SKILLPILLS

Skill Pill: Julia

Lecture 1: Introduction

James Schloss Valentin Churavy

Okinawa Institute of Science and Technology
james.schloss@oist.jp
valentin.churavy@oist.jp

July 4, 2017

OIST

Installation

Windows, Linux, and Mac OSX

Download a precompiled version of 0.6 from
`https://julialang.org/downloads/`

Linux and Mac OSX

1. Use your package manager (Mac OSX: `brew cask install julia`)
2. Follow the build instructions from
   `https://github.com/JuliaLang/julia/`

Sango and Tombo

OIST has Julia installed on Sango and Tombo, in case the version you need is not there let it-help@oist.jp know.

Resources

Documentation	`https://docs.julialang.org/en/release-0.6/`
Forum	`https://discourse.julialang.org`
Issue Tracker	`https://github.com/JuliaLang/julia`
Downloads	`https://julialang.org/downloads/`
Packages	`https://pkg.julialang.org/`
	`https://juliaobserver.com/`

Why does Julia exist?

Why does Julia exist?

Hypothesis

1. Scientist like to work in a high productivity programming language.
2. Eventually the problem size will increase and computational intensive

Why does Julia exist?

Hypothesis

1. Scientist like to work in a high productivity programming language.
2. Eventually the problem size will increase and computational intensive

The old mission statement is available at `https://julialang.org/blog/2012/02/why-we-created-julia` and some good discussion is available at `https://discourse.julialang.org/t/julia-motivation-why-werent-numpy-scipy-numba-good-enough/2236` especially Stefan's comment at `https://discourse.julialang.org/t/julia-motivation-why-werent-numpy-scipy-numba-good-enough/2236/10`.

Why does Julia exist?

Hypothesis

1. Scientist like to work in a high productivity programming language.
2. Eventually the problem size will increase and computational intensive

The old mission statement is available at `https://julialang.org/blog/2012/02/why-we-created-julia` and some good discussion is available at `https://discourse.julialang.org/t/julia-motivation-why-werent-numpy-scipy-numba-good-enough/2236` especially Stefan's comment at `https://discourse.julialang.org/t/julia-motivation-why-werent-numpy-scipy-numba-good-enough/2236/10`.

My personal reason

A fast, elegant, high level language that is fast enough to do serious numerical work on a super computer, while also having a language design

The other contenders

The typical languages used in science are

1. Python
2. Matlab
3. R

Once a problem is becoming to big we usually move to

1. C/C++
2. Fortran

This is called the 2+ language problem and Julia is trying to solve that.

