Julia in HEP

Graeme A Stewart
Alexander Moreno Briceño, Allen Christopher Caldwell, Benedikt Hegner, Jerry Ling, Mikhail Mikhasenko, Oliver Schulz, Pere Mato, Philippe Gras, Sam Skipsey, Tamas Gal, Uwe Hernandez Acosta

EP-SFT

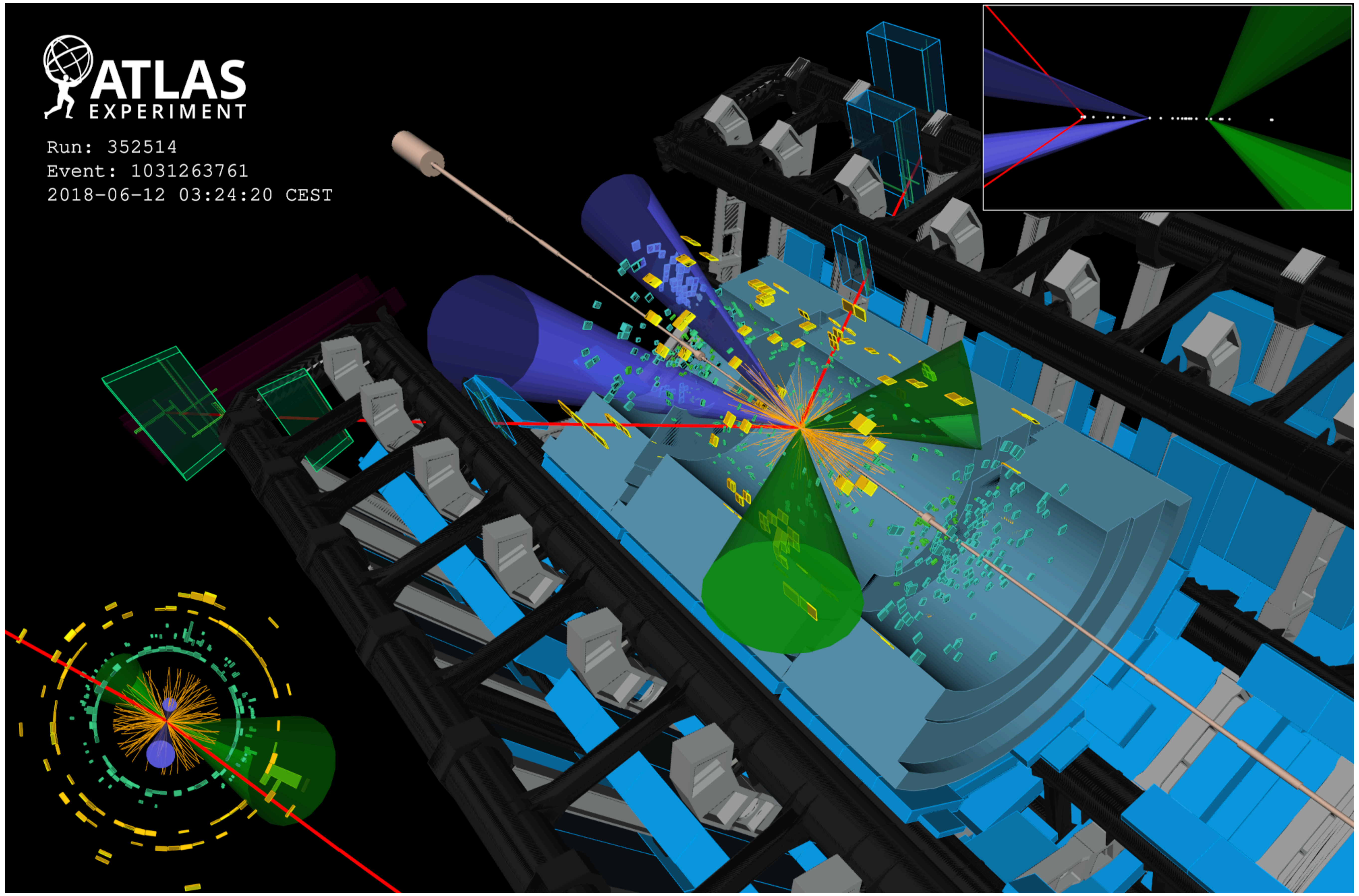
Computing in High Energy Physics, Krakow

2024-10-21

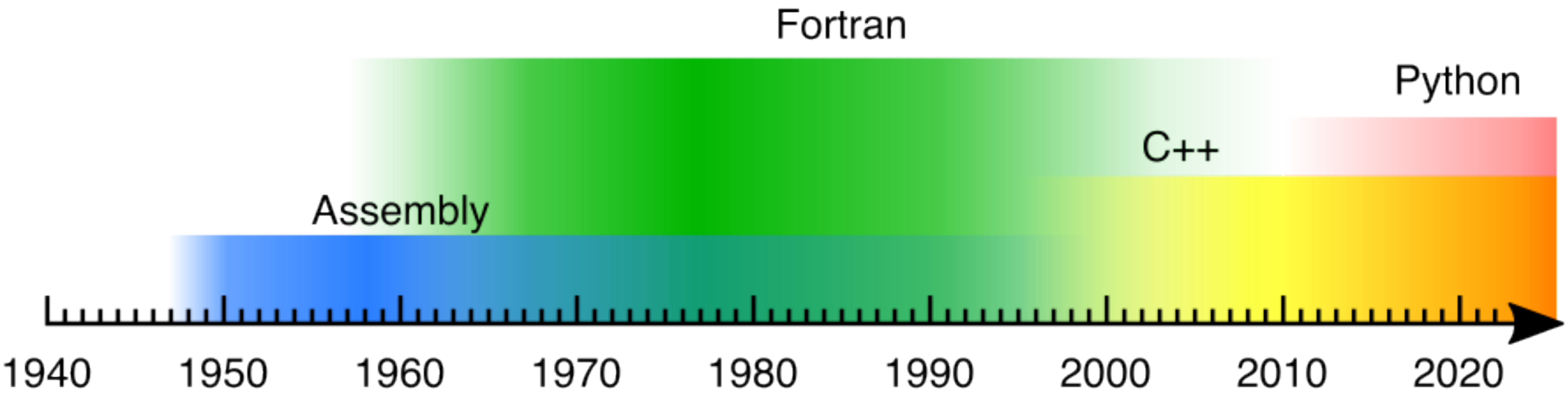
Programming Languages in HEP

What do we need from software?

- Code Efficiency
  - Fast execution
  - High throughput
  - Scalable
- Human Efficiency
  - Low barrier to entry
  - Rapid prototyping
  - Broad ecosystem
  - Excellent tooling



Programming Languages in HEP



[Thanks to Jim Pivarski, Princeton]

- Our languages do change over time, even if at any moment they might seem extremely fixed…

> *If I had to pick one thing likely to still be alive 30 years from now I would choose* **FORTRAN**. *It is as safe a bet as to predict that everything else is going to change.*

30 Years of Computing at CERN, Paolo Zanella, 1990

Where are we now?

There are always tradeoffs

Metric	C++	Python
Performance	✅	❌
Expressiveness	⚠️	✅
Learning Curve	❌	✅
Safety (memory)	⚠️	✅
Composability	❌	⚠️

Where are we now?

There are always tradeoffs

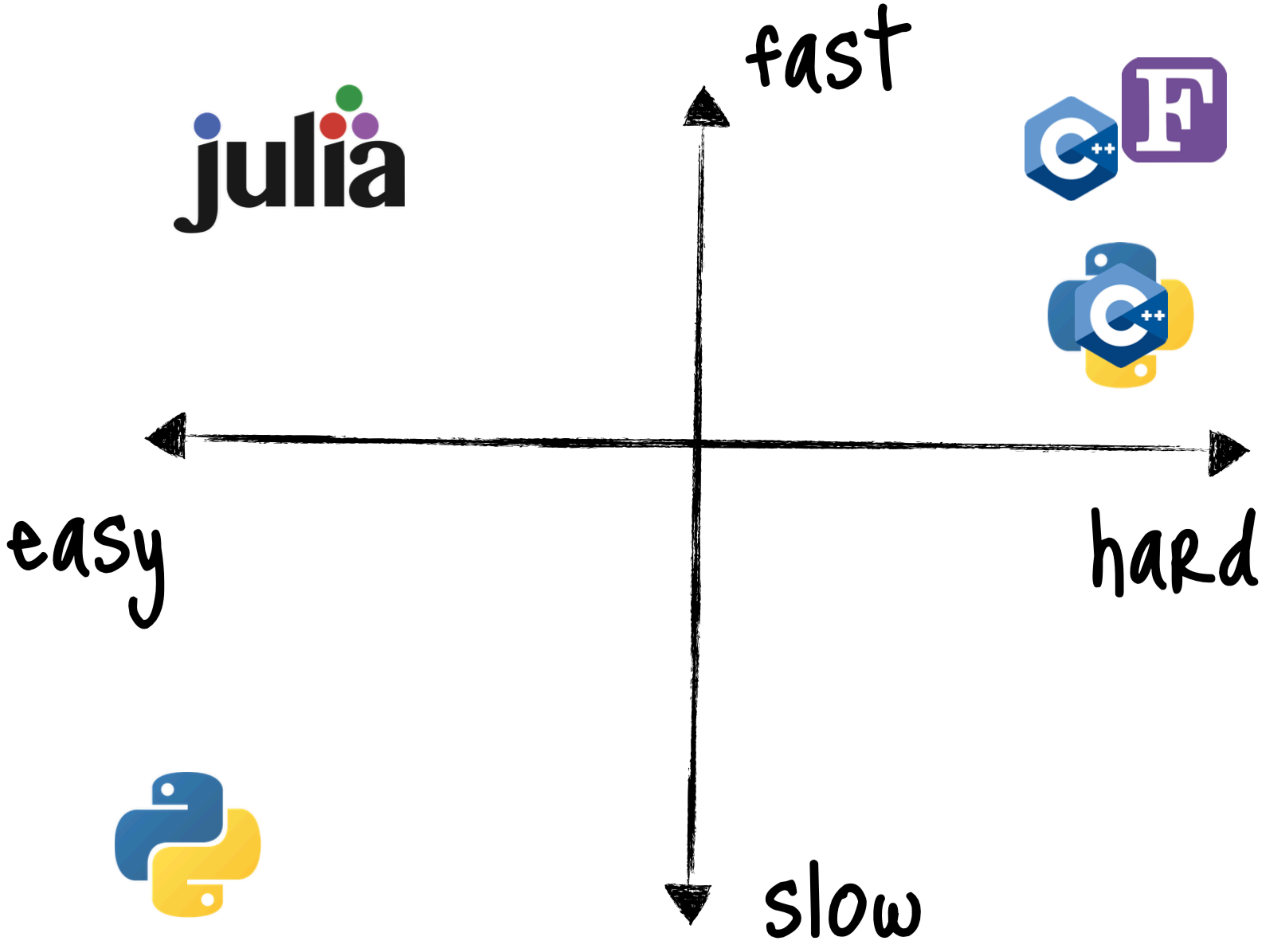
Metric		Python
Performance		❌
Expressiveness		
Learning Curve	❌	
Safety (memory)	⚠️	
Composability	❌	⚠️

Code often gets started in Python, then at some point is rewritten in C++…

The Two Language Problem

Julia Motivations

- Invented 2012 at MIT (mostly)
- Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman
- Design goals and aims
  - Open source
  - Speed like C, but dynamic like Ruby/Python
  - Obvious mathematical notation
  - General purpose like Python
  - As easy for statistics as R
  - Powerful linear algebra like in Matlab
  - Good for gluing programs together like the shell



We love all of these languages [Matlab, Lisp, Perl, Ruby, Mathematica, C]; they are wonderful and powerful. For the work we do — scientific computing, ... — each one is perfect for some aspects of the work and terrible for others. Each one is a trade-off.

... we want more.

Something that is dirt simple to learn, yet keeps the most serious hackers happy.

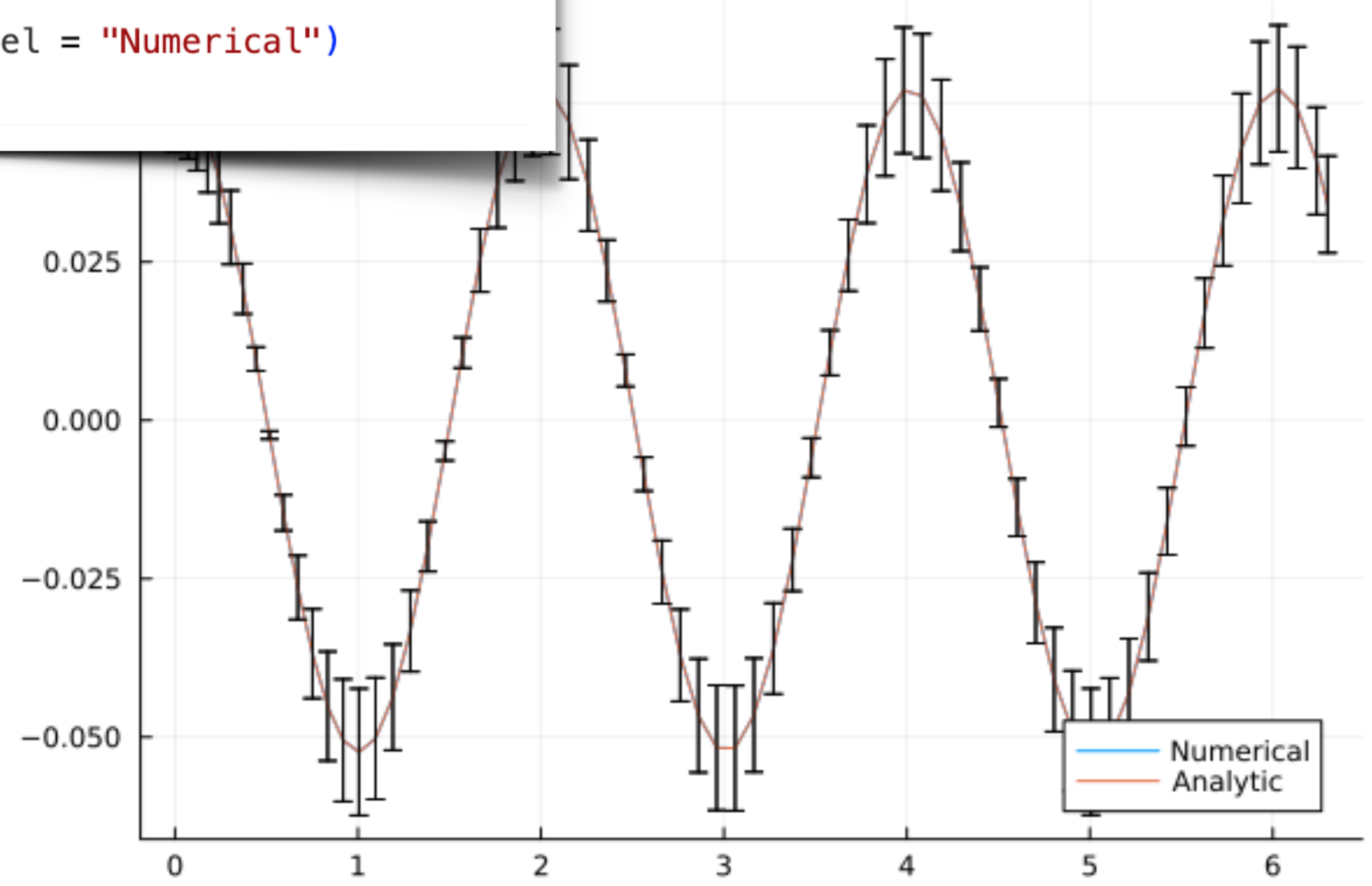
[Bezanson, Karpinski, Shah, Edelman - "Why We Created Julia" (2012)]

Julia in Practice

Julia is Easy

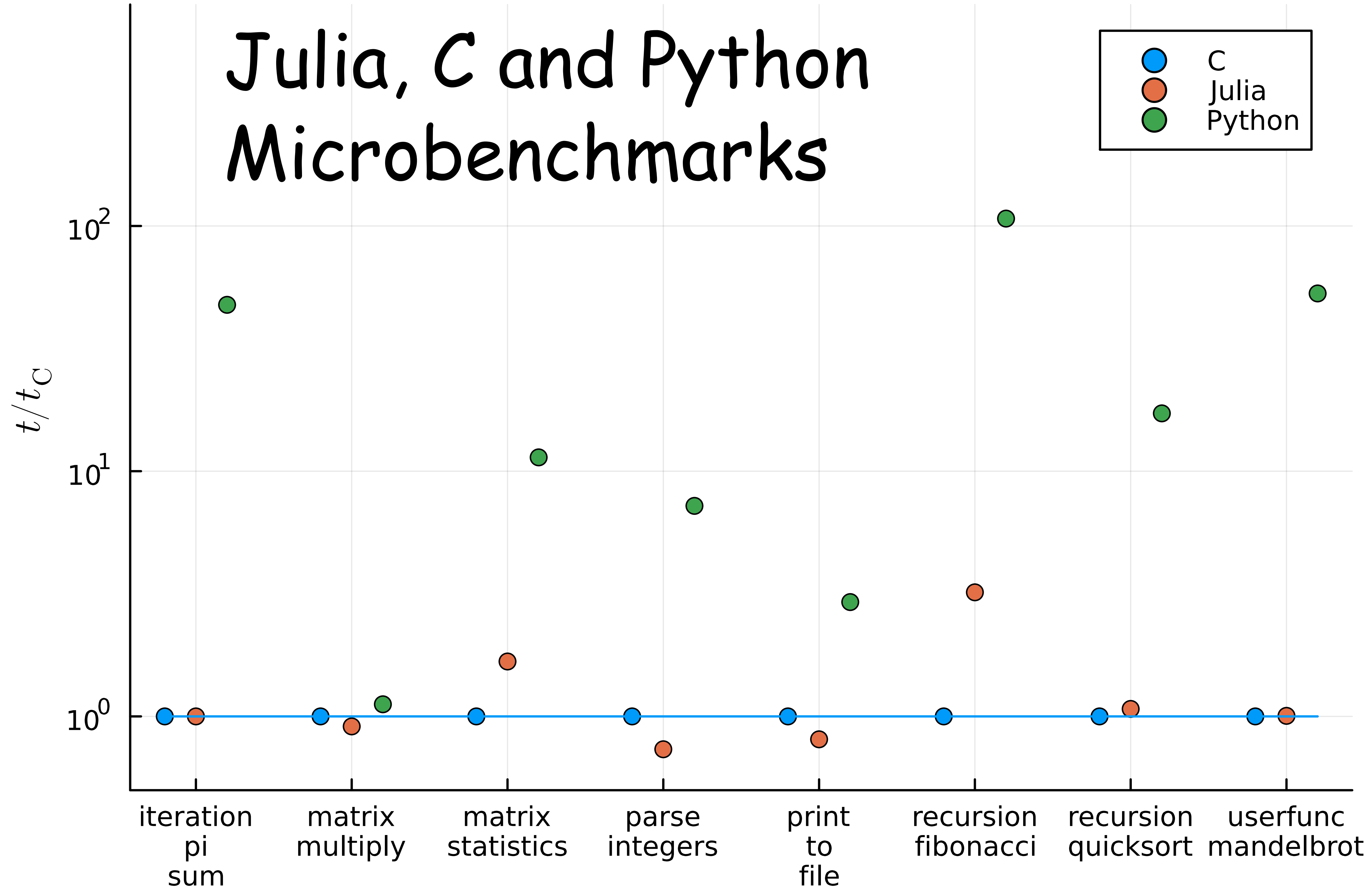
- Excellent REPL mode and notebooks
    - Jupyter (you know it) plus reactive notebooks in Pluto
- Dynamically typed (runtime), but with a powerful type system
- Garbage collected
- Expressive maths syntax
    - Mathematical symbols and notation can be written directly
- Extensive standard library
    - Mostly written in Julia - high performance and comprehensible

```
using DifferentialEquations, Measurements, Plots

g = 9.79 ± 0.02; # Gravitational constants
L = 1.00 ± 0.01; # Length of the pendulum

#Initial Conditions
u₀ = [0 ± 0, π / 60 ± 0.01] # Initial speed and initial angle
tspan = (0.0, 6.3)

#Define the problem
function pendulum(du,u,p,t)
    θ  = u[1]
    dθ = u[2]
    du[1] = dθ
    du[2] = -(g/L)*θ
end

#Pass to solvers
prob = ODEProblem(pendulum, u₀, tspan)
sol = solve(prob, Tsit5(), reltol = 1e-6)

# Analytic solution
u = u₀[2] .* cos.(sqrt(g / L) .* sol.t)

plot(sol.t, getindex.(sol.u, 2), label = "Numerical")
plot!(sol.t, u, label = "Analytic")
```



