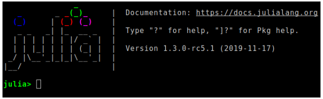
Julia: A fast & fun dynamic programming language



by Mauro Werder (github: @mauro3, werder@vaw.baug.ethz.ch)

Who am I?

- Oberassistent in the glaciology group of Daniel Farinotti @ WSL and ETHZ
  - besides numerical work I also do field and lab experiments

Who am I?

- Oberassistent in the glaciology group of Daniel Farinotti @ WSL and ETHZ
  - besides numerical work I also do field and lab experiments
- Julia user & contributor since 2013 –> probably one of the earliest adopters in Switzerland (Julia was first released in 2012)
  - maintainer of five Julia packages:
    - Parameters.jl – easier handling of large structs of parameters
    - UnPack.jl – packing and unpacking of data-structures
    - SimpleTraits.jl – a simple trait system
    - WhereTheWaterFlows.jl – a water-routing code
    - KissMCMC.jl – a Markov chain Monte Carlo sample
- besides my research I use Julia in teaching at ETHZ:
  - Physics of Glaciers
  - Solving PDEs in parallel on GPUs with Julia `https://eth-vaw-glaciology.github.io/course-101-0250-00/`

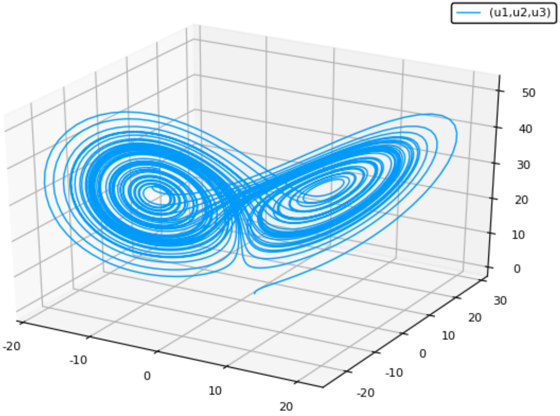
What does Julia code look like?

Example, solve Lorenz system of ODEs:

```julia
using OrdinaryDiffEq, Plots

function lorenz(x, p, t)
    σ = 10
    β = 8/3
    ρ = 28
    return [σ*(x[2]-x[1]), x[1]*(ρ-x[3]), x[1]*x[2] - β*x[3]]
end

# integrate dx/dt = lorenz(t,x) numerically from t=0 to t=50
# and IC x₀
tspan = (0.0, 50.0)
x₀ = [2.0, 0.0, 0.0]
sol = solve(ODEProblem(lorenz, x₀, tspan), Tsit5())

plot(sol, vars=(1,2,3)) # plot Lorenz attractor
```

What does Julia code look like?

```julia
plot(sol, vars=(1,2,3)) # plot Lorenz attractor
```



Julia in brief

Julia 1.0 was released in 2018, current version is 1.8

Features

- general purpose language with a focus on technical computing
- dynamic language
  - interactive development
  - garbage collection: no manual memory management
- good performance on par with C & Fortran (through just-ahead-of-time compilation)
  - No need to vectorize: for-loops are fast
- multiple dispatch
- user-defined types are as fast and compact as built-ins
- Lisp-like macros and other metaprogramming facilities
- designed for parallelism and distributed computation
- good inter-op with other languages

Ok, but why?

The two language problem

One language to prototype — one language for production

- example my institute: GREMS re-write from IDL to C(++)

One language for the users — one language for under-the-hood

- Numpy (python — C)
- machine-learning: pytorch, tensorflow

Ok, but why?

