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)

Making Python more performant

Python is a fantastic language: easy and with a very rich library. (And AstroPy is awesome!)

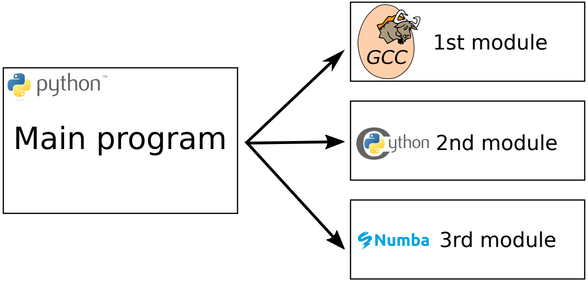
However, its speed is not impressive at all!

```
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:

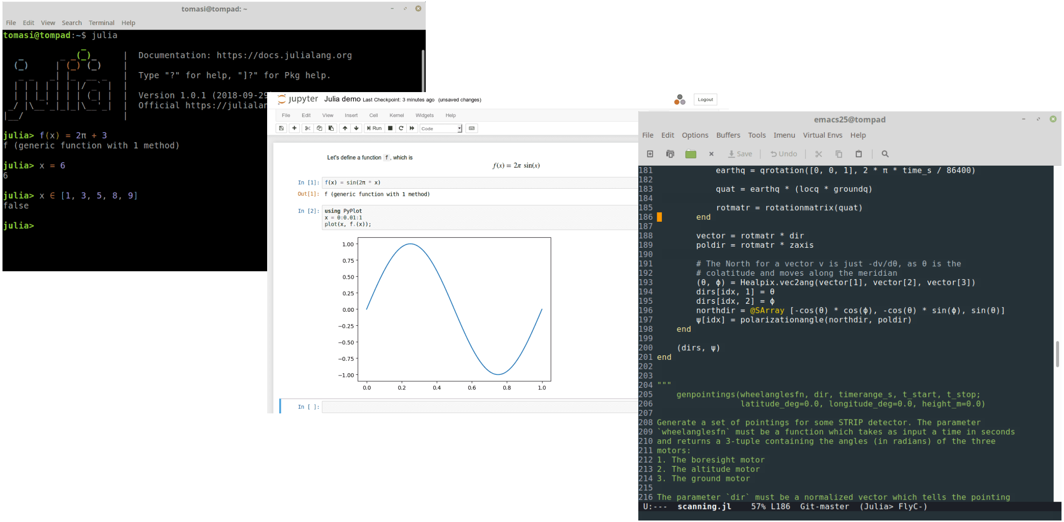


These codes are complex to implement and deploy:

- 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


tomasi@tompad: ~
File Edit View Search Terminal Help
tomasi@tompad:~$ julia
Documentation: https://docs.julialang.org
Type "?" for help, "]?" for Pkg help.
Version 1.0.1 (2018-09-29
Official https://julialan
julia> f(x) = 2π + 3
f (generic function with 1 method)
julia> x = 6
6
julia> x ∈ [1, 3, 5, 8, 9]
false
julia>
jupyter Julia demo Last Checkpoint: 3 minutes ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Run
Code
Let's define a function f, which is
f(x) = 2π sin(x)
In [1]: f(x) = sin(2π * x)
Out[1]: f (generic function with 1 method)
In [2]: using PyPlot
x = 0:0.01:1
plot(x, f.(x));
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.2
0.4
0.6
0.8
1.0
In [ ]:
emacs25@tompad
File Edit Options Buffers Tools Imenu Virtual Envs Help
Save
Undo
181 earthq = qrotation([0, 0, 1], 2 * π * time_s / 86400)
182
183 quat = earthq * (locq * groundq)
184
185 rotmatr = rotationmatrix(quat)
186 end
187
188 vector = rotmatr * dir
189 poldir = rotmatr * zaxis
190
191 # The North for a vector v is just -dv/dθ, as θ is the
192 # colatitude and moves along the meridian
193 (θ, ϕ) = Healpix.vec2ang(vector[1], vector[2], vector[3])
194 dirs[idx, 1] = θ
195 dirs[idx, 2] = ϕ
196 northdir = @SArray [-cos(θ) * cos(ϕ), -cos(θ) * sin(ϕ), sin(θ)]
197 ψ[idx] = polarizationangle(northdir, poldir)
198 end
199
200 (dirs, ψ)
201 end
202
203
204 """
205 genpointings(wheelanglesfn, dir, timerange_s, t_start, t_stop;
206 latitude_deg=0.0, longitude_deg=0.0, height_m=0.0)
207
208 Generate a set of pointings for some STRIP detector. The parameter
209 `wheelanglesfn` must be a function which takes as input a time in seconds
210 and returns a 3-tuple containing the angles (in radians) of the three
211 motors:
212 1. The boresight motor
213 2. The altitude motor
214 3. The ground motor
215
216 The parameter `dir` must be a normalized vector which tells the pointing
U:--- scanning.jl 57% L186 Git-master (Julia> FlyC-)

A taste of Julia (1/5)