Julia is Fast

- It's not an interpreter
- Just ahead of time compiler (JAOT)
  - Powered by LLVM
  - Specialises and de-virtualises
- Built in vectors and arrays
  - Static sizing available
- Pinpoint optimisation (`@fastmath`*, `@simd`)
- Reflection and metaprogramming built in
- User friendly native support for threads
- And GPU support too!



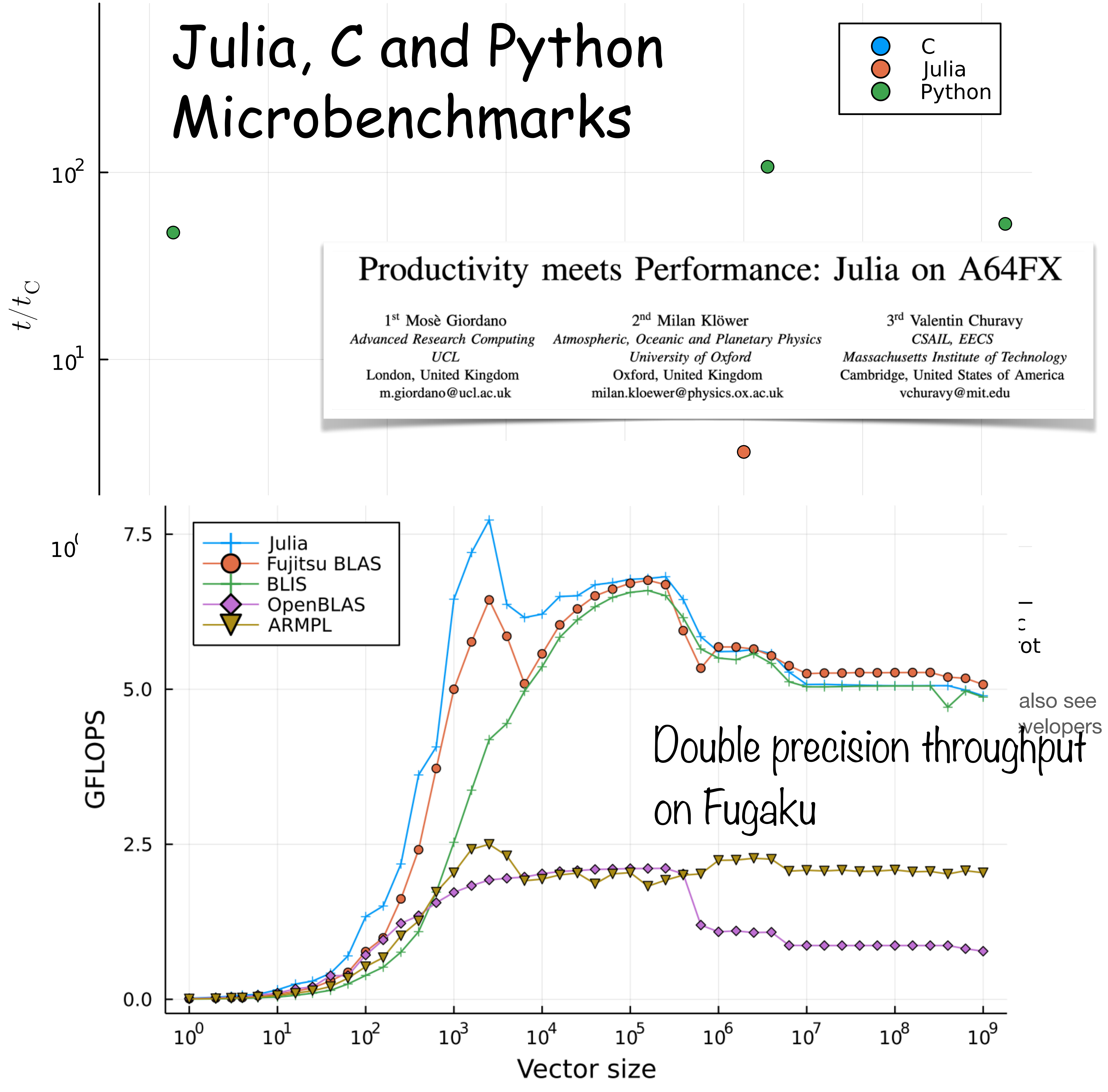
Data taken from https://julialang.org/benchmarks/, also see more sophisticated benchmarks from the Chapel developers

*Can be applied to single statements or code blocks, more aggressive than `--ffast-math`

Julia is Fast

- It's not an interpreter
- Just ahead of time compiler (JAOT)
  - Powered by LLVM
  - Specialises and de-virtualises
- Built in vectors and arrays
  - Static sizing available
- Pinpoint optimisation (`@fastmath`*, `@simd`)
- Reflection and metaprogramming built in
- User friendly native support for threads
- And GPU support too!

*Can be applied to single statements or code blocks, more aggressive than `--ffast-math`



From [arXiv: 2207.12762], Mosè Giordano, Milan Klöwer, Valentin Churavy

Tooling is Great

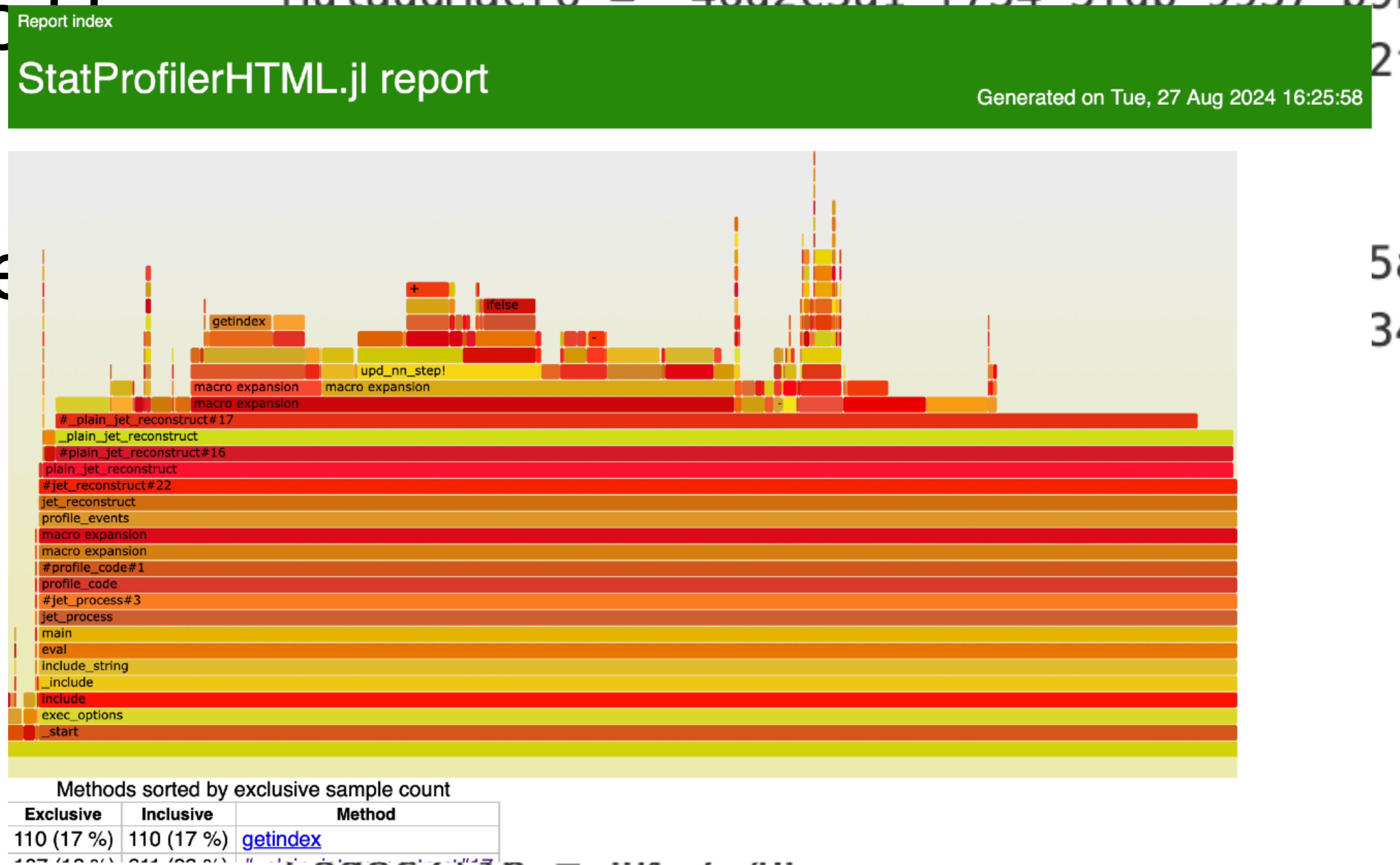
- Julia has an outstanding package manager
  - Express package interdependence with as few or as many constraints as needed - `Project.toml`
  - Preserve an exact environment for reproducibility - `Manifest.toml` (with binary reps)
  - Easy to create and register your own packages
  - Semantic versioning universally adopted
- Built in profiling and debugging
- First class VSCode integration
- Easy to use package documentation system

```
name = "JetReconstruction"
uuid = "44e8cb2c-dfab-4825-9c70-d4808a591196"
authors = ["Atell Krasnopolski <delta_atell@protonmail.com>",
  "Graeme A Stewart <graeme.andrew.stewart@cern.ch",
  "Philippe Gras <philippe.gras@cern.ch>"]
version = "0.4.2"

[deps]
Accessors = "7d9f7c33-5ae7-4f3b-8dc6-eff91059b697"
CodecZlib = "944b1d66-785c-5afd-91f1-9de20f533193"
EnumX = "4e289a0a-7415-4d19-859d-a7e5c4648b56"
JSON = "682c06a0-de6a-54ab-a142-c8b1cf79cde6"
Logging = "56ddb016-857b-54e1-b83d-db4d58db5568"
LoopVectorization = "bdcacae8-1622-11e9-2a5c-532679323890"
LorentzVectorHEP = "f612022c-142a-473f-8cfd-a09cf3793c6c"
LorentzVectors = "3f54b04b-17fc-5cd4-9758-90c048d965e3"
MuladdMacro = "46d2c3a1-f734-5fdb-9937-b9b9aeba4221"
StructArrays = "09ab397b-f2b6-538f-b94a-2f83cf4a842a"

[weakdeps]
Makie = "ee78f7c6-11fb-53f2-987a-cfe4a2b5a57a"
EDM4hep = "eb32b910-dde9-4347-8fce-cd6be3498f0c"

[extensions]
JetVisualisation = "Makie"
EDM4hepJets = "EDM4hep"

[compat]
Accessors = "0.1.36"
CodecZlib = "0.7.4"
EDM4hep = "0.4"
EnumX = "1.0.4"
JSON = "0.21.4"
```

Tooling is Great

- Julia has an outstanding package manager
    - Express package interdependence with as few or as many constraints as needed - `Project.toml`
    - Preserve an exact environment for reproducibi… `Manifest.toml` (with binary reps)
    - Easy to create and register your own package…
    - Semantic versioning universally adopted
- Built in profiling and debugging
- First class VSCode integration
- Easy to use package documentation system

```
name = "JetReconstruction"
uuid = "44e8cb2c-dfab-4825-9c70-d4808a591196"
authors = ["Atell Krasnopolski <delta_atell@protonmail.com>",
  "Graeme A Stewart <graeme.andrew.stewart@cern.ch",
  "Philippe Gras <philippe.gras@cern.ch>"]
version = "0.4.2"

[deps]
Accessors = "7d9f7c33-5ae7-4f3b-8dc6-eff91059b697"
CodecZlib = "944b1d66-785c-5afd-91f1-9de20f533193"
EnumX = "4e289a0a-7415-4d19-859d-a7e5c4648b56"
JSON = "682c06a0-de6a-54ab-a142-c8b1cf79cde6"
Logging = "56ddb016-857b-54e1-b83d-db4d58db5568"
LoopVectorization = "bdcacae8-1622-11e9-2a5c-532679323890"
LorentzVectorHEP = "f612022c-142a-473f-8cfd-a09cf3793c6c"
LorentzVectors = "3f54b04b-17fc-5cd4-9758-90c048d965e3"
MuladdMacro = "46d2c3a1-f734-5fdb-9937-b9b9aeba4221"
2f83cf4a842a"
5a57a"
3498f0c"
CodecZlib = "0.7.4"
EDM4hep = "0.4"
EnumX = "1.0.4"
JSON = "0.21.4"
```



Tooling is Great

- Julia has an outstanding package manager
  - Express package interdependence with as few or as many constraints as needed - `Project.toml`
  - Preserve an exact environment for reproducibility - `Manifest.toml` (with binary reps)
  - Easy to create and register your own package
  - Semantic versioning universally adopted
- Built in profiling and debugging
- First class VSCode integration
- Easy to use package documentation system

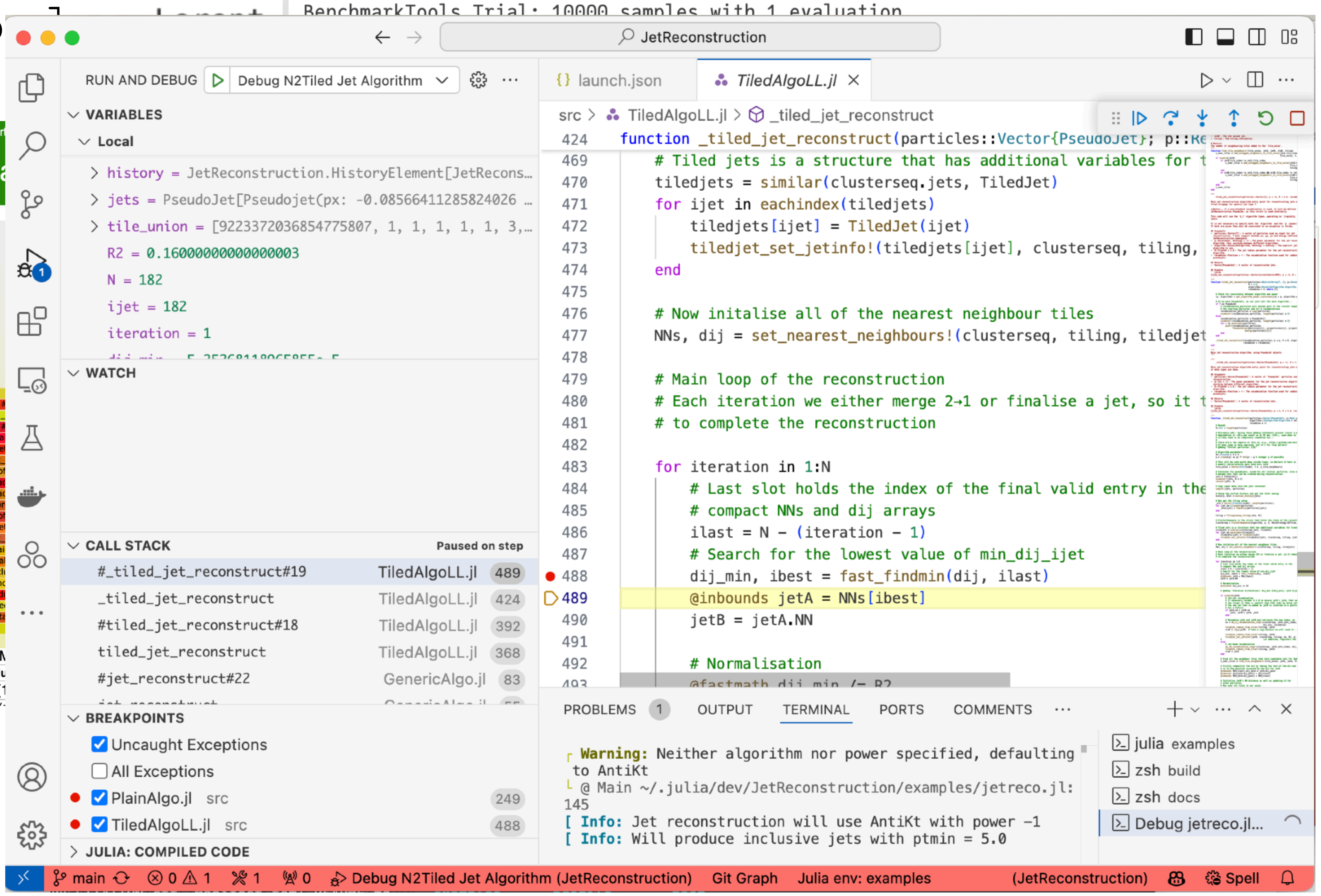
Tooling is Great

```
name = "JetReconstruction"
uuid = "44e8cb2c-dfab-4825-9c70-d4808a591196"
authors = ["Atell Krasnopolski <delta_atell@protonmail.com>",
  "Graeme A Stewart <graeme.andrew.stewart@cern.ch",
  "Philippe Gras <philippe.gras@cern.ch>"]
version = "0.4.2"

[deps]
Accessors = "7d9f7c33-5ae7-4f3b-8dc6-eff91059b697"
CodecZlib = "944b1d66-785c-5afd-91f1-9de20f533193"
EnumX = "4e289a0a-7415-4d19-859d-a7e5c4648b56"
JSON = "682c06a0-de6a-54ab-a142-c8b1cf79cde6"
Loggin
LoopVe
```

Fast findmin (vectorised)

BenchmarkTools.Trial: 10000 samples with 1 evaluation.