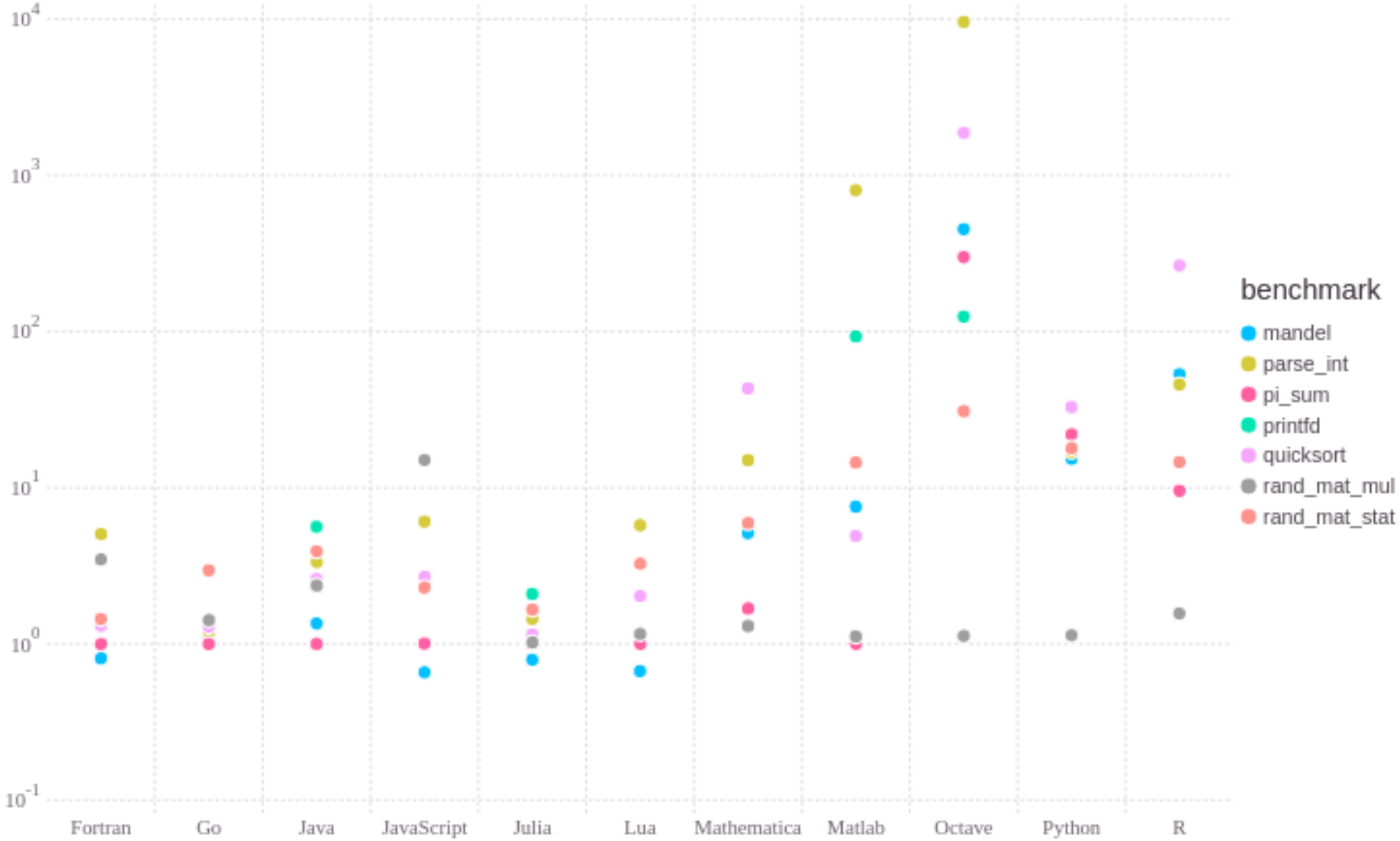
Python and Numpy

- Object are essentially dicts and can be changed at runtime.
- The compilers that exist (Numba) only work on primitive types and not user defined ones.
- GIL (Global Interpreter Lock) mask multi-threading hard.
- For fast code you need to write it in C.
- Numpy is great, but awful syntax for math.

Matlab

- It costs alot of money and is not open-source.
- Matlab will only be fast for a subset of operations.
- Matlab tends to hide the computer from the programmer.

A (biased) performance comparision



The Read-Eval-Print-Loop

The REPL is a command-line interface to Julia and is ideal for short experiments.

```
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: https://docs.julialang.org
   _ _   _| |_  __ _   |  Type "?help" for help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.6.0 (2017-06-19 13:05 UTC)
 _/ |\__'_|_|_|\__'_|  |
|__/                   |  x86_64-pc-linux-gnu

julia>
```

In the REPL you can use ? to switch your REPL mode into help mode and get information about functions.

IDEs

There are two main IDEs that are *feature* complete and can be used for Julia. The main one is based on Atom and is called Juno `http://junolab.org/`.
The second one is based on on Visual Studio Code and available at `https://marketplace.visualstudio.com/items?itemName=julialang.language-julia`.
I do not use either of them, but if that is the kind of environment you like and are used to give them a try.

Jupyter

Jupyter is an interactive web-based client for Python, Julia, R and many other languages. It offers a programming environment that is well suited for explorative data analysis or prototyping.

Installation

```
julia> ENV["JUPYTER"] = ""
julia> Pkg.add("IJulia")
```

Starting a Jupyter session

```
julia> using IJulia
julia> notebook()
```

JuliaBox

There is an online service provided by JuliaComputing at `https://juliabox.com` that gives you a cloud version of Jupyter.

Variables and datatypes

Julia is a dynamic language and so you can simply create variables in any scope.

```
x = 1  # x will be of type Int64
y = 1.0 # y will be of type Float64
z = 1.0 - 2.0im # z will be an Complex{Float64}
1//2 # Rational numbers
"This is a String"
"""
This is a multiline
String
"""
'C' # Character literal
1.0f0 # Float32 literal
```

Use `typeof` to check the type of any variable. Variable names can be unicode and so greek symbols can be used. In the `REPL` and most editors you can insert them by entering their LaTeXname and press `[Tab]`.

Conditionals

Julia has all the typical conditionals `if`, `else`, `ifelse` which have to end in an `end`. Blocks in Julia are not whitespace sensitive and conditionals do not need to be wrapped in round brackets.

```
if rand() < 0.5
  println("Hello there!")
else
  println("Go away!")
end
```

Loops

Julia has `for` and `while` loops. A `while` loop takes a condition and a `for` loop takes a iteratior. One can use `break` to break out of a loop and `continue` to skip to the next iteration. It is noteworthy that a `for` loop can take an arbitrary iterator and even desugar tuples.

```
for (i, x) in enumerate(['A', 'B', 'C'])
  if x == 'B'
    continue
  end
  println(i)
end

while true
  # ternary operator ?!
  rand() < 0.1 ? break : println("You are trapped!")
end
```

Functions and lambdas

Julia uses functions not scripts to organise operations. Every function is compiled for the combination of input parameters.

```
"""
    f(x, y)

`f` will add two numbers together.
"""
function f(x, y)
  return x + y
end

g(x) = x^2

h = (x)-> 1/x
map(lowercase, ['A', 'B', 'C'])
map((x)->x+2, [1, 2, 3])
```