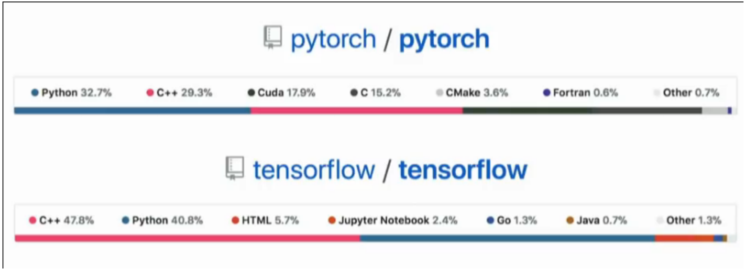
The two language problem

One language to prototype — one language for production

- example my institute: GREMS re-write from IDL to C(++)

One language for the users — one language for under-the-hood

- Numpy
- machine-learning: pytorch, tensorflow



Two language problem

Prototype/interface language:

- easy to learn and use
- interactive
- productive
- –> but slow
- Examples: Python, Matlab, R, IDL...

Production/fast language:

- fast
- –> but complicated/verbose/not-interactive/etc
- Examples: C, C++, Fortran, Java...

Julia solves the two-language problem

Julia is:

- easy to learn and use
- interactive
- productive

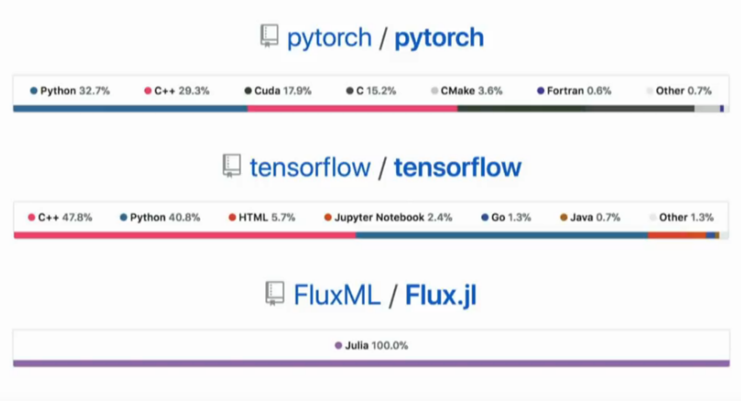
and also:

- fast

This blurs the line between users and developers!

Julia solves the two-language problem

Example:



Flux is a pure Julia deep learning package. This is possible, because Julia runs at native speed both on the CPU and GPU.

The good: easy to learn

- the basics of Julia are as simple to learn as Python, Matlab, etc
- however, the language has many powerful/advanced features which can be accessed as the programmer progresses in skills (rich type system, multiple dispatch, meta-programming, access to compiler) –> the language grows with the user
- writing code is very close to mathematics (not like the Python/Numpy stuff) –> one reason this works so well is *multiple dispatch* (next slide)

The good: nice syntax, cool features

Examples:

Compact function definition and unicode:

```
ϕ(t) = sin(2π * t)\\[1mm]
```

The good: nice syntax, cool features

Examples:

Compact function definition and unicode:

```
ϕ(t) = sin(2π * t)\\[1mm]
```

Vectorized function application:

```
ϕ.([1,2,3]) .+ [4,5,6]
```

(note, this gets fused into one loop, no temporary arrays are created)

The good: nice syntax, cool features

Examples:

Compact function definition and unicode:

```
ϕ(t) = sin(2π * t)\\[1mm]
```

Vectorized function application:

```
ϕ.([1,2,3]) .+ [4,5,6]
```

(note, this gets fused into one loop, no temporary arrays are created)

Parametric types

```
struct Point{N, T}
    coords::NTuple{N,T}
end
Point((3,4))   # 2D -> Point{2, Int64}((3, 4))
Point((3,4,5)) # 3D -> Point{3, Float64}((3.0, 4.0, 5.0))
```

Interlude: multiple dispatch

- Julia is **not** an object oriented language

Object oriented:

- methods belong to objects
- method is selected based on first argument (self)

Multiple dispatch:

- methods are separate from objects
- are selected based on all arguments
- very natural for mathematical programming

Juliacon 2019 presentation on the subject by Stefan Karpinski (co-creator of Julia):
"The Unreasonable Effectiveness of Multiple Dispatch" (link)