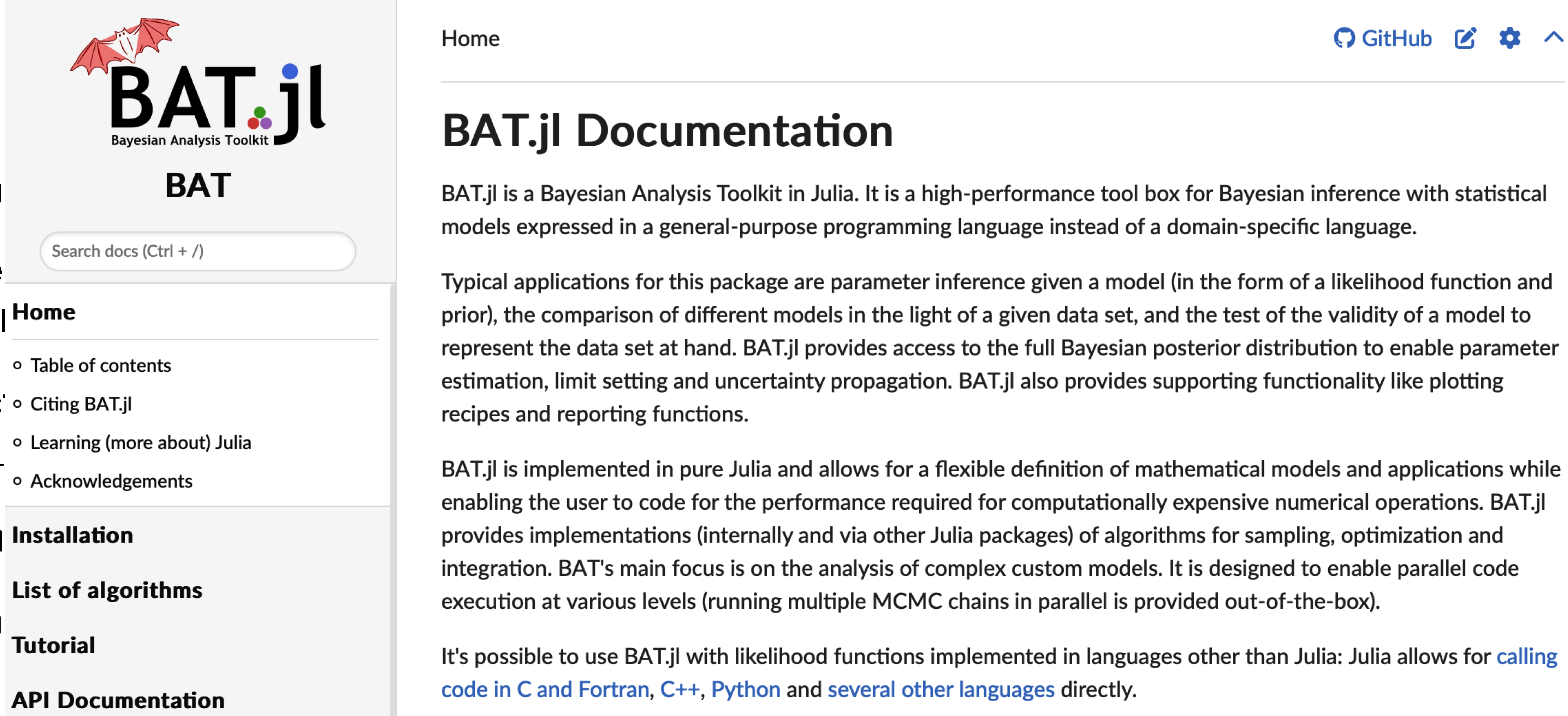
- Julia has an outstanding package manager
  - Express package interdependence with as few or as many constraints as needed - `Project.to`
  - Preserve an exact environment for reproducib `Manifest.toml` (with binary reps)
  - Easy to create and register your own package
  - Semantic versioning universally adopted
- Built in profiling and debugging
- First class VSCode integration
- Easy to use package documentation system

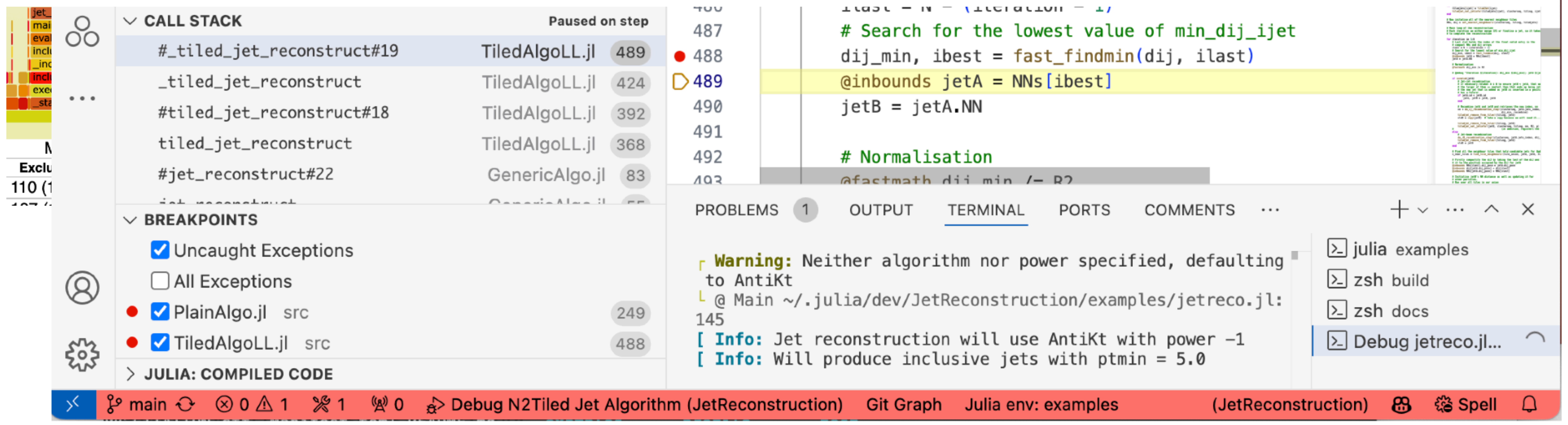


```
name = "JetReconstruction"
uuid = "44e8cb2c-dfab-4825-9c70-d4808a591196"
authors = ["Atell Krasnopolski <delta_atell@protonmail.com>",
  "Graeme A Stewart <graeme.andrew.stewart@cern.ch",
  "Philippe Gras <philippe.gras@cern.ch>"]
version = "0.4.2"
```

Tooling is Great

- Julia has an outstan
  - Express package
    as many constrai
  - Preserve an exac
    `Manifest.toml`
  - Easy to create an
  - Semantic version
- Built in profiling and
- First class VSCode integration
- Easy to use package documentation system





Rich Ecosystem

More than 10k packages available

Visualization

Plots.jl

Makie.jl

PGFPlots.jl

Data and Statistics

JuliaData

JuliaStats

BAT.jl

Bayesian Analysis Toolkit

Turing.jl

Machine learning

MJL.jl

Flux.jl

SciML

JuliaDiff

Notebooks

Pluto.jl

IJulia.jl

GPU support

JuliaGPU

CUDA.jl

AMDGPU.jl

oneAPI.jl

Metal.jl

KernelAbstractions.jl

Interoperability

JuliaInterop

CxxWrap.jl

PythonCall.jl

RCall.jl

MathLink.jl

Rich Ecosystem

More than 10k packages available

Visualization

Plots.jl

Makie.jl

Data and Statistics

JuliaData

JuliaStats

Machine learning

MJL.jl

Flux.jl

JuliaDiff

```
pkg > add DifferentialEquations
. . .
julia > using DifferentialEquations
```

Notebooks

Pluto.jl

IJulia.jl

JuliaGPU

CUDA.jl

AMDGPU.jl

oneAPI.jl

Metal.jl

KernelAbstractions.jl

nteroperability

JuliaInterop

CxxWrap.jl

PythonCall.jl

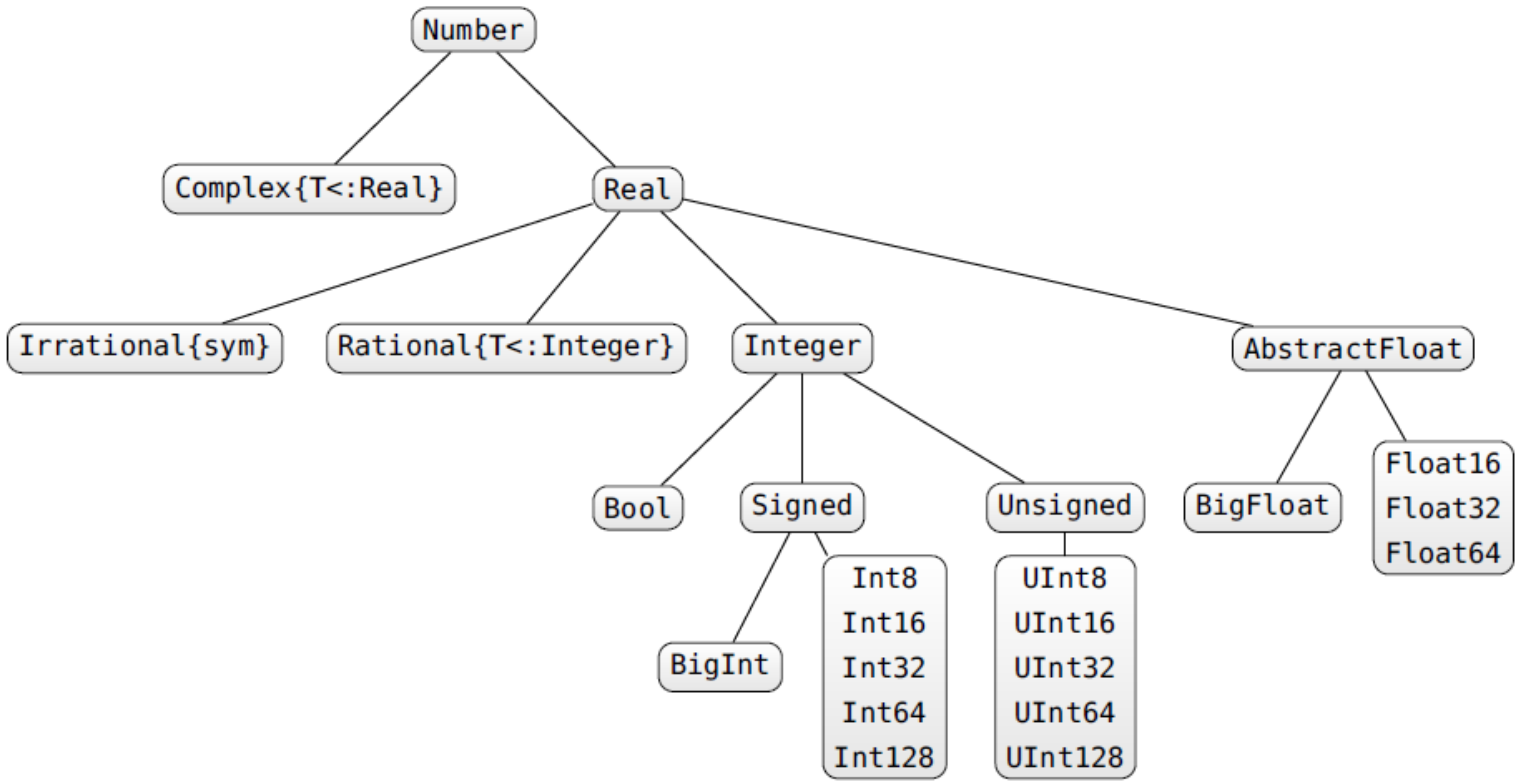
RCall.jl

MathLink.jl

Secret Sauce I

Type system

- Julia has an advanced type system (based on set theory)
  - Basic types are part of the type system
  - Concrete types are always leaves - performance!
  - The tree terminates at `::Any`
- Hierarchy
  - `A <: B` - type A is a subtype of B
  - `B >: A` - type B is a supertype of A
- Powerful and sophisticated type expressions
  - `AbstractArray{T, 2}` - expresses any two dimensional array type of Ts
  - And there are many array types (dense, sparse, diagonal, tri-diagonal, static, GPU arrays…) - any of them will work here



Secret Sauce II

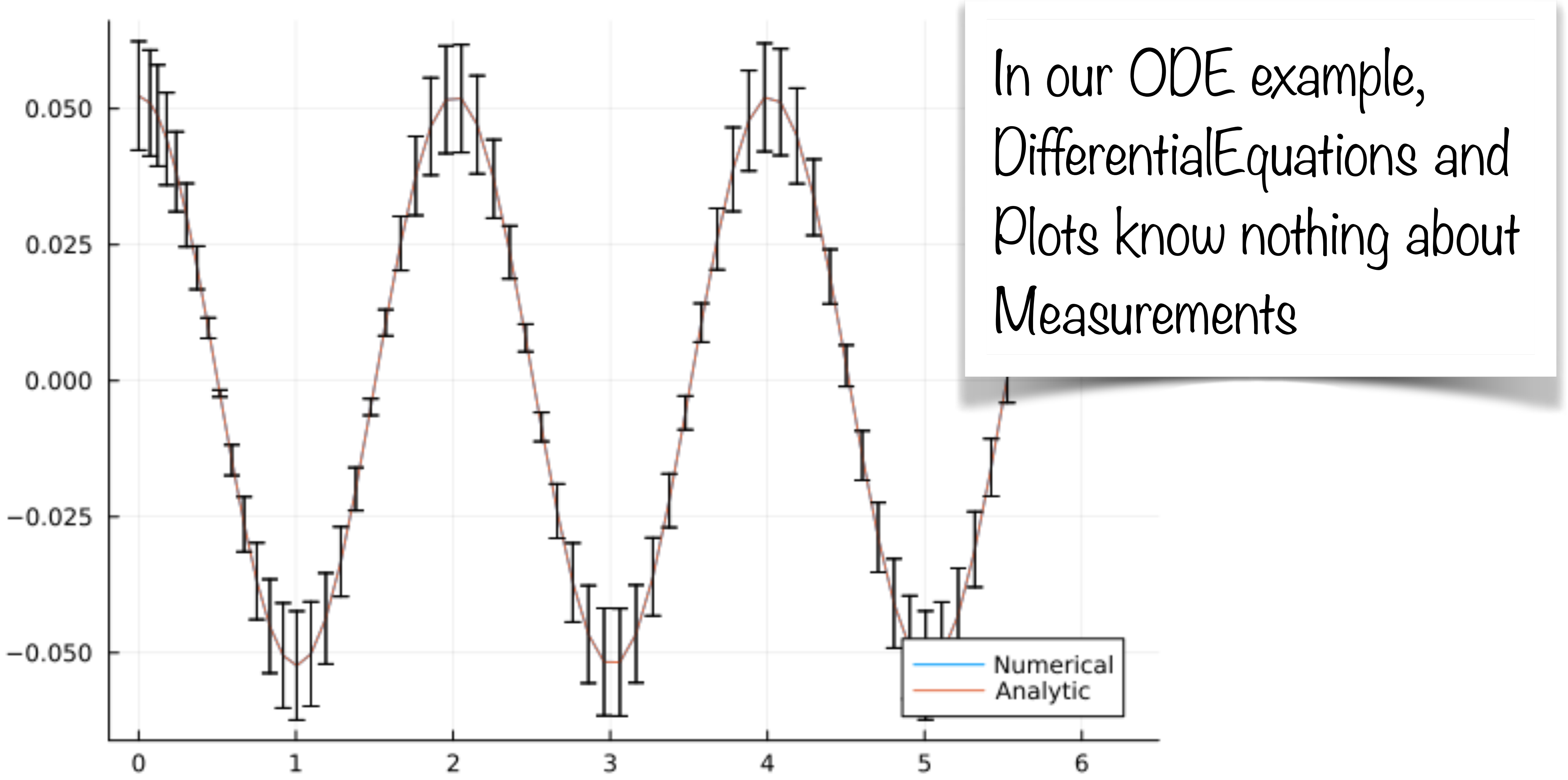
Multiple dispatch

- Multiple dispatch
    - Choice of method to use depends dynamically on *all* argument types

```
foo(f::Real, g::Real)
foo(f::Real, g::Complex)
foo(f::AbstractFloat, g::Complex)
foo(f::AbstractFloat, AbstractArray{T, 1})
foo(f::AbstractFloat, AbstractArray{T, 2})
foo(f::Float64, SparseArray{T, 2})
```

    - All of these things will `foo` their arguments but the implementation can be optimised
    - And the compiler will generate low level machine code for each method

- *You can add methods for types defined in other packages*
- This allows packages to **compose** without knowing about each other



See *The Unreasonable Effectiveness of Multiple Dispatch*, Stefan Karpinski [a, b, c]