Types

Julia's type system allows you to restrict functions to certain types and specialise functions for others. You can also create your own types. The names of types are typicaly captialised while functions are lowercase.

```
abstract type Entity end
mutable struct Player <: Entity
  mass::Float64
  name::String
  position::Tuple{Float64, Float64}
end
struct Object <: Entity
  position::Tuple{Float64, Float64}
end
```

Multiple dispatch in a nutshell

Julia is not an Object-Oriented programming language functions do not belong to an object. A function is a set of multiple methods each with their own signatures. When you call a function the most specific methods is executed.

```
function h(x::Number)
  println("x is most definitly a number.")
end
function h(x::Integer)
  println("x is a integer")
end
function h(x::Int8)
  println("Specific method for Int8")
end
```

This becomes really powerful when having multiple argurments and being able to select the most specific method.

Modules

Julia code is organised as modules (namespaces). Module names are capitalised and you can nest modules as well.

```
module MyModule
  export f

  g() = "Internal function"
  f() = println(g())
end
using MyModule
```

Installing packages

Julia has an inbuilt package manager called `Pkg`. Julia packages end in `.jl` and it is customary to refer to them by their full name online, but within Julia you drop the `.jl`. So to install the Julia package `Distributions.jl` in the `REPL` just run:

```
Pkg.add("Distributions")
```

A few other commands:

```
# Updating the installed packages
Pkg.update()
# What packages are installed?
Pkg.status()
```

In order to create your own packages you have to install `PkgDev.jl`.

Github

Most of the development of Julia packages and the base language happens on Github.
Check out `https://github.com/JuliaLang/julia` for the main action.

Using Fortran and C in Julia

Julia allows you to use other languages (such as Fortran or C) by using the `ccall` function:

```
julia> t = ccall((:clock, "libc"), Int32, ())
2292761
```

Here, we are calling the `clock` function from the `libc` library in C.

Your legacy code

Let's say you want to use a simply multiply function in Fortran:

```
!! We'll be using subroutines intead of functions
subroutine multiply(A, B, C)
    REAL*8 :: A, B, C
    C = A * B
    return
end
```

or C:

```
// Nothing fancy here...
double multiply(double A, double B){
    return A*B;
}
```

Preparing your legacy code

In order to use your favorite C or Fortran code in Julia, you need to compile it into a library, like so:

```
gcc -shared -O2 multiply.c -fPIC -o c_multiply.so
gfortran -shared -O2 multiply.f90 -fPIC -o
    fortran_multiply.so
```

Preparing your legacy code

In order to use your favorite C or Fortran code in Julia, you need to compile it into a library, like so:

```
gcc -shared -O2 multiply.c -fPIC -o c_multiply.so
gfortran -shared -O2 multiply.f90 -fPIC -o
    fortran_multiply.so
```

These will create libraries with all of the necessary functions you could want, but beware:

C and Fortran compilers mangle function names!

Using your legacy code

There are 3 things to keep. Make sure you

1. Have the right mangled name
2. Are using the right type
3. Are using the function correctly.

Using your legacy code

There are 3 things to keep. Make sure you

1. Have the right mangled name
2. Are using the right type
3. Are using the function correctly.

For example, in C:

```
# This function multiplies a and b into c by using the
# created C library
function call_c()
    a = Cdouble(1.0)
    b = Cdouble(3.0)
    c = ccall((:multiply, "/full/path/to/c_multiply"),
        Cdouble,(Cdouble, Cdouble),a,b)
    println(c)
end
```

Using your legacy code

Pointers are okay! For example, in Fortran:

```
# This function multiplies a and b into c by using
# the created FORTRAN library
function call_fortran()
    a = Cdouble[1.0]
    b = Cdouble[2.0]
    c = Cdouble[0.0]
    ccall((:multiply_, "/full/path/to/fortran_multiply"),
       Void,(Ptr{Float64},Ptr{Float64},Ptr{Float64}),
             a,b,c)
    println(c[1])
end
```

Using your legacy code

Pointers are okay! For example, in Fortran:

```
# This function multiplies a and b into c by using
# the created FORTRAN library
function call_fortran()
    a = Cdouble[1.0]
    b = Cdouble[2.0]
    c = Cdouble[0.0]
    ccall((:multiply_, "/full/path/to/fortran_multiply"),
       Void,(Ptr{Float64},Ptr{Float64},Ptr{Float64}),
             a,b,c)
    println(c[1])
end
```

More information can be found here: `https://docs.julialang.org/en/stable/manual/calling-c-and-fortran-code/`

Support for other languages

Python	https://github.com/JuliaPy/PyCall.jl
R	https://github.com/JuliaInterop/RCall.jl
C++	https://github.com/Keno/Cxx.jl
Matlab	I have heard rumours of such a thing existing, but the horror ...

Conclusion

Start writing Julia code now without being worried about losing your prior work!

What is next?

Question?!

What do you want to hear learn about?

Next Session How does the compiler work and how do we get performance.

Next Tuesday Data Structures and Algorithms

Last Session Parallel computing, threading, GPUs? Up to grabs.