Julia for NHEP

Philippe Gras

CEA/IRFU - Saclay

Feb 8, 22

Introduction

- Julia is a recent programming language (first release in 2013)
  - Designed to provide high performance (like C/C++) and easy programming (like Python) within the same language
  - Rich ecosystem, especially for scientific domain
- The ideal language for NHEP applications, with a growing interest.
  - Attractive as a replacement of C++ $\oplus$ Python paradigm

Julia solving the two-language problem

Fast/easy coding		Fast running
Python	⇔	C/C++

⇒ Effect: mix of languages and going back-and-forth between them

- J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah tackled the problem in 2009 aiming to design a programming language that provides both Fast/easy coding **AND** Fast running
  - Birth of Julia, release 0.1 in **2013**
  - This breakthrough was recognised by awards attributed to the authors
    - James H. Wilkinson Prize in Numerical Analysis and Scientific in 2019 IEEE Computer Society Sidney Fernbach Award in 2019
    - IEEE Computer Society Sidney Fernbach Award in 2019
- In 12 years since its conceptualisation, Julia has been improved from release to release and has aggregated many package developers

Nowadays, Julia is a mature language, with a wide ecosystem

Examples of Julia uses

- The climate modeling alliance Clima code is written in Julia: https://clima.caltech.edu/
- Celeste: a new parallel computing method to process the Sloan Digital Sky Survey (SDSS) data set and produce the most accurate catalogue of 188 million astronomical objects in just 14.6 min.:
- Pharmaceutical (Pfizer and Moderna partly use Julia): https://juliacomputing.com/case-studies/pfizer/
- Energy network research at Los Alamos: https://lanl-ansi.github.io/
- Federal Reserve bank of New York: https://libertystreeteconomics.newyorkfed.org/2015/12/the-frbny-dsge-model-meets-julia/
- Electronics simulation and quantum computing: https://www.hpcwire.com/off-the-wire/julia-computing-receives-darpa-award-to-accelerate\-electronics-simulation-by-1000x/

Julia in NHEP

- KM3Net high-level software has a Julia environment in development in addition to the Python one (reported here)
- The LEGEND $0\nu\beta\beta$ experiment uses two parallel stacks, the primary in Python and the secondary (for validation and experimentation) in Julia. C++ is used for Geant-4 simulation software (reported here)
- LHCb analysis that leads to the first observation of the $\Omega_b^- \to \Xi_c^+ K^- \pi^-$ decay (10.1103/PhysRevD.104.L091102) uses Julia: see Julia for data analysis in High Energy Physics, Mikael Mikhasenko. M. Mikkhasenko has used Julia also for a JPAC analysis (doi:10.1103/PhysRevD.98.096021) and a COMPAS analysis (doi:10.1103/PhysRevLett.127.082501) as reported here.
- Performance of Julia for High Energy Physics Analyses, Marcel Stanitzki and Jan Strube
- Julia HEP organization on github
- Julia-in-HEP session of PyHEP 2021 workshop, HSF Julia for HEP Mini-worksop

Use of Julia for NHEP still limited, the interest is growing.

An incursions into Loops

Photo by Roberto Bormann from FreeImages

HEP data analysis is a looping game

HEP enjoys loop: we loop on physics events to loop on particles/physics objects. We often perform particle matching and clustering and for this we loop on events to loop on objects to loop on objects.

```
for event in billions_of_lhc_events
    for tens_or_hundreds_of_objects in event
        for tens_or_hundres_of_objects_to_match in event
            ...
        end
    end
end
```

- This is repeated several times for each analysis.

$\Rightarrow$ For an LHC analysis, lines of code executed billions of times even for a Kleenex code, written specially for a publication.

A simple loop in C/C++

```cpp
#include <iostream>
#include <sys/time.h>x

int main(){
  struct timeval t0, t1;
  gettimeofday(&t0, 0);

  double a = 0.;
  for(unsigned i = 1; i <= 1000000; ++i) a += 1.0/i;
  std::cout << "Computation Result: " << a << "\n";

  gettimeofday(&t1, 0);
  std::cerr << "Duration: " << (t1.tv_sec-t0.tv_sec)
    + 1.e-6*(t1.tv_usec-t0.tv_usec)
        << " seconds\n";
  return 0;
}
```

```
run(`g++ -Wall -o simple-loop simple-loop.cc`)
run(`./simple-loop`)
;
run(`g++ -O3 -Wall -o simple-loop simple-loop.cc`)
run(`./simple-loop`)
;
```

```
Computation Result: 14.3927
Computation Result: 14.3927
```

```
Duration: 0.002332 seconds
Duration: 0.001021 seconds
```