Julia for GPUs and Scientific Computing

Julia on GPUs

- Julia's JAOT compilation model makes it ideal for running on GPUs
  - Compiler can target the specific GPU model at runtime
- Supported backends are `CUDA.jl`, `AMDGPU.jl`, `Metal.jl` and `OneAPI.jl`
- Applications with GPU support are easy

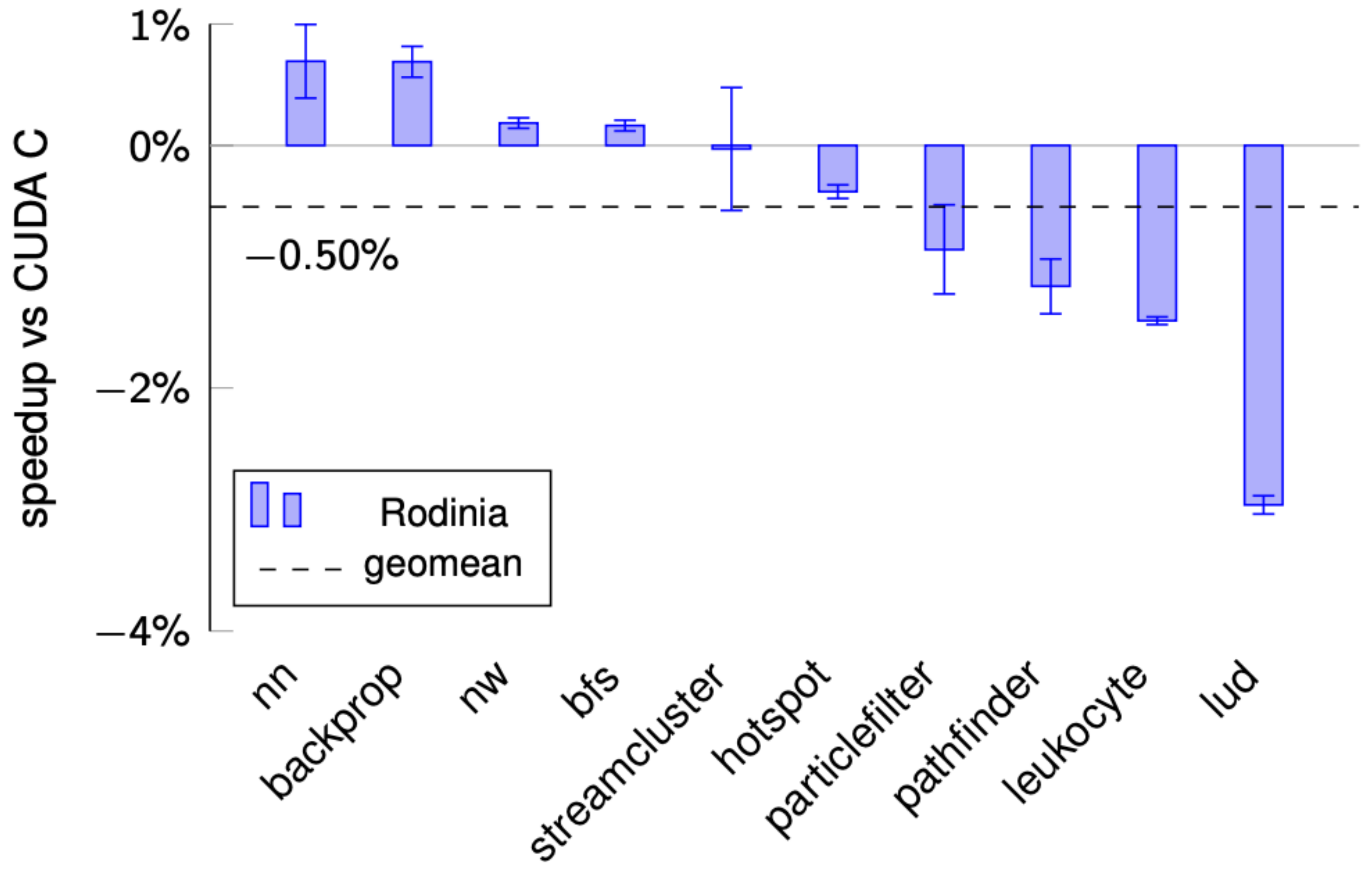
```
m = Dense(10,5) |> gpu
```

- Array based calculations are trivial to execute on the GPU

```
a = CuArray([1 2 3]) # CuArray allocates
                     # on the device

a * 2
```

  - Packages which do LinearAlgebra, FFTs, Neural Networks, etc. all support the GPU backends
- Kernel programming is close to the native toolkits
- `KernelAbstractions.jl` allows writing of generic code, backend independent

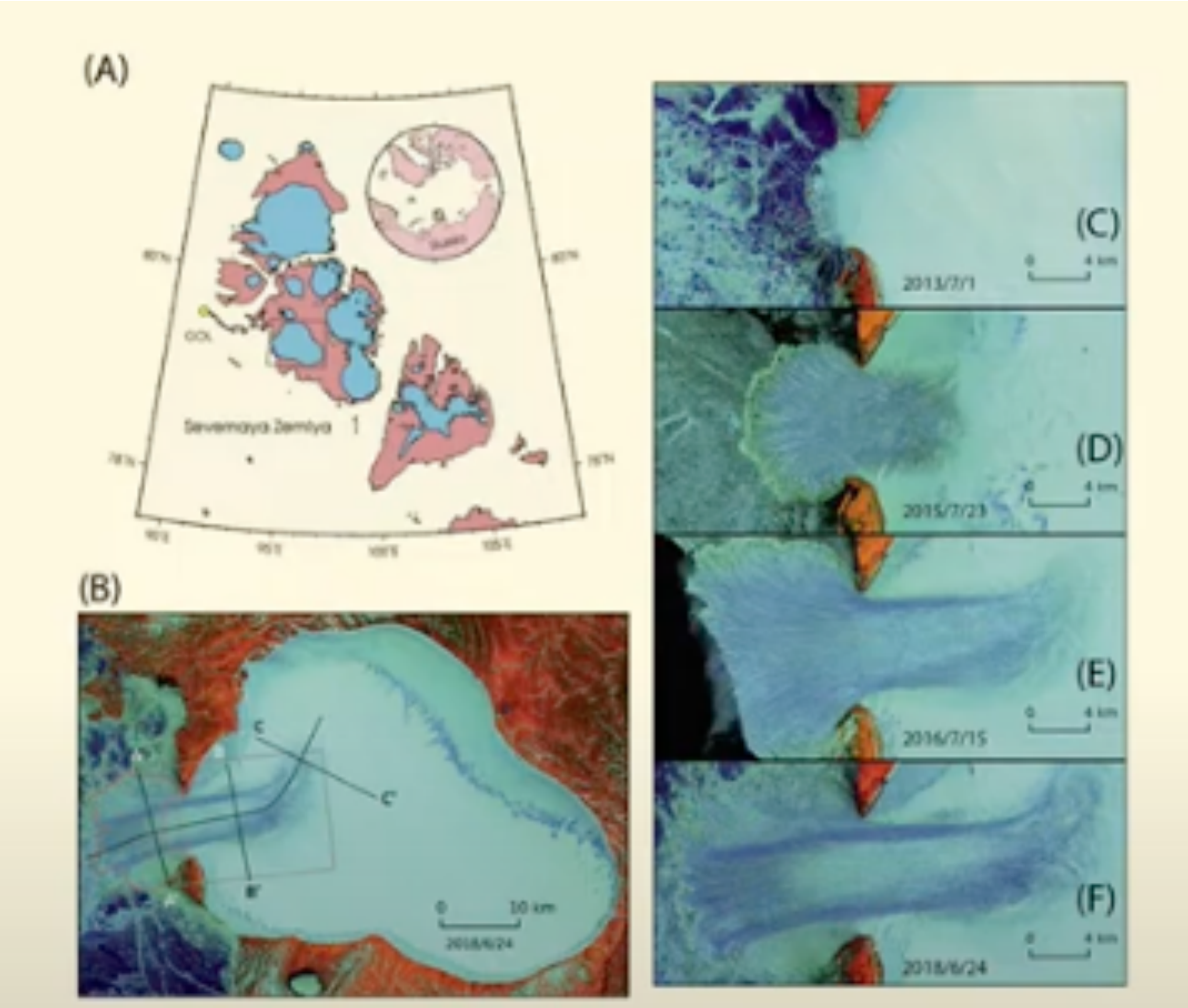
GPU performance



Rodinia benchmarks [implemented in Julia with CUDA.jl](), Besard, Tim, et al. "Rapid software prototyping for heterogeneous and distributed platforms." Advances in engineering software 132 (2019): 29-46

Julia HPC Codes

- `FastIce.jl` is a state of the art thermo-mechanically coupled full-Stokes ice flow model
  - Essential to model correctly ice streams - corridors of fast flowing ice, 3km/year
  - Multiscale problem: ice sheets 1000km, stream 100km, shear margins <1km
- Uses `KernelAbstractions.jl`
  - Write code once, run on any GPU
- Code is extremely close to the maths!
  - Aim for locality in the code
  - Asynchronous non-blocking communication
- Scales up to 21k GPUs at >90% weak scaling on LUMI



Zheng et al, 2019

Chmy.jl

```julia
@kernel inbounds = true function update_σ!(Pr, τ, V, η, Δτ, g::StructuredGrid{3}, O=Offset())
    I = @index(Global, Cartesian)
    I += O
    # strain rates
    ε̇xx = ∂x(V.x, g, I)
    ε̇yy = ∂y(V.y, g, I)
    ε̇zz = ∂z(V.z, g, I)
    ε̇xy = 0.5 * (∂x(V.y, g, I) + ∂y(V.x, g, I))
    ε̇xz = 0.5 * (∂x(V.z, g, I) + ∂z(V.x, g, I))
    ε̇yz = 0.5 * (∂y(V.z, g, I) + ∂z(V.y, g, I))
    # velocity divergence
    ∇V = ε̇xx + ε̇yy + ε̇zz
    # hydrostatic stress
    Pr[I] -= ∇V * lerp(η, location(Pr), g, I) * Δτ.Pr
    # deviatoric diagonal
    τ.xx[I] -= (τ.xx[I] - 2.0 * lerp(η, location(τ.xx), g, I) * (ε̇xx - ∇V / 3.0)) * Δτ.τ.xx
    τ.yy[I] -= (τ.yy[I] - 2.0 * lerp(η, location(τ.yy), g, I) * (ε̇yy - ∇V / 3.0)) * Δτ.τ.yy
    τ.zz[I] -= (τ.zz[I] - 2.0 * lerp(η, location(τ.zz), g, I) * (ε̇zz - ∇V / 3.0)) * Δτ.τ.zz
    # deviatoric off-diagonal
    τ.xy[I] -= (τ.xy[I] - 2.0 * lerp(η, location(τ.xy), g, I) * ε̇xy) * Δτ.τ.xy
    τ.xz[I] -= (τ.xz[I] - 2.0 * lerp(η, location(τ.xz), g, I) * ε̇xz) * Δτ.τ.xz
    τ.yz[I] -= (τ.yz[I] - 2.0 * lerp(η, location(τ.yz), g, I) * ε̇yz) * Δτ.τ.yz
end
```

$\dot{\varepsilon}_{xx} = \partial V_x/\partial x$
$\dot{\varepsilon}_{yy} = \partial V_y/\partial y$
$\dot{\varepsilon}_{zz} = \partial V_z/\partial z$
$\dot{\varepsilon}_{xy} = 1/2(\partial V_y/\partial x + \partial V_x/\partial y)$
$\dot{\varepsilon}_{xz} = 1/2(\partial V_z/\partial x + \partial V_x/\partial z)$
$\dot{\varepsilon}_{yz} = 1/2(\partial V_z/\partial y + \partial V_y/\partial z)$

$\nabla V = \dot{\varepsilon}_{xx} + \dot{\varepsilon}_{yy} + \dot{\varepsilon}_{zz}$

$\tau_{xx} = 2\eta(\dot{\varepsilon}_{xx} - \nabla V/3)$
$\tau_{yy} = 2\eta(\dot{\varepsilon}_{yy} - \nabla V/3)$
$\tau_{zz} = 2\eta(\dot{\varepsilon}_{zz} - \nabla V/3)$

$\tau_{xy} = 2\eta\dot{\varepsilon}_{xy}$
$\tau_{xz} = 2\eta\dot{\varepsilon}_{xz}$
$\tau_{yz} = 2\eta\dot{\varepsilon}_{yz}$

Now: the code is very close to the math notation

Languages that have run at the PetaFlop level: C/C++, Fortran and Julia

Julia in HEP

Challenges of HEP Computing

- Large data volumes
- High computational costs
- Large-scale heterogeneous environments
- Legacy and maintenance
  - Old codebases
  - Interoperability
- Human challenges
  - Train people to be effective fast (and retain)

HEP computing collaborations for the challenges of the next decade

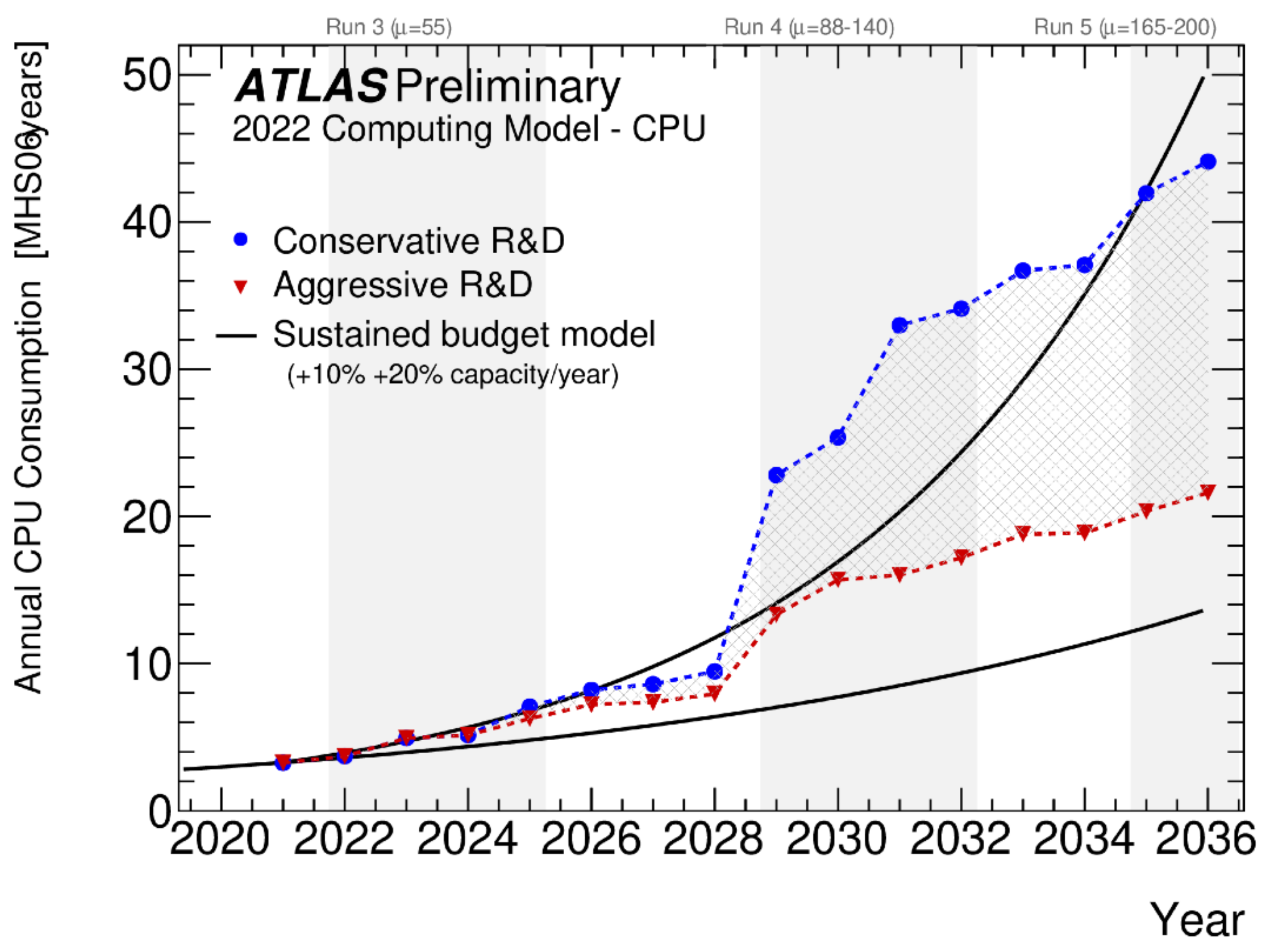
Contacts: Simone Campana (Simone.Campana@cern.ch), Zach Marshall (ZLMarshall@lbl.gov), Alessandro Di Girolamo (Alessandro.Di.Girolamo@cern.ch), Heidi Schellman (H... Stewart (grae...