```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)
```

A taste of Julia (2/5)

```julia
# 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"
```

A taste of Julia (3/5)

```
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"
```

A taste of Julia (4/5)

```
# 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"
```

A taste of Julia (5/5)

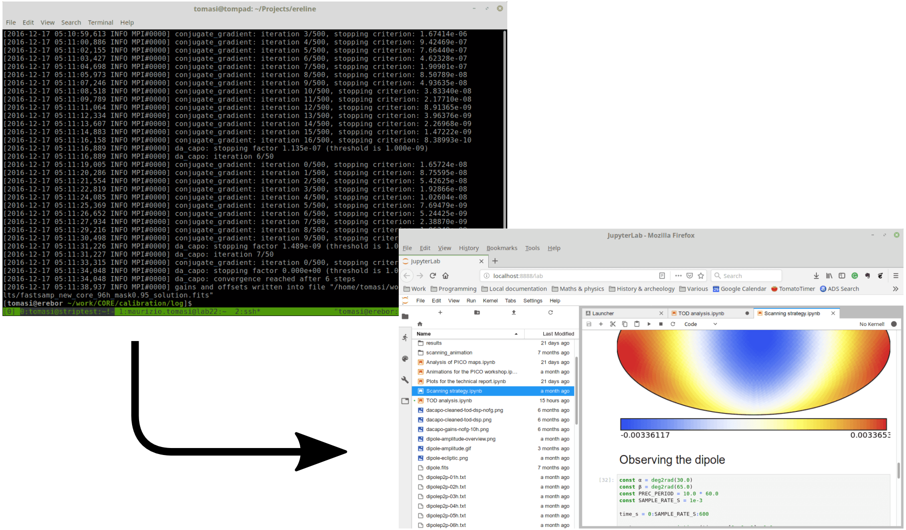
```julia
using PyCall

@pyimport numpy.random as nr
x = nr.randn(5)
```

JuliaAstro on GitHub

Package	Description
`AstroImages.jl`	Visualization of astronomical images (by MG)
`AstroLib.jl`	Astronomical and astrophysical routines (by MG)
`AstroTime.jl`	Astronomical time keeping
`Cosmology.jl`	Library of cosmological functions
`DustExtinction.jl`	Models for the interstellar extinction due to dust
`ERFA.jl`	Wrapper to `liberfa`
`EarthOrientation.jl`	Earth orientation parameters from IERS tables
`FITSIO.jl`	Flexible Image Transport System (FITS) file support
`LombScargle.jl`	Compute Lomb-Scargle periodogram (by MG)
`SPICE.jl`	Julia wrapper for NASA NAIF's SPICE toolkit
`SkyCoords.jl`	Support for astronomical coordinate systems
`UnitfulAstro.jl`	An extension of `Unitful.jl` for astronomers
`WCS.jl`	Astronomical World Coordinate Systems library

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:

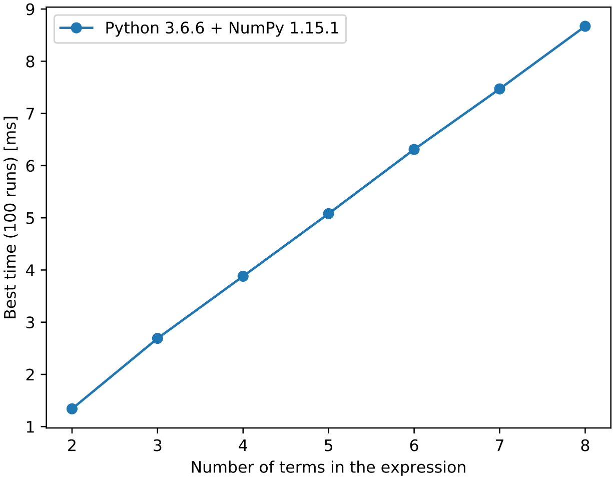
```
def f(r, x1, x2, x3, x4):
    r = x1 - x2 + x3 - x4
```

This code is executed by NumPy as if it were

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

thus three `for` loops are ran.

Performance of NumPy codes



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

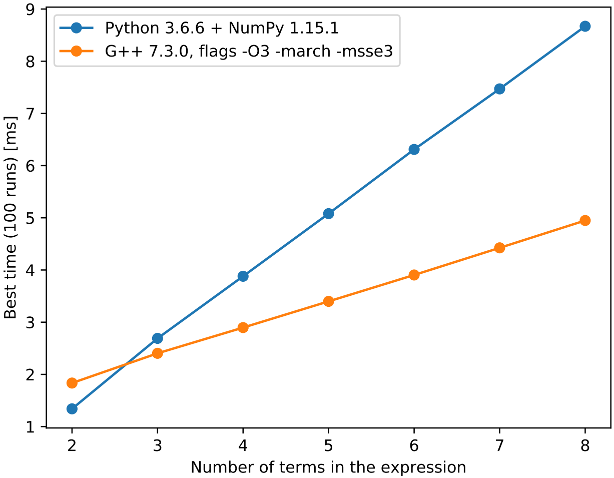
Calculations with C++ vectors

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

