Towards new solutions for scientific computing: the case of Julia

Maurizio Tomasi  Mosè Giordano

2018/11/15
Who are we?

Maurizio Tomasi (myself)
- Worked on the Planck mission (calibration, simulations, data analysis...)
- Currently involved in other CMB experiments

Mosè Giordano
- Worked on gravitational microlensing
- Author of several Julia packages (github.com/giordano)
Python is a fantastic language: easy and with a very rich library. (And AstroPy is awesome!)

However, its speed is not impressive at all!

In [1]: %time x = [i*i for i in range(100_000_000)]
CPU times: user 5.27 s, sys: 860 ms, total: 6.13 s
Wall time: 6.18 s

Several solutions have been developed: NumPy, PyPy, Numba, Cython... They can be extremely performant in their own domains, but picking the right one requires careful consideration.
The two-language problem

In order to make Python codes more performant, it is common to link them to C/C++/Fortran, using tools like f2py, SWIG, Cython, and so on:

These codes are complex to implement and deploy:

- Need to master many languages
- Try to write a portable setup.py for projects using f2py!
Meet Julia

- Relatively new language (first official release was 0.2, in Nov 2013)
- Julia 1.0 released on August, 9th 2018
- Released under the MIT license
- julialang.org
Towards new solutions for scientific computing: the case of Julia (Maurizio Tomasi, Mosè Giordano)

Julia> f(x) = 2x + 2
   ^ (generic function with 1 method)
Julia> x = 6
Julia> x ∈ [1, 3, 5, 8, 9]
False
Julia>
# One-liner definition of a function

\[
f(x) = 3x + 1
\]

# Floating-point

@time f(0.1)  # 0.005716 seconds (15.63 k allocations: 872.499 KiB)
@time f(0.3)  # 0.000002 seconds (5 allocations: 176 bytes)

# Integer

@time f(2)    # 0.002888 seconds (2.00 k allocations: 117.656 KiB)
@time f(10)   # 0.000001 seconds (4 allocations: 160 bytes)

# Rational

@time f(3//2) # 0.070596 seconds (209.83 k allocations: 10.785 MiB)
@time f(4//9) # 0.000005 seconds (6 allocations: 224 bytes)
# Load default packages
using Printf
using Pkg

# Install a few new packages from Internet
for name in ["Cosmology", "Measurements", "Zygote#master", "PyCall"]
    Pkg.add(name)
end

using Cosmology
c = cosmology(h=0.69, Neff=3.04, OmegaM=0.29, Tcmb=2.7255)
z = 0.1

@printf("Universe age at z=%.1f: %.1f Gyr\n", z, age_gyr(c, z))
# Prints "Universe age at z=0.1: 12.5 Gyr"
using Measurements  # Define the ± binary operator

z = 0.1 ± 0.01
println(z)
# Prints "0.1 ± 0.01"

age = age_gyr(c, z)
println(age)
# Prints "12.465336269441773 ± 0.12305608850870296"

@printf("%.2f ± %.2f Gyr\n", age.val, age.err)
# prints "12.47 ± 0.12 Gyr"
# See https://arxiv.org/abs/1810.07951
using Zygote # Long time to compile...

\[ g(x) = 2x + 1 \]

println(g(1)) # Print 3
println(g'(1)) # Print 2 (derivative of g at x=1)
@code_llvm g'(1) # Surprise! "ret i64 2"
using PyCall

@pyimport numpy.random as nr
x = nr.randn(5)
<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AstroImages.jl</td>
<td>Visualization of astronomical images (by MG)</td>
</tr>
<tr>
<td>AstroLib.jl</td>
<td>Astronomical and astrophysical routines (by MG)</td>
</tr>
<tr>
<td>AstroTime.jl</td>
<td>Astronomical time keeping</td>
</tr>
<tr>
<td>Cosmology.jl</td>
<td>Library of cosmological functions</td>
</tr>
<tr>
<td>DustExtinction.jl</td>
<td>Models for the interstellar extinction due to dust</td>
</tr>
<tr>
<td>ERFA.jl</td>
<td>Wrapper to liberfa</td>
</tr>
<tr>
<td>EarthOrientation.jl</td>
<td>Earth orientation parameters from IERS tables</td>
</tr>
<tr>
<td>FITSIO.jl</td>
<td>Flexible Image Transport System (FITS) file support</td>
</tr>
<tr>
<td>LombScargle.jl</td>
<td>Compute Lomb-Scargle periodogram (by MG)</td>
</tr>
<tr>
<td>SPICE.jl</td>
<td>Julia wrapper for NASA NAIF's SPICE toolkit</td>
</tr>
<tr>
<td>SkyCoords.jl</td>
<td>Support for astronomical coordinate systems</td>
</tr>
<tr>
<td>UnitfulAstro.jl</td>
<td>An extension of Unitful.jl for astronomers</td>
</tr>
<tr>
<td>WCS.jl</td>
<td>Astronomical World Coordinate Systems library</td>
</tr>
</tbody>
</table>
Simulating cosmological experiments with Julia
Data science with Julia