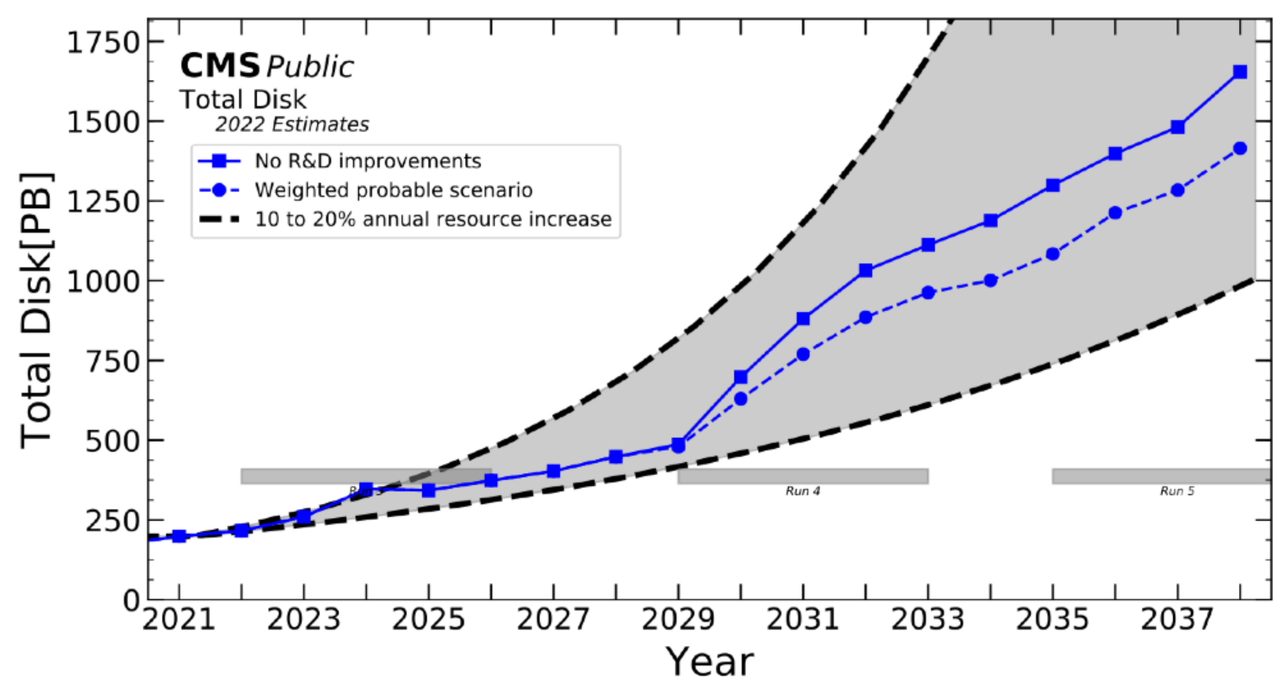
A Roadmap for HEP Software and Computing R&D for the 2020s

The HEP Software Foundation[5] · Johannes Albrecht[69] · Antonio Augusto Alves Jr[81] · Guilherme Amadio[5] · Giuseppe Andronico[27] · Nguyen Anh-Ky[122] · Laurent Aphecetche[66] · John Apostolakis[5] · Makoto Asai[63] · Luca Atzori[5] · Marian Babik[5] · Giuseppe Bagliesi[32] · Marilena Bandieramonte[5] · Sunanda Banerjee[16] · Martin Barisits[5] · Lothar A. T. Bauerdick[16] · Stefano Belforte[35] · Douglas Benjamin[82] · Catrin Bernius[63] · Wahid Bhimji[46] · Riccardo Maria Bianchi[105] · Ian Bird[5] · Catherine Biscarat[52] · Jakob Blomer[5] · Kenneth Bloom[97] · Tommaso Boccali[32] · ... Concezio Bozzi[28] · Ma...

Challenges in Monte Carlo Event Generator Software for High-Luminosity LHC

The HSF Physics Event Generator WG · Andrea Valassi[1] · Efe Yazgan[2] · Josh McFayden[1,3,4] · Simone Amoroso[5] · Joshua Bendavid[1] · Andy Buckley[6] · Matteo Cacciari[7,8] · Taylor Childers[9] · Vitaliano Ciulli[10] · Rikkert Frederix[11] · ... · Francesco Giuli[13] · Alexander Grohsjean[5] · Christian Gütschow[14] · Stefan Höche[15] · Philip Ilten[16,17] · Dmitri Konstantinov[18] · Frank Krauss[19] · Qiang Li[20] · Leif Lönnblad[11] · ... · Michelangelo Mangano[1] · Zach Marshall[3] · Olivier Mattelaer[22] · Javier Fernandez Menendez[23] · ... · Servesh Muralidharan[1,9] · Tobias Neumann[14,24] · Simon Plätzer[25] · Stefan Prestel[11] · ...arek Schönherr ... ...ndrzej Siódmo...

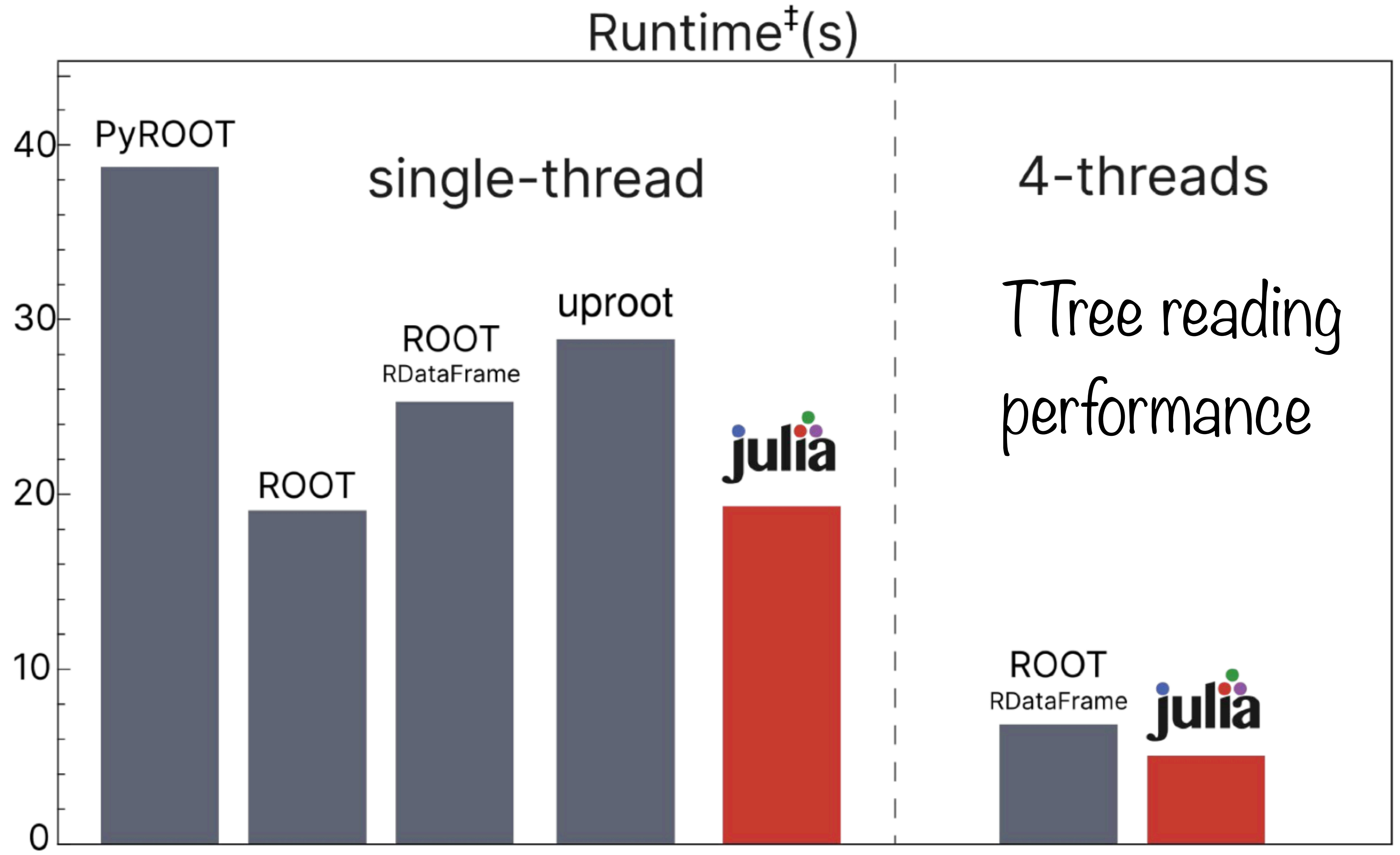




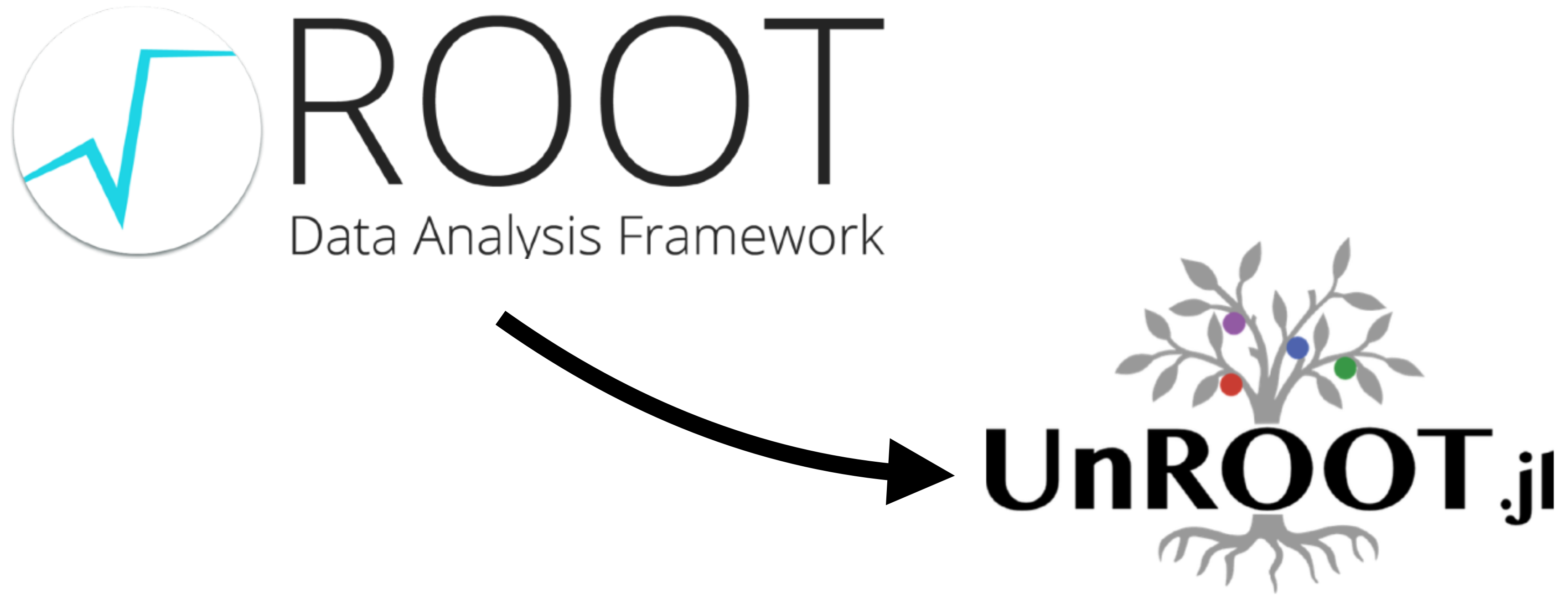
Data Formats in Julia

Including HEP data

- Reading industry standard data formats in Julia is well supported
    - HDF5, Parquet, Arrow
- We have some very specific data formats in HEP and these can be read too
    - `UnROOT.jl` is a pure Julia package that can read TTrees and RNTuples
    - Backend for `EMD4hep.jl`, implementing a complete EDM for future colliders
- Other HEP data format readers include LHEF and LCIO



‡: Exact ranking depends on the workload

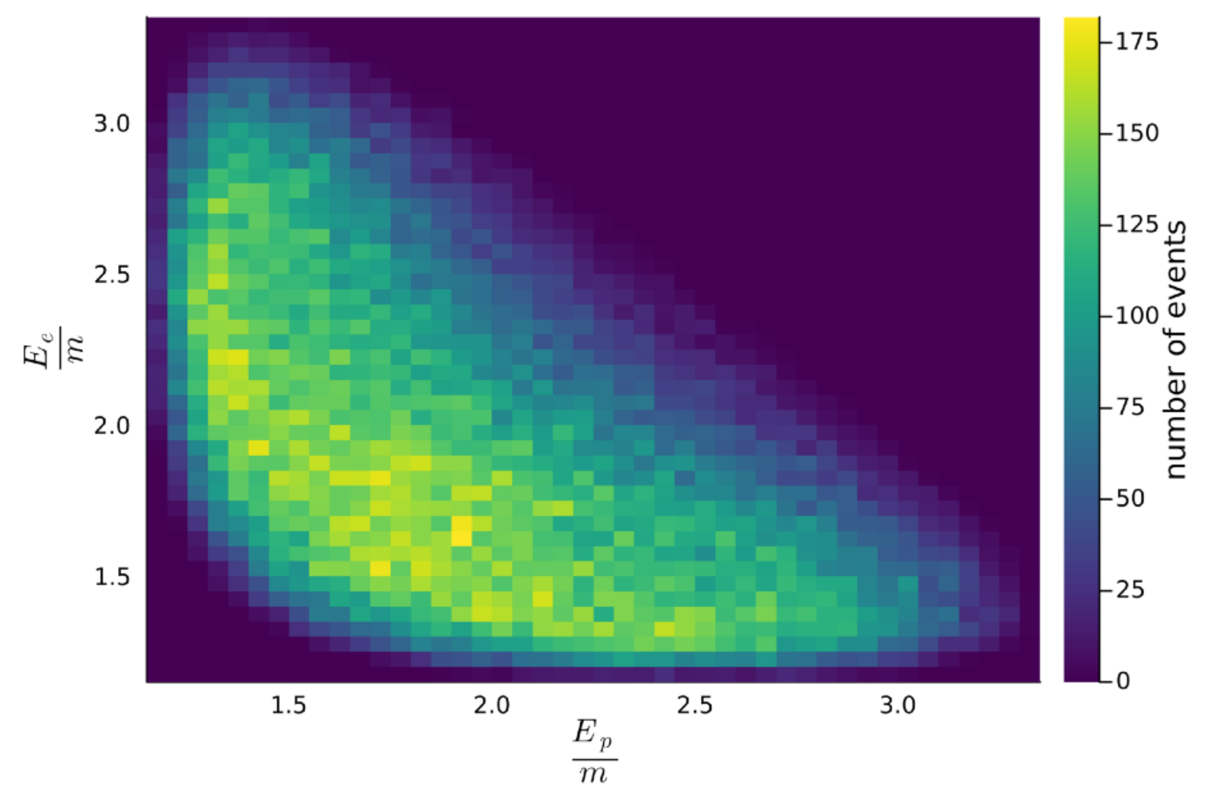


QuantumElectrodynamics.jl

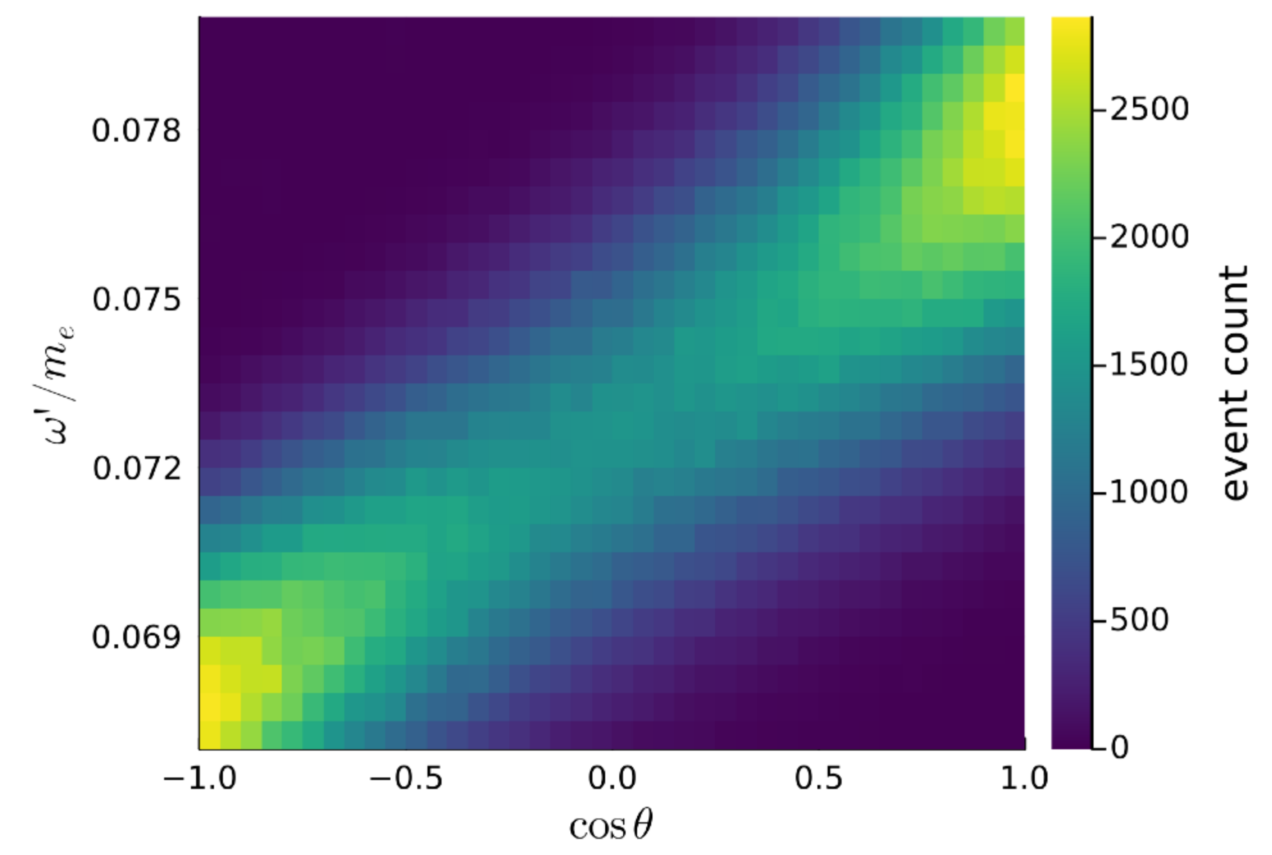
High performance QED generator

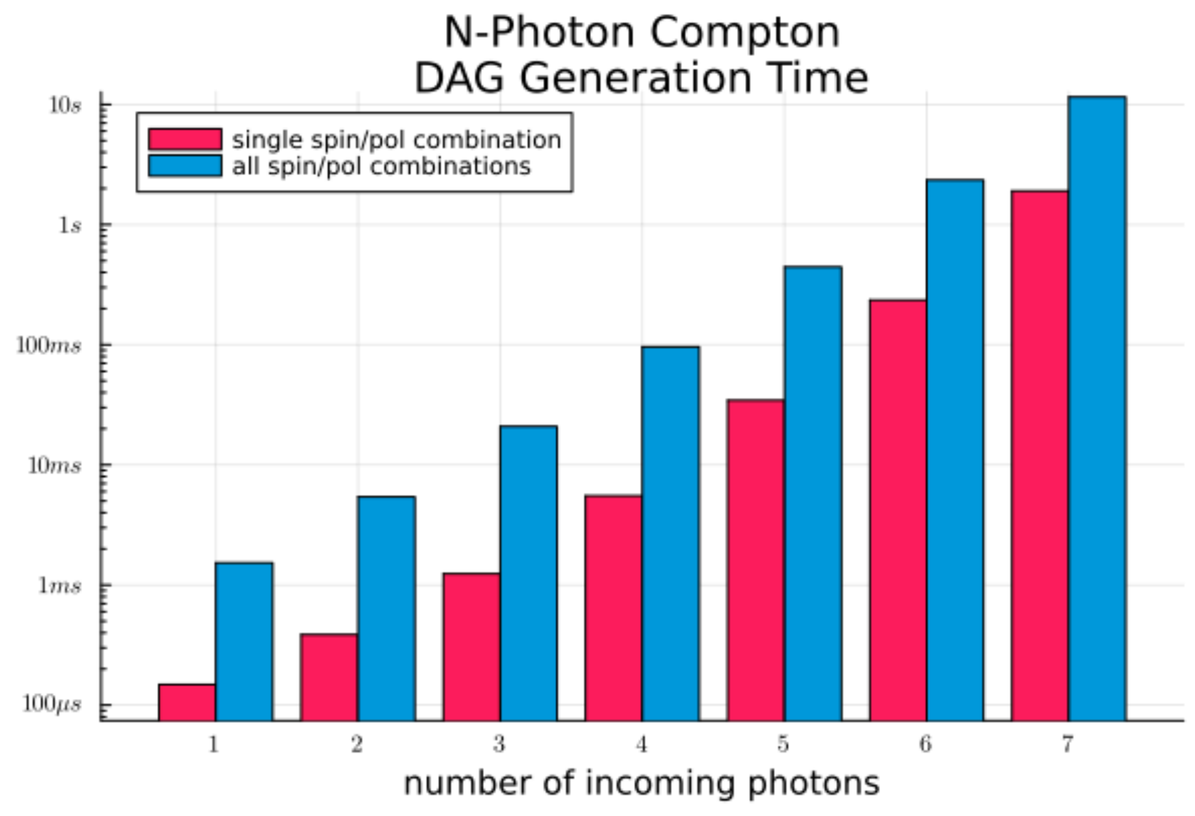
- All you need for QED!
  - Interfaces for particles; phase space points; computational models, scattering, distributions, laser fields, events
- Dev-tooling, eco-system and composability make development and end-to-end simulations much easier
- Type system allows the right level of *physics* abstractions
  - Adding new things is much easier
- Multiple dispatch used to slot in analytic formula, huge performance benefit
- High performance code generated, along with easy GPU integration