```
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



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

Performance of Julia programs

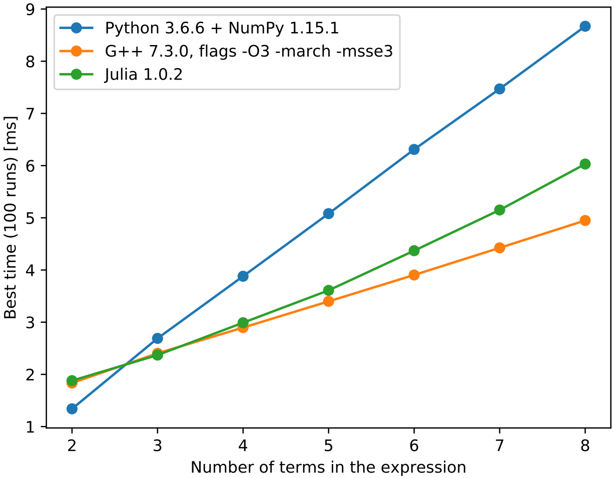
```
# 4 terms
f(r, x1, x2, x3, x4) = @. r = x1 - x2 + x3 - x4
g(r, x1, x2, x3, x4) = @. r = x1 - x2 + x3
h(r, x1, x2, x3, x4) = @. r = x1 - x2

# Etc.
```

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

```
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



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

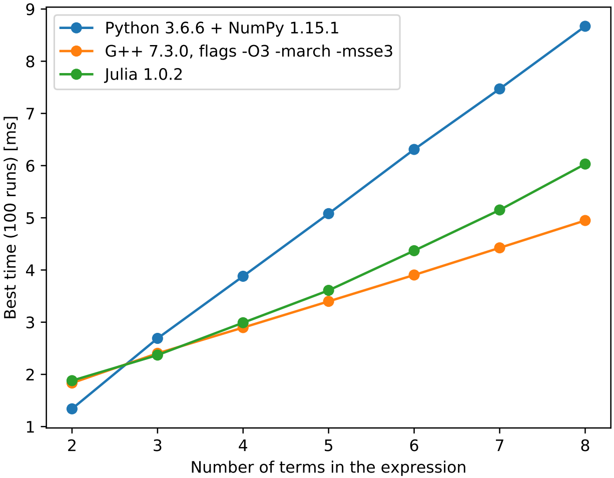
Using SIMD instructions in Julia

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++:

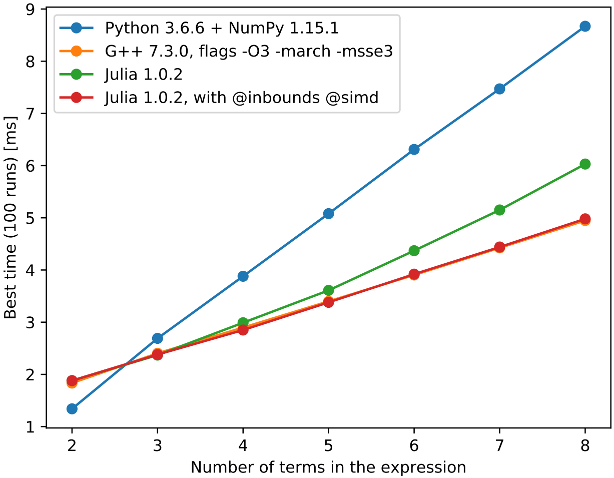
```
function f(r, x1, x2, x3, x4)
    @inbounds @simd for i in eachindex(r)
        r[i] = x1[i] - x2[i] + x3[i] - x4[i]
    end
end
```

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-.