Demo-script

```julia
struct Rock end
struct Paper end
struct Scissors end
## of course structs could have fields as well
# struct Rock
#     color
#     name::String
#     density::Float64
# end

# define multi-method
play(::Rock, ::Paper) = "Paper wins"
play(::Rock, ::Scissors) = "Rock wins"
play(::Scissors, ::Paper) = "Scissors wins"
play(a, b) = play(b, a) # commutative

play(Scissors(), Rock()) # -> "Rock wins"
```

Demo-script (cont.)

```julia
# Extend-later: with new type
struct Pond end
play(::Rock, ::Pond) = "Pond wins"
play(::Paper, ::Pond) = "Paper wins"
play(::Scissors, ::Pond) = "Pond wins"

play(Scissors(), Pond()) # -> "Pond wins"

# Extend-later: with new function
combine(::Rock, ::Paper) = "Paperweight"
combine(::Paper, ::Scissors) = "Two pieces of papers"
# ...

combine(Rock(), Paper()) # -> "Paperweight"
```

The good: productive

- Julia is probably as productive or more so than your favourite language.
- Packages can be easily installed, into project environments if desired.
- Registered packages are listed on `https://juliahub.com/ui/Packages`
- The package ecosystem has not reached the level of the Python ecosystem but is growing fast. There are currently about 9000 registered packages.
  –> packages work well together, e.g. combine OrdinaryDiffEq.jl with Unitful.jl and Measurements.jl

The good: inter-op

- Good interoperability with other languages: `ccall` (C, Fortran), `Cxx.jl`, `PythonCall.jl`, `RCall.jl`, `MATLAB.jl`, etc.

```
ccall((:exp , "libm.so.6"), Cdouble , (Cdouble ,), 1.57) # ->4.806648193775178
```

The good: inter-op

- Good interoperability with other languages: `ccall` (C, Fortran), `Cxx.jl`, `PythonCall.jl`, `RCall.jl`, `MATLAB.jl`, etc.

```
ccall((:exp , "libm.so.6"), Cdouble , (Cdouble ,), 1.57) # ->4.806648193775178
```

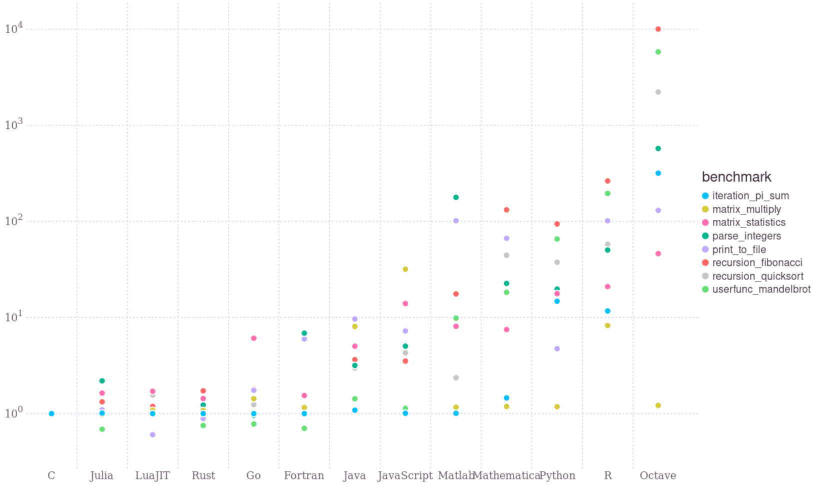
- runs on a number of platforms including CPU, GPU, TPU...

```
# on CPU
f(x) = 3x^2 + 5x + 2
C = Array([1f0 ,2f0 ,3f0]) # normal array
C .= f.(2 .* C.^2 .+ 6 .* C.^3 .- sqrt.(C)) # runs on the CPU, note `.`

# on GPU
using CUDA
B = CuArray([1f0 ,2f0 ,3f0]) # GPU array
B .= f.(2 .* B.^2 .+ 6 .* B.^3 .- sqrt.(B)) # runs on the GPU
```