$e^- + \text{laser} \rightarrow e^- + (e^+e^-)$



$e^- + \text{laser} \rightarrow e^- + \gamma$

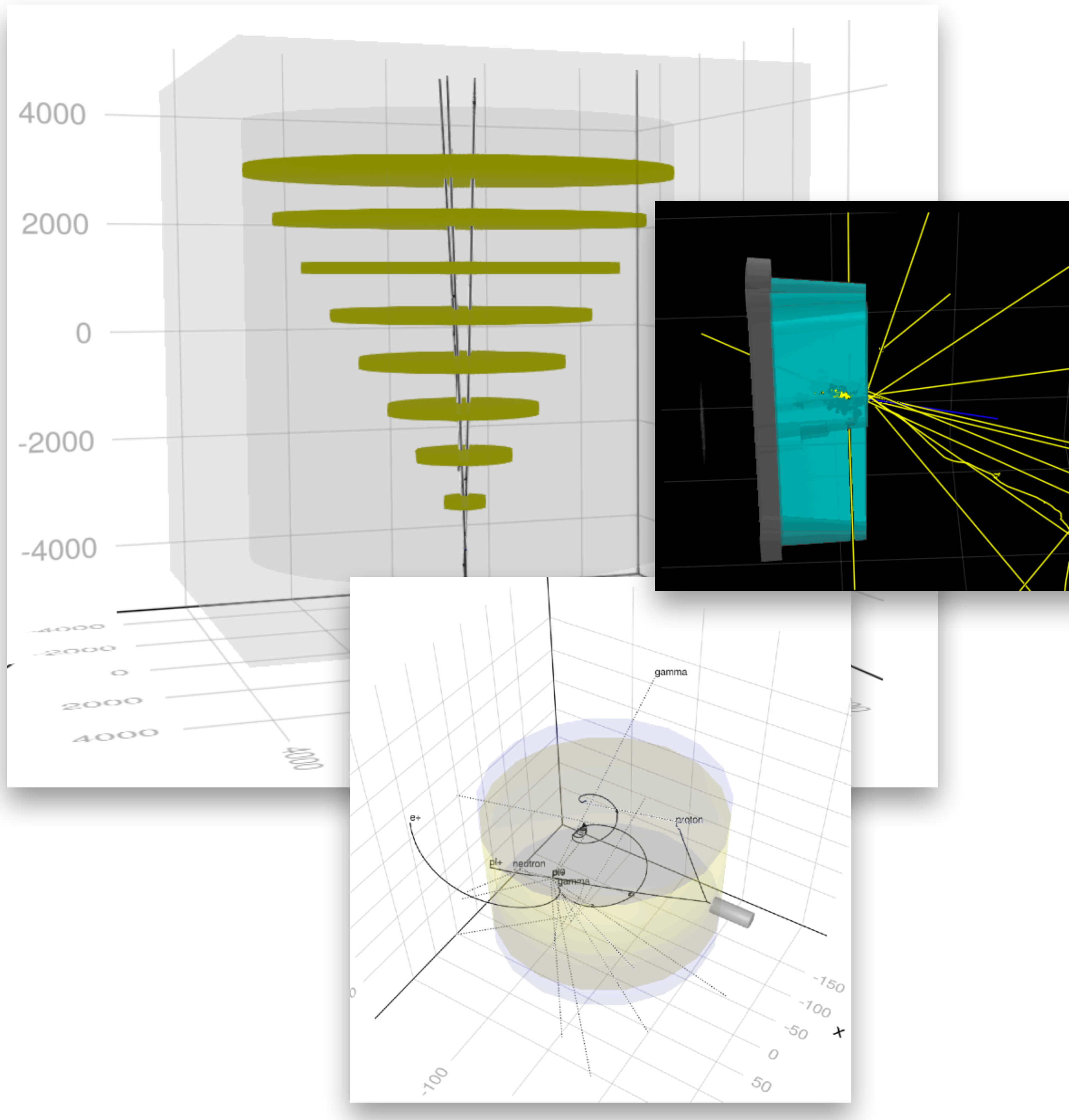




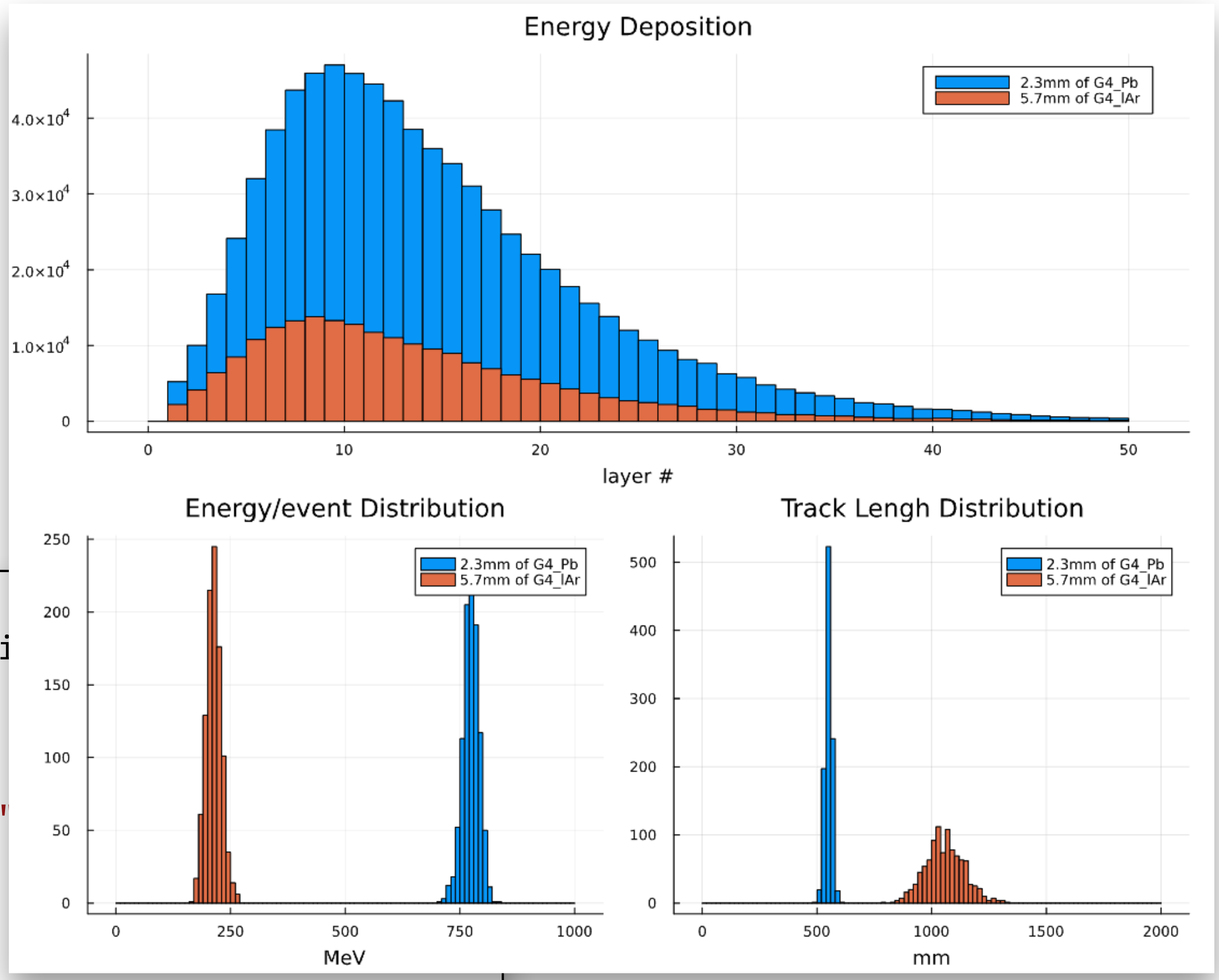
Geant4.jl

Or how to mesh easily with existing HEP codes

- How to make a large C++ application available in Julia?
  - Answer: `CxxWrap.jl` provides the binding layer and `WrapIt` helps automate the generation of these bindings
- The Geant4 C++ itself is provided via Julia's excellent BinaryBuilder system, making installation a snap!
- Improved user interfaces (less boilerplate) and an interactive environment
  - Speed is as fast as Geant4 native
- Use the power of Julia's visualisation and plotting packages to see results

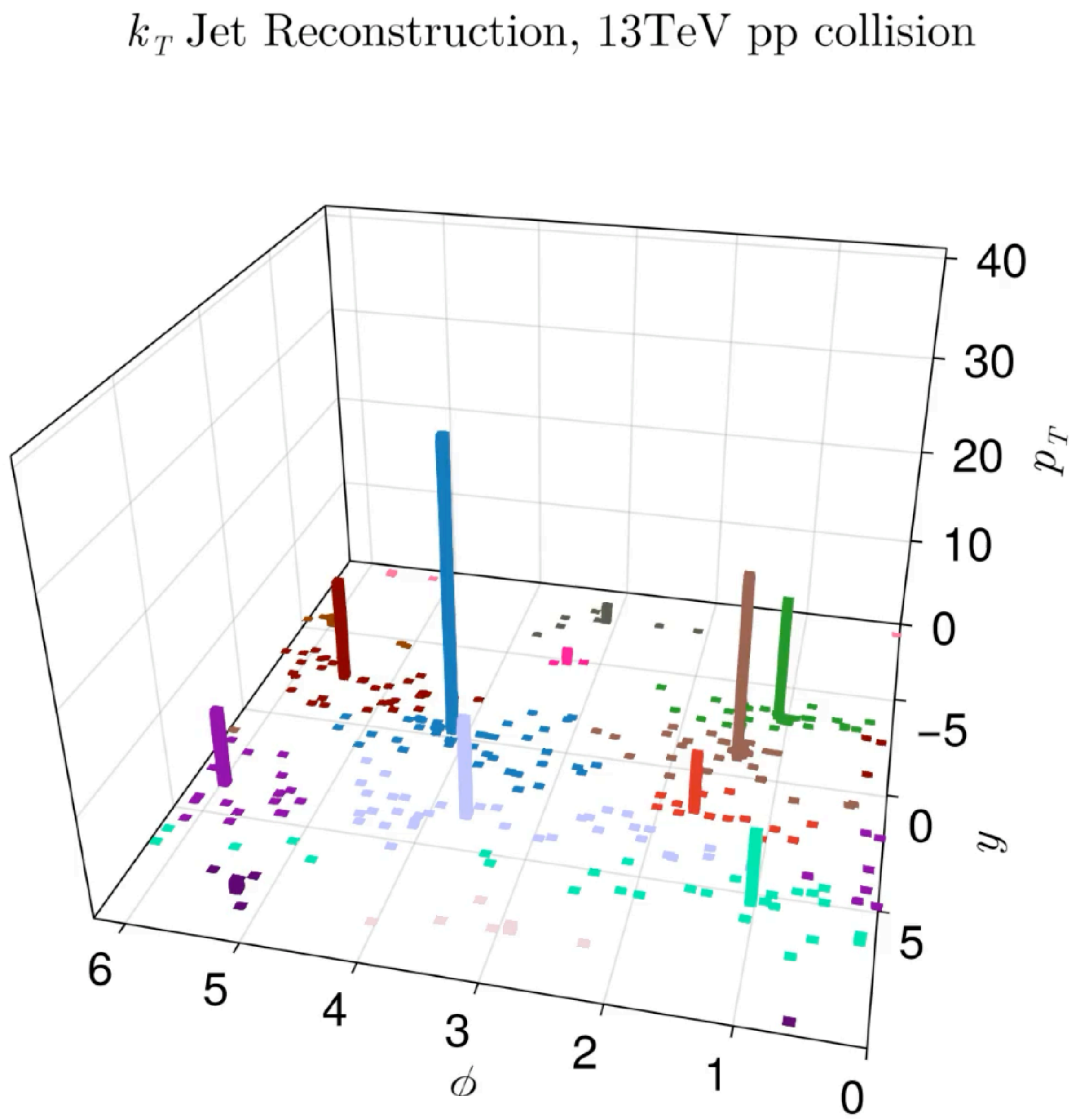


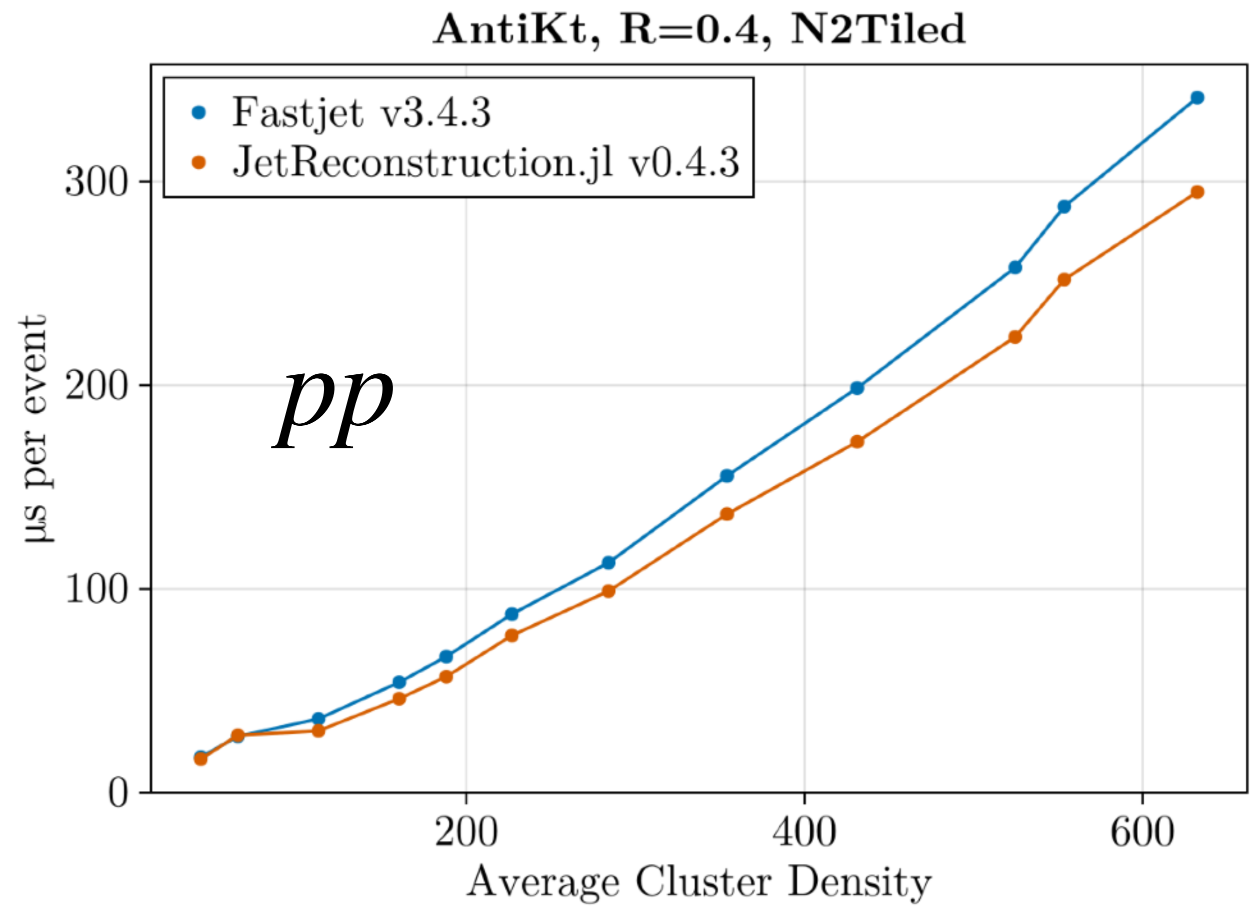

```julia
function do_plot(data::TestEm3SimData)
    (;fEdepHistos, fEdepEventHistos, fTrackLengthChHi
    lay = @layout [°; ° °]
    plot(layout=lay, show=true, size=(1400,1000))
    for (h, l) in zip(fEdepHistos, fAbsorLabel)
        plot!(subplot=1, h, title="Energy Deposition"
                 xlabel="layer #", label=l, show=true)
    end
    ...
end
```

