …
  • AWS.jl: interface to Amazon Web Services (EC2, S3)
And more

- Distributed arrays for splitting large matrices across workers
- Primitives for pushing data back and forth
- @sync, @async
Interop
function getenv(var::String)
    val = ccall( (:getenv, "libc"),
                Ptr{Uint8}, (Ptr{Uint8},), bytestring(var))
    if val == C_NULL
        error("getenv: undefined variable: ", var)
    end
    bytestring(val)
end

getenv("SHELL")
# => "/bin/bash"
Calling Python

- PyCall.jl offers automatic conversion of types between Julia and Python
  - numeric, boolean, string, functions
  - tuples, arrays, and dictionaries of above
- Julia arrays are converted to NumPy arrays without making a copy
- Demo
function thinboundary(bmap::BitMatrix)
    @mput bmap
    @matlab bmapthin = bwmorph(bmap, "thin", inf)
    convert(BitArray, @mget bmapthin)
end
Calling C++

- More difficult
- Usually easiest to create thin C-language wrapper around the code you want to call
- `Cpp.jl` handles ABI name-mangling, but C++ objects cannot be converted

``` differentiated
int timestwo(int x) {
    return 2*x;
}

x = 3.5
x2 = @cpp ccall((:timestwo, libdemo), Float64, (Float64,), x)
```
Calling Julia

- From C/C++
  - Use Julia’s C API
- From MATLAB
  - julia-matlab package
  - Can write performance critical code in Julia without resorting to MEX
- From Python
  - pyjulia: still experimental at this point
Interactivity
Environments

- REPL
- IJulia notebook
  - http://nbviewer.ipython.org to share notebooks
- Forio Julia Studio
  - Visual IDE
Plotting

• Several options
  • Gadfly.jl
    • similar to ggplot2
    • influenced by Leland Wilkinson’s “Grammar of Graphics”
  • Winston.jl
    • similar to Matplotlib, still light on features
  • Gaston.jl
    • interface to gnuplot
• No clear winner yet
PyPlot.jl

• Wrapper around Matplotlib’s API

using PyPlot
x = linspace(-2pi, 2pi)
y = sin(x)
plot(x, y, "--b")
Community
Adoption

- Already used in courses at:
  - Stanford
  - Penn State
  - Cornell
  - MIT
  - Western
Community

- Small but active, helpful, growing
- Strong open-source culture
  - Design discussions take place in the julialang repo on Github

julia-users, posts from June 2013 - present
Package Manager

- git based, integrated with Github
- Installing a package
  Pkg.add("Images")
- Creating a package for local development
  Pkg.generate("FooBar", "MIT")
- When you’re ready to publish to Github
  Pkg.register("FooBar")
- To make your package visible to everybody, submit a pull request to the METADATA repo
Lots of useful packages

- Images.jl
- Graphs.jl
- DataFrames.jl
- DimensionalityReduction.jl
- Distributions.jl
- NLOpt.jl
- ArgParse.jl
- Logging.jl
- FactCheck.jl
- Many more - check out https://github.com/JuliaLang/METADATA.jl for the full list
GPU Computing

- At this point: mostly low-level API wrappers
  - CUDA.jl
  - CUDArt.jl
  - OpenCL.jl
More on the way

- **Google Summer of Code 2014 projects**
  - Julia wrappers for high performance GPU programming
  - Computer vision using OpenCV
  - Julia frontend for Halide, an image processing language
Assessment
Desiderata

• Syntax looks like pseudocode
• Vocabulary to talk about data & operations
• Large standard library
• Good performance
Reasons to choose Julia

• Fast prototyping without sacrificing speed
• Easy to parallelize code
• Types make expressing algorithms simpler
• Plays well with other languages
• Standard library written in Julia
• Friendly and helpful community
Reasons not to choose Julia

• You are primarily a NN researcher (not just a user)
  • GPU infrastructure not quite in place
• You need to write production code
  • Language is still growing, interfaces may change
• You want to write the next NLTK or other widely used package
  • May want to go with a more popular language
Resources
How to Try

- Try Julia online at http://forio.com/julia/repl/
- To install:
  
git clone https://github.com/JuliaLang/julia
  cd julia
  make (or make -j N, where N is your desired number of parallel processes)
  # ln -s $PWD/julia /usr/bin/julia
Resources

• Julia Manual
• Julia Standard Library
• Learn Julia in Y minutes
• User groups
  • julia-users
  • julia-dev
  • julia-stats
• Talks
  • Julia Tutorial at MIT, Jan 2013
  • Stefan Karpinski @ Code Mesh 2013
• Read the source, Luke!
Helpful Commands

?map
  • display documentation for the map function
apropos("reduce")
  • show all functions with the term “reduce” in their documentation
methods(+)
  • display all instantiations of generic + function
methodswith(BigInt)
  • display all functions involving BigInts
fin