(for CUDA-package to run, you need a Nvidia graphics card)

The good: fast

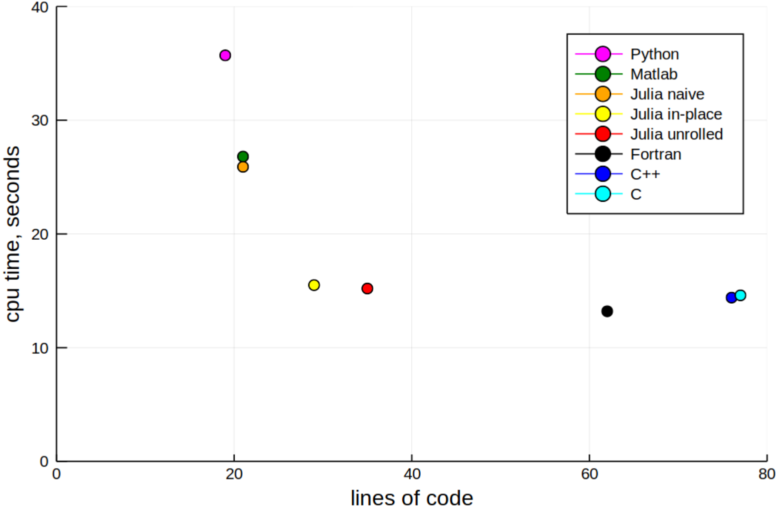
Julia is as fast as C and Fortan:



Micro-benchmarks from `https://julialang.org/benchmarks/`

The good: fast

Julia is as fast as C and Fortan, but more productive:

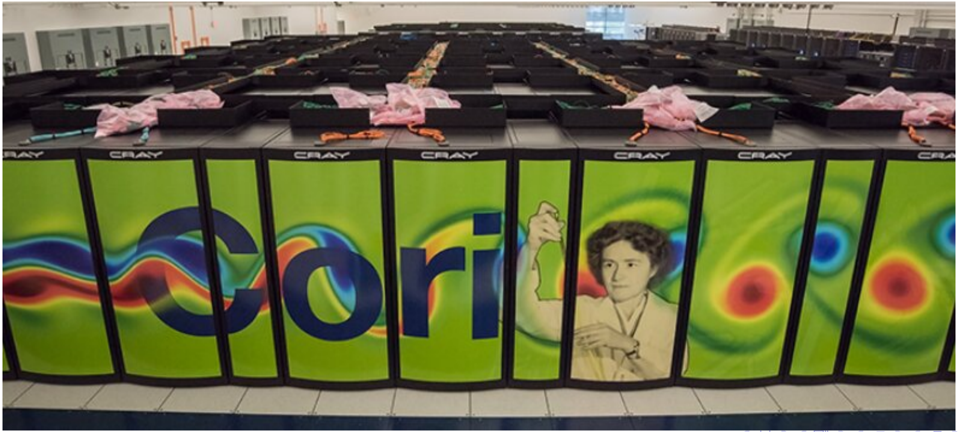


(c) @johnfgibson solving the Kuramoto-Sivashinksy PDE (time + 1D space)

Case-study: Celeste.jl

Project to produce an accurate catalogue of 188 million astronomical objects
Ran on Cori supercomputer (@ Berkeley Lab):