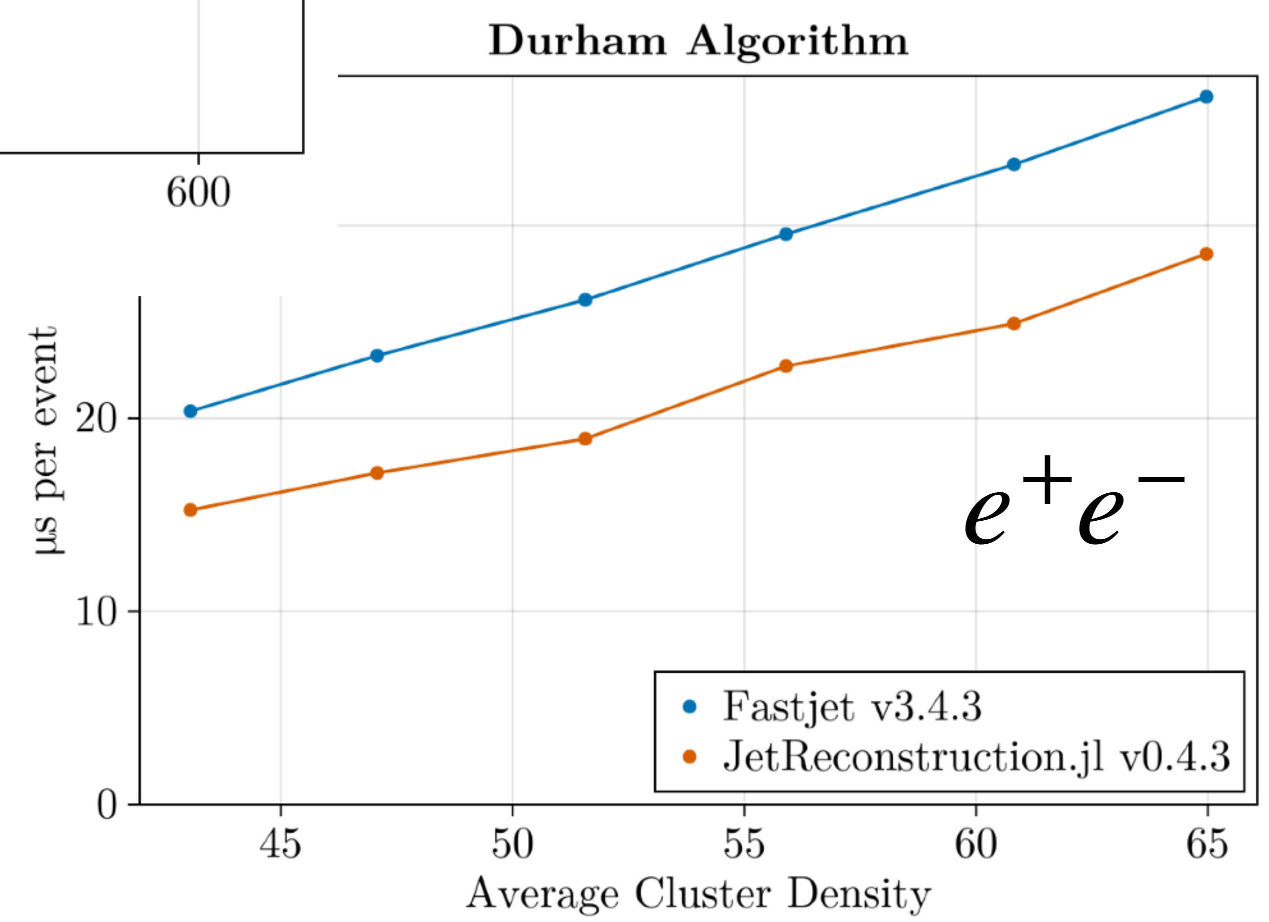


Reconstruction

- `JetReconstruction.jl` implements sequential jet reconstruction algorithms natively in Julia
- Performance is better than Fastjet
- Takes advantage of Julia compiler's native use of SIMD registers
  - Spot optimisations then accelerate minimisation finding (next pseudo jets to merge or finalise)
- Better and more flexible ergonomic interfaces
  - Easier use of experiment specific types
- Nice integration with plotting libraries
- Used to find jets in ATLAS and FCCee studies

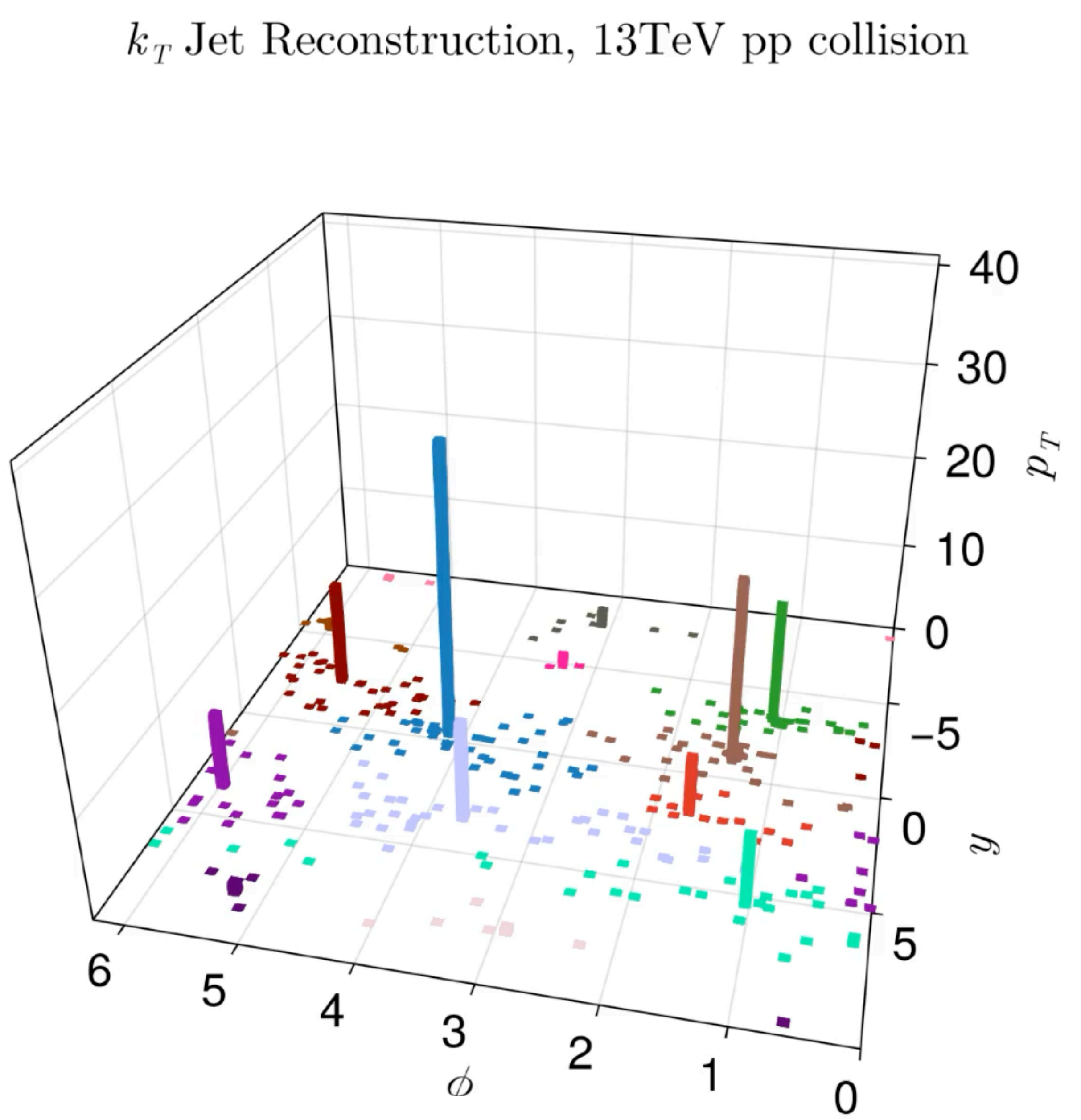


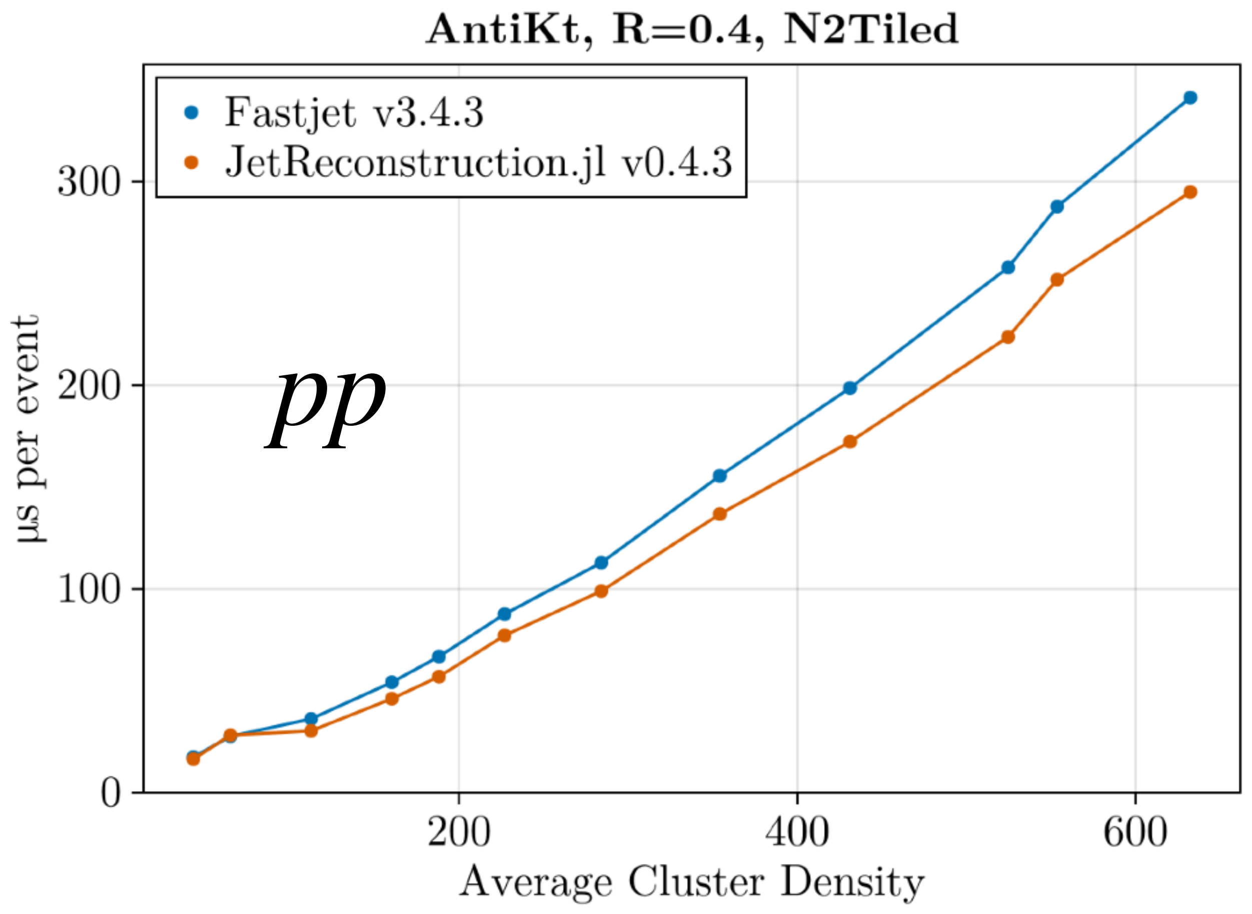


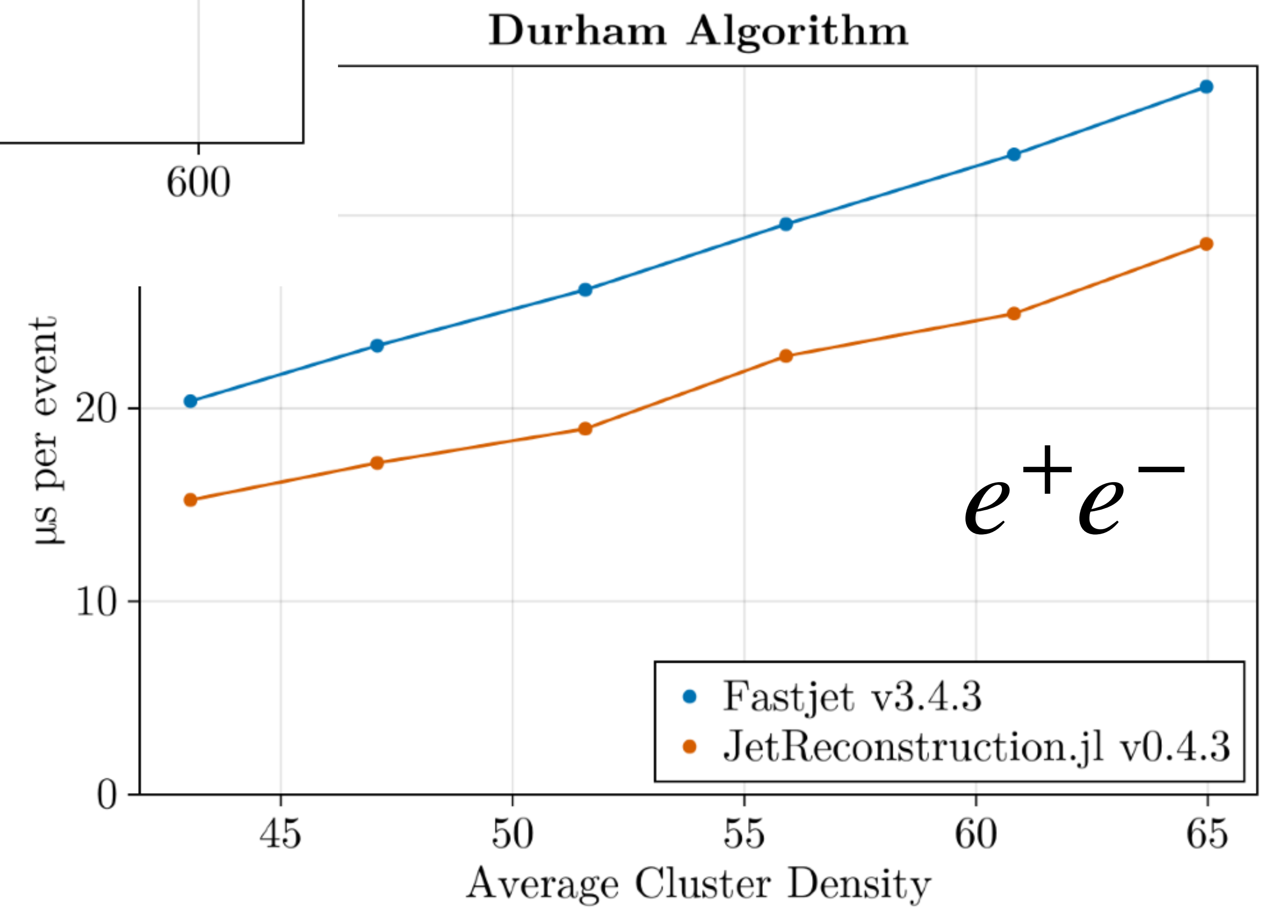


Reconstruction

- `JetReconstruction.jl` implements sequential jet reconstruction algorithms natively in Julia
- Performance is better than Fastjet
- Takes advantage of Julia compiler's native use of SIMD registers
    - Spot optimisations then accelerate minimisation finding (next pseudo jets to merge or finalise)
- Better and more flexible ergonomic interfaces
    - Easier use of experiment specific types
- Nice integration with plotting libraries
- Used to find jets in ATLAS and FCCee studies







Analysis

- Julia is a naturally productive language for analysis use
  - Close integration with plotting, statistics, numerical solvers, machine learning…
  - e.g., hemisphere mixing in ATLAS Z' analysis using `LorentzVectorsHEP.jl` and `Rotations.jl`
- Can rapidly prototype, e.g., in notebooks
  - But it's still lightning fast
- Growing ecosystem of HEP specific packages to work with four-vectors, particles, decays, lineshapes, partial wave functions and so on

The determination of the spin and parity of a vector-vector system

Liupan An[a], Ronan McNulty[b] and Mikhail Mikhasenko[c,*]

Study of the doubly charmed tetraquark T_{cc}^{+}

LHCb Collaboration*

PHYSICAL REVIEW D **98**, 096021 (2018)

Pole position of the $a_1(1260)$ from τ-decay

M. Mikhasenko,[1,*] A. Pilloni,[2,3] A. Jackura,[4,5] M. Albaladejo,[2,6] C. Fernández-Ramírez,[7] V. Mathieu,[2] J. Nys,[8] A. Rodas,[9] B. Ketzer,[1] and A. P. Szczepaniak[4,5,2]

(Joint Physics Analysis Center Collaboration)

Note on Klein-Nishina effect in strong-field QED: the case of nonlinear Compton scattering

U. Hernandez Acosta[1,2], B. Kämpfer[1,3]

PHYSICAL REVIEW D **104**, L091102 (2021)

Letter

Observation of excited Ω_c^0 baryons in $\Omega_b^- \to \Xi_c^+ K^- \pi^-$ decays

R. Aaij *et al.*[*]

(LHCb Collaboration)