$$\frac{\textbf{C/C++}}{1.0\text{ms}}$$

A simple loop in Python

```
def f():
    a = 0.
    for i in range(1, 1_000_000 +1):
        a = a + 1.0/i
    return a
```

```
%%time
print(f())
```

```
14.392726722864989
CPU times: user 44.2 ms, sys: 0 ns, total: 44.2 ms
Wall time: 43.6 ms
```

C/C++	Python
1.0ms	44ms

- Coding is simpler
- No need to compile
- The code runs **44 times slower than C/C++**.

Python dislikes loops

- A master rule for high-performance code in Python is to avoid writing loop in Python

$\Rightarrow$ push the loop to underlying compiled libraries. Approach of the numpy vectorization.

A simple loop in Julia

```julia
#
# Julia
#
function f()
    a = 0.0
    for i in 1:1_000_000 # ✨ Note the underscores that improves legibility
       a = a + 1.0/i
    end
    return a
end
f()
@time b = f()
```

```
  0.001004 seconds (1 allocation: 16 bytes)

14.392726722864989
```

C/C++	Python	Julia
1.0ms	44ms	1.0ms

- As simple as Python, as fast as C/C++

What makes Julia unique

Developed from Day-1 with the goal of conciliating high performance computing with easy coding

Just-in-time compilation

- Provides both fast execution and a good interactive experience

Its type system

Its multiple dispatch paradigm

Support for **Ju**pyter notebook

- (**Ju** stands for Julia).

The Julia type system

- Dynamic
- The JIT compiler infers the variable types when possible to produce optimised code
- Possibility to explicitly indicate a type
  - To provide polymorphism (annotation of argument types)
  - For efficiency, but type can most often be inferred by the compiler
  - For explicit type checking.
- Parametric types, like C++ template

```
struct P4{T}
px::T
py::T
pz::T
E::T
end
```

- Inheritance from abstract types. The abtract types allow writing generic functions, later called with concrete types.
- Julia provides polymorphism in both generic programming and function overriding meanings in a more consistent manner than C++

The multiple dispatch paradigm

Dispatch = dynamic polymorphism

- The executed code when calling a function depends on the type of its argument. Selection done at runtime.

Single dispatch: **dynamic** polymorphism for a single parameter

- The case of C++ with the virtual class member functions

Multiple dispatch: **dynamic** polymorphism for every parameter of a function

- A central feature of Julia

The Multiple dispatch eases remarkably use/extension of third-party libraries

- It explains the rapid grow of the Julia ecosystem.
- See why in S. Karpinski's The Unreasonable Effectiveness of Multiple Dispatch talk.

Programming with Julia is easy

- Code syntax and grammar is similar to Pythons. No *std::map<std:string, std::vector<MyClass»*..., no compilation step.
- Dynamic type system
- Easy to learn
- Syntactic sugars similar to Python for a concise code: list comprehension, a < b < c, 1_000_000, support of symbols for variables...

and more: e.g. a function call is "vectorized" (ala numpy) with a simple dot, f.(x)

- Interactive help, nice tools to debug, to optimise code, for introspection.

Programming in a community

- Internet search engine and stack overflow play is an essential ingredient in nowadays programming workflow.
- Julia is already widespread enough, to find all the information on the Internet.
- In addition to usual resource, Julia has dedicated fora on Discourse, Slack, and Zulip with an active and friendly community.

Go to `https://www.duckduckgo.com` or your preferred search engine and make a try.

A rich ecosystem

- Large set of libraries and active development
  - Julia is firstly used by scientific community $\Rightarrow$ oriented to our needs
- Machine Learning, GPU, Plotting, DataFrames, etc...
- I did the following exercise during the PyHEP2021 workshop: I've looked for a Julia equivalent each time a speaker mention a Python library (apart from HEP specific ones).
  - Found a Julia equivalent of 16 out of the 18 mentioned libraries: missing one was a binding to FreeCAD (which is in discussion) and the software testing library with a specify technique (Hypothesis).