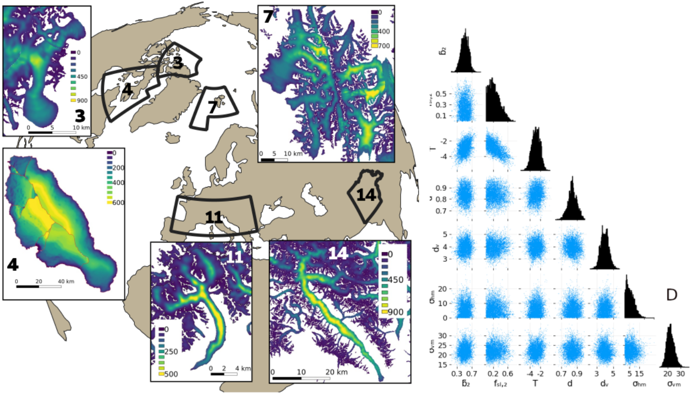
- 1.54 petaflops using 1.3 million threads on 9,300 Knight Landing (KNL) nodes
  –> the first dynamical language to join the petascale club!
- it took 15min to catalogue the objects using Bayesian techniques



Case-study: BITE-model

The Bayesian Ice Thickness Estimation (BITE) model is a project of mine:

- simple forward model to calculate ice thickness maps
- using Bayesian techniques (MCMC) to fit it to observations
- fitted to 30'000 glaciers, calculated 10^8 ice-thickness maps –> Julia's performance needed

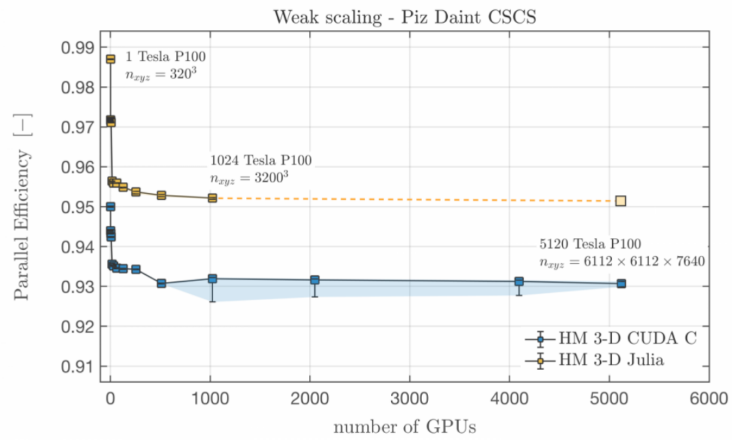


(Werder et al., 2020) `https://github.com/mauro3/BITEmodel.jl`

Case-study: Julia at CSCS

3D multi physics flow solver in Julia running on Piz Daint by Ludovic Räss (WSL/VAW Glaciology), Samuel Omlin (CSCS) & Yury Podladchikov (Uni Lausanne) (link)

- original code was a Matlab prototype + CUDA C + MPI production code
- Julia code running on 5120 NVIDIA Tesla P100 GPUs on the hybrid Cray XC-50 showing nearly perfect scaling



The good: cutting-edge

Whilst the breath of the package ecosystem is not comparable to, say, Python. There are many cutting-edge packages (see this afternoon).

The good: community

Last, the Julia community is very friendly and helpful:

- Easy to get help on the discourse forum, StackOverflow and on Slack
- Active developer community on GitHub
- JuliaCon is as fun and friendly (next one in July fully online and at MIT)

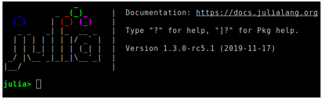
The bad

- There are no interface-based abstractions in the language
- It is not a statically compiled language and thus it does not have the associated safety features.
  –> there might be static checkers coming in the future

The ugly

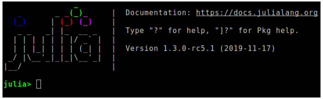
- Compilation times can be very long. E.g. time to first plot can 20s+ (but fast afterwards)
  - in general there are improvements from Julia version to version but it is still far from satisfying
  - apparently Julia 1.9, coming out in the next few weeks, should improve significantly on that again

Conclusions



- Julia sure is fast and fun
- solves the two language problem & blurs lines between devs and users

Conclusions



- Julia sure is fast and fun
- solves the two language problem & blurs lines between devs and users

Let's get started then!