**The good**

- Very fast execution for codes with lots of calculations
- Powerful features (many numerical types, metaprogramming, missing values...)
- Ability to call C, Fortran, Python, R
- Package management is rock solid (reproducible builds, like Rust’s cargo)
- Native support for parallel computing (no GIL here!)
- Profiling tools immediately available (e.g., --track-allocation)

**The bad**

- Slow execution if every function is called just once
- Not as many packages as other languages (Python, C++, ...)
- Plotting is promising (PyPlot.jl, Plots.jl, UnicodePlots.jl, Makie.jl, ...), but still lacking
- Avoid global variables as the plague! They make the compiler highly inefficient
When to use Julia

Julia is interesting if:

- You are going to implement a code that will do lots of calculations and is going to spend much time in doing it, and you will write this code *from scratch*. 
When to use Julia

Julia is interesting if:

▶ You are going to implement a code that will do lots of calculations and is going to spend much time in doing it, and you will write this code from scratch.

▶ You have an existing large, monolithic code and want to turn it into something to be used interactively, without sacrificing speed.
When to use Julia

Julia is interesting if:

▶ You are going to implement a code that will do lots of calculations and is going to spend much time in doing it, and you will write this code *from scratch*.

▶ You have an existing large, monolithic code and want to turn it into something to be used interactively, without sacrificing speed.

▶ You plan to use Julia’s homoiconicity to do something really innovative, like Zygote.jl!
More information

- Julia compiler: julialang.org
- Julia user’s manual: docs.julialang.org/en/v1
- Package list available at juliaobserver.com
- User’s and developers’ forums: discourse.julialang.org
- Very good blogpost about Numba, Cython, and Julia:
  www.stochasticlifestyle.com/why-numba-and-cython-are-not-substitutes-for-julia
- JuliaAstro: github.com/JuliaAstro

- These slides and additional material: bitbucket.org/Maurizio_Tomasi/adass2018-julia
- For questions, feel free to ask me or write me an email: maurizio.tomasi@unimi.it
Backup slides
Calculations with NumPy arrays

Consider this code, where all the parameters for f are NumPy arrays:

```python
def f(r, x1, x2, x3, x4):
    r = x1 - x2 + x3 - x4
```

This code is executed by NumPy as if it were

```python
tmp = x1 - x2
tmp += x3
r = tmp - x4
```

thus three for loops are ran.
Performance of NumPy codes

![Graph showing the performance of NumPy codes with the number of terms in the expression and the best time for 100 runs in milliseconds. The graph displays a clear linear relationship.]

Source codes available at github.com/ziotom78/python-julia-c-.

In C++, the code needs to be written in this way:

```cpp
for(size_t i = 0; i < r.size(); ++i) {
    r[i] = x1[i] - x2[i] + x3[i] - x4[i];
}
```

We need to write the `for` loop explicitly, but there is only **one** of them.
Performance of NumPy/C++ codes


# 4 terms

\[ f(r, x_1, x_2, x_3, x_4) = r = x_1 - x_2 + x_3 - x_4 \]

\[ g(r, x_1, x_2, x_3, x_4) = r = x_1 - x_2 + x_3 \]

\[ h(r, x_1, x_2, x_3, x_4) = r = x_1 - x_2 \]

# Etc.

The `@.` macro fuses all the operations on loops. Thus, \( f \) above is equivalent to

```julia
function f(r, x1, x2, x3, x4)
    for i in eachindex(r)
        r[i] = x1[i] - x2[i] + x3[i] - x4[i]
    end
end
```

Performance of NumPy, C++, and Julia codes


C++ was given an unfair advantage, as it was allowed to use SIMD instructions (-msse3). Moreover, it did not check array boundaries (Julia does automatically).

In Julia, we can use the @inbounds and @simd macro to make Julia code equivalent to C++:

```julia
function f(r, x1, x2, x3, x4)
    @inbounds @simd for i in eachindex(r)
        r[i] = x1[i] - x2[i] + x3[i] - x4[i]
    end
end
```
Performance of NumPy, C++, and Julia codes

Source codes available at github.com/ziotom78/python-julia-c-.

Performance of NumPy, C++, and Julia codes

Source codes available at github.com/ziotom78/python-julia-c-