Data format support

Non-HEP format

- HDF5 and Parquet are fully supported (also CSV and Excel, less relevant for NHEP)

ROOT

- Two packages, developed by users.
  - Writtten in Julia, fast, and read-only: UnROOT.jl from Tamas Gal and Jerry Ling. Can read KM3Net data and tree of simple type and/or vector of simple type like CMS NanoAOD.
  - Providing both read and write support: UpROOT.jl from Oliver Schulz. A wrapper to uproot. Support xroot.

Tools

IDE

- Emacs and vim support
- Atom and VScode support. Many features. Code can be run and debugged with the IDE, with support for plots.

Notebooks

- **Ju**pyter
- Pluto. A new generation notebook with automatic update of cells.

Debugger

- Debugger, Rebugger, Juno debugger (for Atom IDE)

Code optimisation

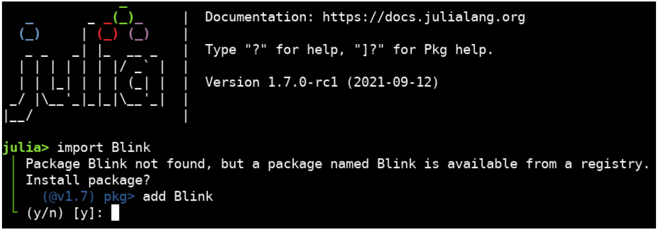
- Integrates nice and easy-to-use tools to optimize code performance.

Package installation

Package installation

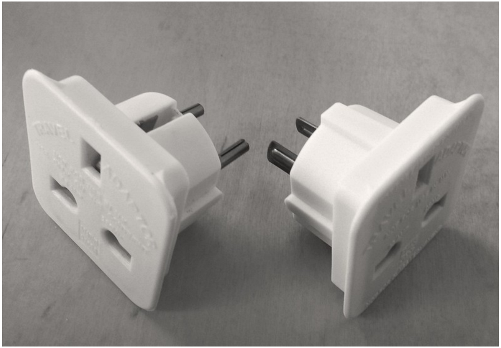
Python made it easy with Conda and pip. It's even easier in Julia

- A standard library part of the Julia installation
- Give instructions to the user, when he or she tries to import a missing package.



Language Interporability

In NHEP, we have a large legacy of software
$\Rightarrow$ Reuse of libraries written in different languages is essential



Language Interoperability provided by Julia

Use of library written in a different language

- Python, C, Fortran code: direct call from Julia and Jupyter Julia kernels
- C++ code: call via a wrapper. Lacking a tool for automatic generation of wrapper like swig. Project for direct-call (ala cppyy) on hold and not working for recent versions of Julia.

The other way around

- Python code can call Julia as well
- C/C++ code can call Julia code

Calling Python from Julia

```
# Enable Python call:
using PyCall

# Inport a python module:
math = pyimport("math")

# Use it as a Julia module:
math.sin(math.pi / 4)
```

```
0.7071067811865475
```

Calling Julia from Python

```
$ python3 -m pip install julia    # install PyJulia
...                               # you may need `--user` after `install`

$ python3
>>> import julia
>>> julia.install()               # install PyCall.jl etc.
>>> from julia import Base        # short demo
>>> Base.sind(90)
1.0
```

Mixing Julia and Python code in a notebook

Julia code cells can be included in a Jupyter notebook running a Python kernel:

Load the Julia magic extension

```
%load_ext julia.magic
```

```
The julia.magic extension is already loaded. To reload it, use:
  %reload_ext julia.magic
```

Excute some code written in "Julia"

```
%%julia
x = 10
y = x^2
println("x = $(x) ⇒ x² = $(y)")
```

```
x = 10 ⇒ x² = 100
```

Variables defined in Julia can be accessed from Python

```
x = %julia x
y = %julia y
print(f'x = {x} ⇒ x² = {y}')
```

```
x = 10 ⇒ x² = 100
```

Calling C or Fortran from Julia

```
path = ccall(:getenv, Cstring, (Cstring,), "SHELL")
unsafe_string(path)
```

```
"/bin/bash"
```

You will typically write a wrapper in Julia to handle errors, like this:

```
function getenv(var::AbstractString)
    val = ccall(:getenv, Cstring, (Cstring,), var)
    if val == C_NULL
        error("getenv: undefined variable: ", var)
    end
    return unsafe_string(val)
end
```

```
getenv (generic function with 1 method)
```

```
println(getenv("USER"))
println(getenv("SMOKE")) # ⇒ will through an exception unless you have SMOKE in your environment
```

```
pgras

getenv: undefined variable: SMOKE

Stacktrace:
 [1] error(::String, ::String)
   @ Base ./error.jl:42
```

Julia from C/C++

```c
#include <julia.h>
JULIA_DEFINE_FAST_TLS

int main(int argc, char *argv[])
{
    /* required: setup the Julia context */
    jl_init();

    /* run Julia commands */
    jl_eval_string("print(sqrt(2.0))");

    /* notify Julia that the program is about
       to terminate. */
    jl_atexit_hook(0);
    return 0;
}
```

A proof-of-concept of integrating Julia in a C++ HEP Framework:
https://github.com/grasph/JuliaInACxxHepFramework

A simple HEP example

- Dimuon spectrum using data from the CMS detector.
- Analysis and data available from the CERN opendata portal: here
- Uses the UnROOT package to read the open CMS data from its ROOT format.
- Data format:

Column name	Data type	Description
`nMuon`	`unsigned int`	Number of muons in this event
`Muon_pt`	`float[nMuon]`	Transverse momentum of the muons (stored as an array of size `nMuon`)
`Muon_eta`	`float[nMuon]`	Pseudorapidity of the muons
`Muon_phi`	`float[nMuon]`	Azimuth of the muons
`Muon_mass`	`float[nMuon]`	Mass of the muons
`Muon_charge`	`int[nMuon]`	Charge of the muons (either 1 or -1)

A simple HEP example

The analysis code

```
function analyze_tree(t, maxevents = -1)
    bins = 30_000 # Number of bins in the histogram
    low = 0.25 # Lower edge of the histogram
    up = 300.0 # Upper edge of the histogram
    h = H1{Float64}(Axis(bins, low, up))
    for (ievt, evt) in enumerate(t)
        maxevents >= 0 && ievt > maxevents && break
        evt.nMuon ==2 || continue
        evt.Muon_charge[1] != evt.Muon_charge[2] || continue
        dimuon_mass = m(ptetaphim(evt.Muon_pt[1], evt.Muon_eta[1], evt.Muon_phi[1], evt.Muon_mass[1])
                        + ptetaphim(evt.Muon_pt[2], evt.Muon_eta[2], evt.Muon_phi[2], evt.Muon_mass[2]))
        hfill!(h, dimuon_mass)
    end
    h
end;
```

Running the analysis

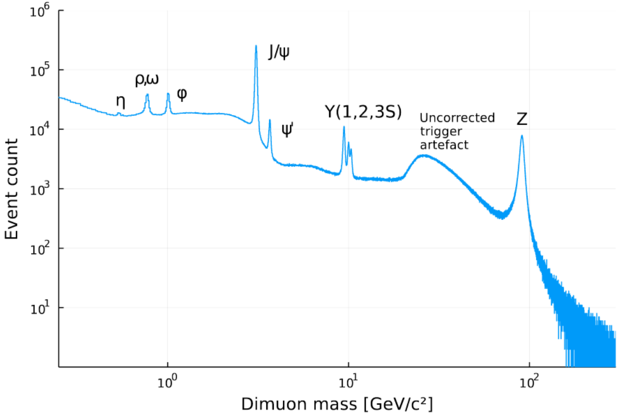
```
t = LazyTree(ROOTFile(fname),"Events")
@time h = analyze_tree(t);
```

```
 32.669432 seconds (165.32 M allocations: 22.806 GiB, 12.62% gc time, 2.41% compilation time)
```

- 33 secs to run over 61.5M events

A simple HEP example

- The results plotted using tools from the Julia ecosystem



A simple HEP example: code speed

- Time to execute the code was compared to implementations performed in Python

Julia	Python event loop	Python RDataFrame JIT-compiled C++	Python RDataFrame JIT-compiled python (Numba)
35 s	4h 5min	60 s	125s

Similar performance expected for a DataFrame-based Julia implementation

$\Rightarrow$ Julia runs fast out of the box

No need to think of performance when writing the code

Conclusions

- Julia is not just another language. It meets a need,

  Reconciling fast running with easy and fast coding

- It comes with a large Ecosystem.
- Mature enough to be used for NHEP.

> Julia is a very promising language for NHEP.