Eur. Phys. J. C (2021) 81:647

https://doi.org/10.1140/epjc/s10052-021-09420-1

THE EUROPEAN PHYSICAL JOURNAL C

Check for updates

Regular Article - Theoretical Physics

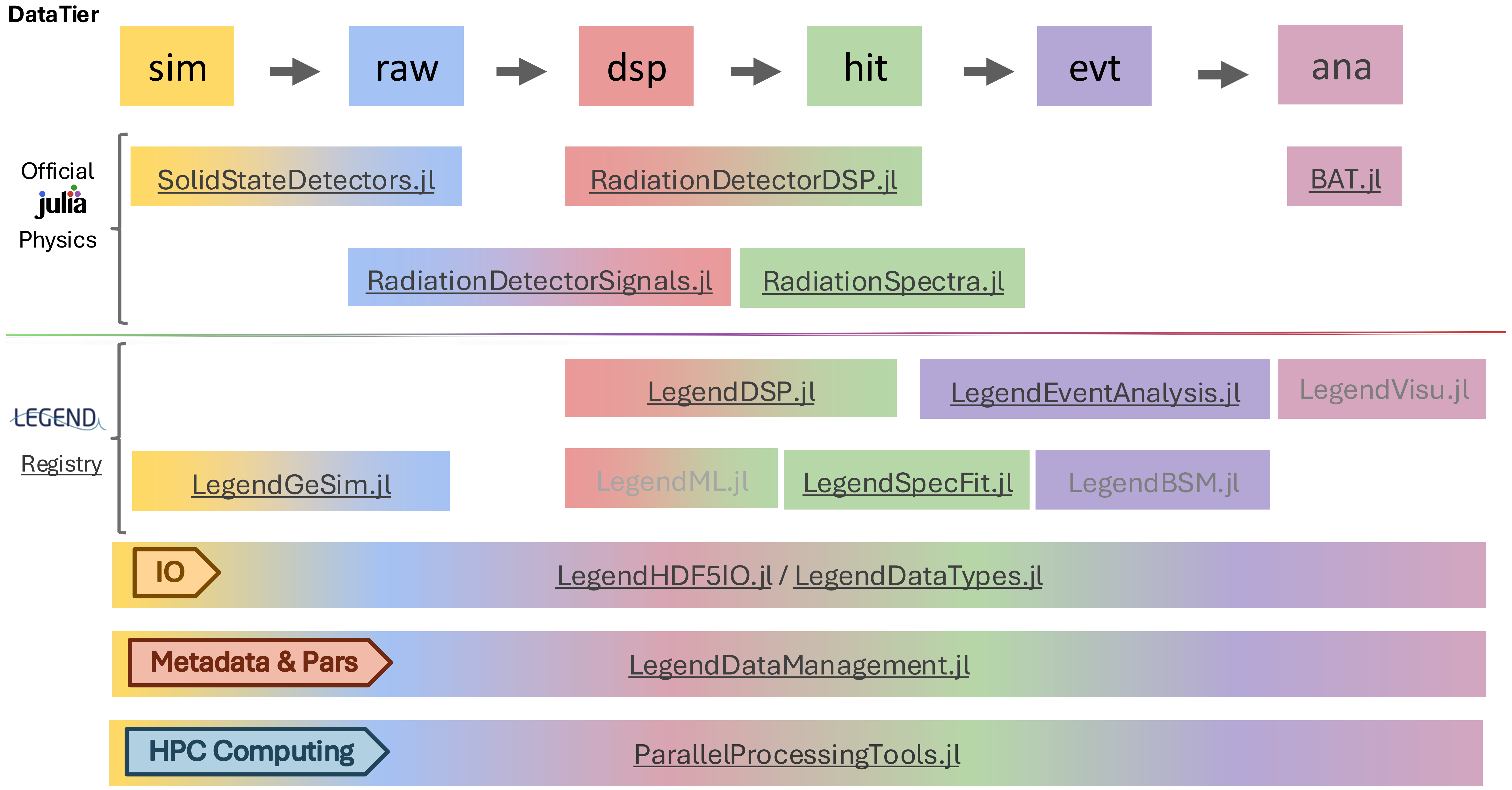
$\pi^- p \to \eta^{(\prime)} \pi^- p$ in the double-Regge region

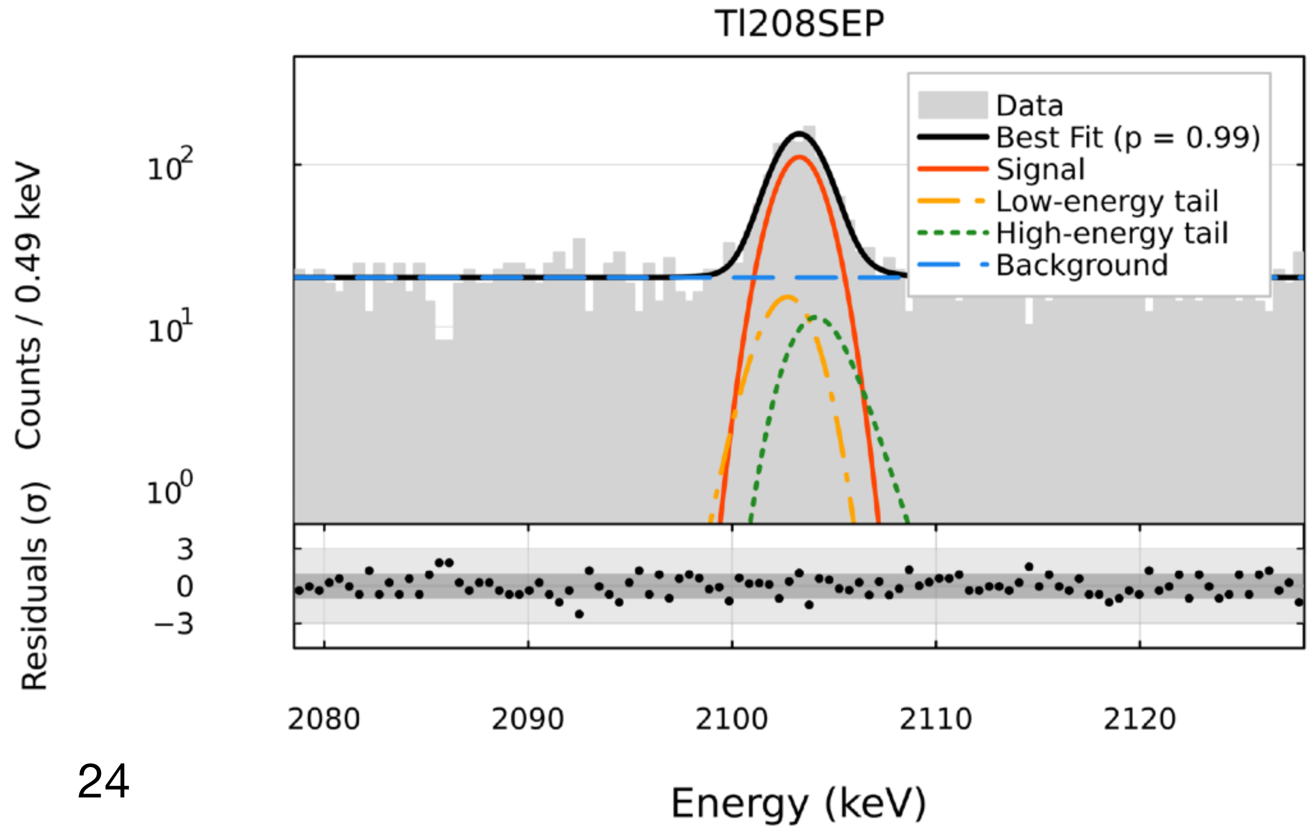
Joint Physics Analysis Center

Ł. Bibrzycki[1,2,3,a], C. Fernández-Ramírez[4,b], V. Mathieu[5,6], M. Mikhasenko[7], M. Albaladejo[3], A. N. Hiller Blin[3], A. Pilloni[8], A. P. Szczepaniak[2,3,9]

Why not run your experiment in Julia?

- Yes, you can! LEGEND experiment neutrinoless double beta decay Ge detector at Gran Sasso
- Complete secondary software stack running in Julia*
  - Simulation
  - Reconstruction
  - Analysis
- There is no part that Julia can't handle





*Used for validation against Python stack and exploring technology for upgrade to LEGEND-1000

[Plots and graphics Florian Henkes]

Conclusions

Julia's Key Features

- Easy to learn and use
- Great tooling
- Broad ecosystem with outstanding composition

→ Human productivity ✅

- Fast to execute
- Scales really well
- Support for GPUs

→ Code productivity ✅

- Integrates with existing code (in other languages)

→ Migration and adoption ✅

Julia's Key Features

- Easy to learn and use
- Great tooling
- Broad ecosystem with outstanding composition

Human p

- Fast to execute
- Scales really well
- Support for GPUs

Code p

- Integrates with existing code (in other languages)

Migration and ado

And... the compiler is still advancing with beneficial features arriving

- Improved artifact distribution, easy for CVMFS (1.11)
- Improved static compilation to callable libraries (1.12)

Julia for HEP

- Julia is the best-in-class language for scientific computing
  - And we know we need to do a lot of that in HEP
- Fast developing set of packages to add to and bridge to all that we need in HEP
  - Julia can be productive for your code **now**
- Julia has a very active and supportive user and developer community
  - Slack*, Discourse , YouTube
  - And we have the HSF JuliaHEP group as well

Potential of the Julia Programming Language for High Energy Physics Computing

Jonas Eschle[1] · Tamás Gál[2] · Mosè Giordano[3] · Philippe Gras[4] · Benedikt Hegner[5] · Lukas Heinrich[6] · Uwe Hernandez Acosta[7,8] · Stefan Kluth[6] · Jerry Ling[9] · Pere Mato[5] · Mikhail Mikhasenko[10,11] · Alexander Moreno Briceño[12] · Jim Pivarski[13] · Konstantinos Samaras-Tsakiris[5] · Oliver Schulz[6] · Graeme Andrew Stewart[5] · Jan Strube[14,15] · Vassil Vassilev[13]

HEP Software Foundation

Julia adoption would really benefit high-energy physics

There is a lot happening already - lots of scope to do even more!

*We hang out on the #HEP channel!

Julia in *CHEP*

- ROOT RNTuple implementation in Julia programming language, Monday 13h30 (Track 5, Large Hall B)
- EDM4hep.jl: Analysing EDM4hep files with Julia, Monday Poster Session (Track 5, Ground Floor Lobby)
- R&D towards heterogenous frameworks for Future Experiments, Monday 16h15 (Track 3, Room 1.A (Medium Hall A))
- Comparative efficiency of HEP codes across languages and architectures, Monday 16h33 (Track 6, Room 2.A (Seminar Room))
- Fast Jet Reconstruction in Julia, Wednesday 13h30 (Track 3, Medium Hall A)
- BAT.jl, the Bayesian Analysis Toolkit in Julia, Wednesday 17h09 (Track 5, Large Hall A)
- Navigating the Multilingual Landscape of Scientific Computing: Python, Julia, and Awkward Array, Thursday 13h30 (Track 9, Large Hall B)

Backup

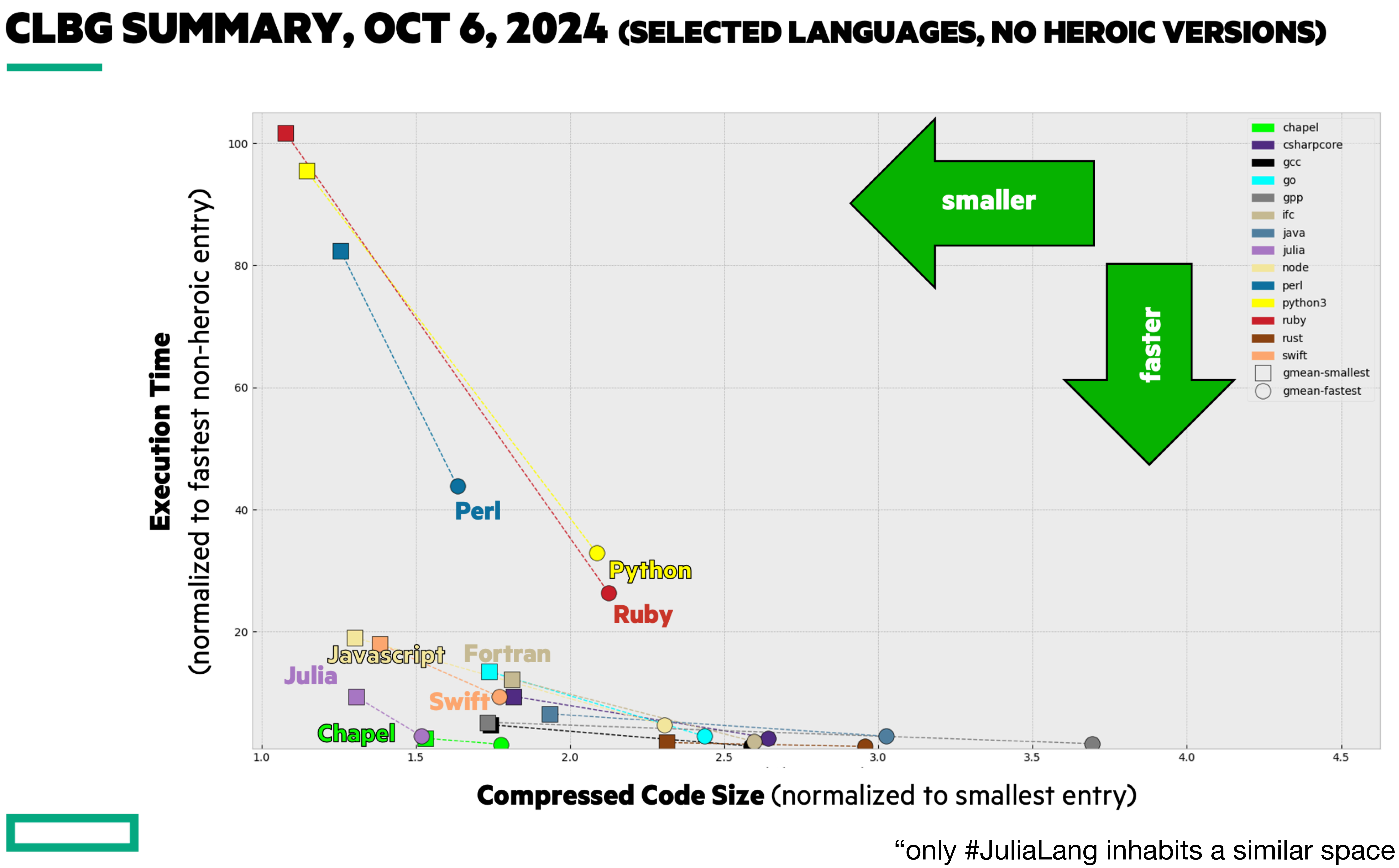
Where would Julia fit for tradeoffs?

Metric	C++	Python	Julia
Performance	✅	❌	✅
Expressiveness	⚠️	✅	✅
Learning Curve	❌	✅	✅
Safety (memory)	⚠️	✅	✅
Composability	❌	⚠️	✅

- Julia isn't perfect or magic
  - Startup time
  - Only LLVM backend
  - Static binaries and performance analysis a bit cumbersome
  - Pure Julia ML libraries not beating PyTorch
- But it does have clear advantages in many areas
- So its tradeoffs compare favourably

Computer Language Benchmarks Game

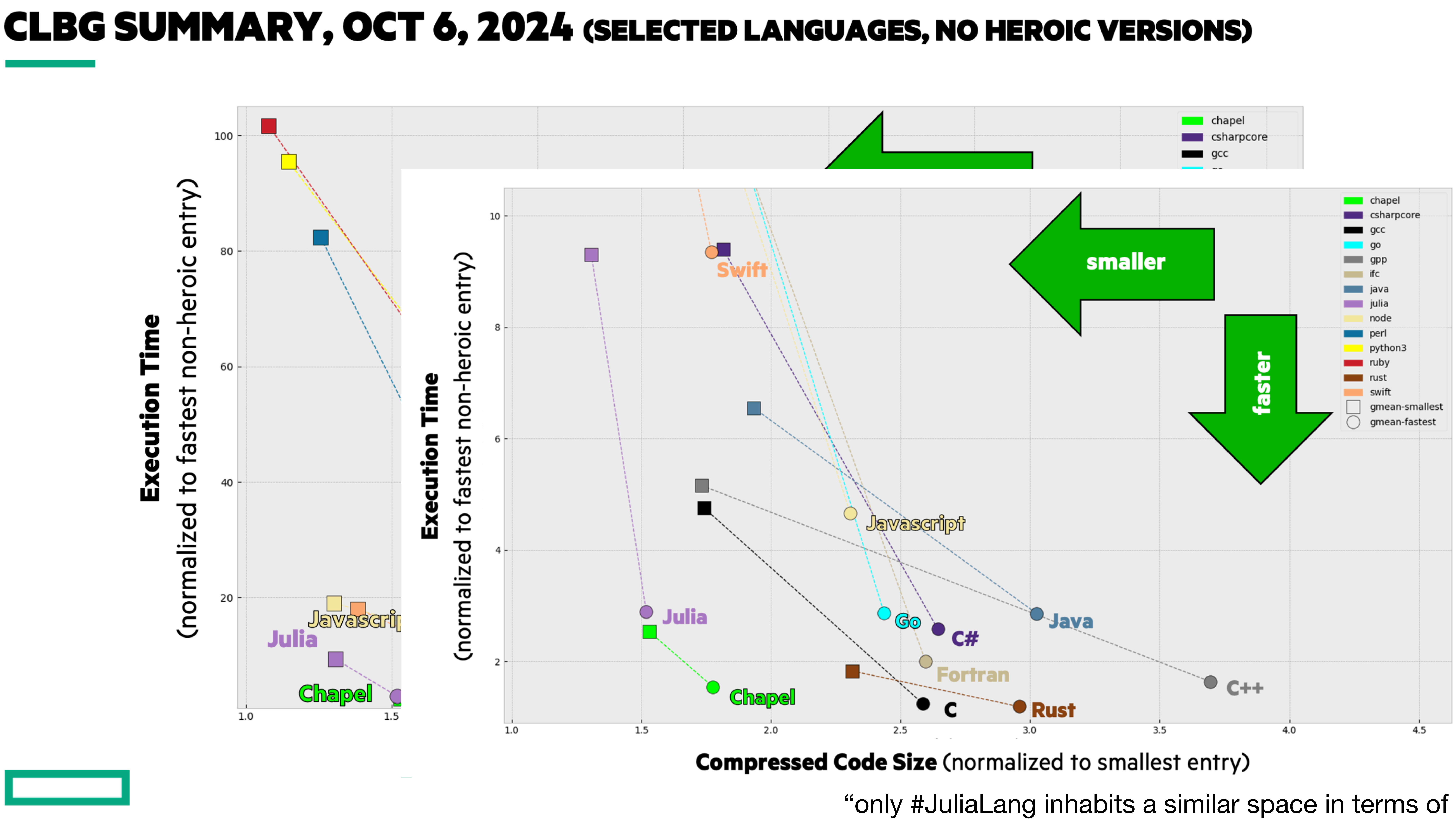
Posted by the Chapel developers



"only #JuliaLang inhabits a similar space in terms of code compactness & performance as Chapel"

Computer Language Benchmarks Game

Posted by the Chapel developers

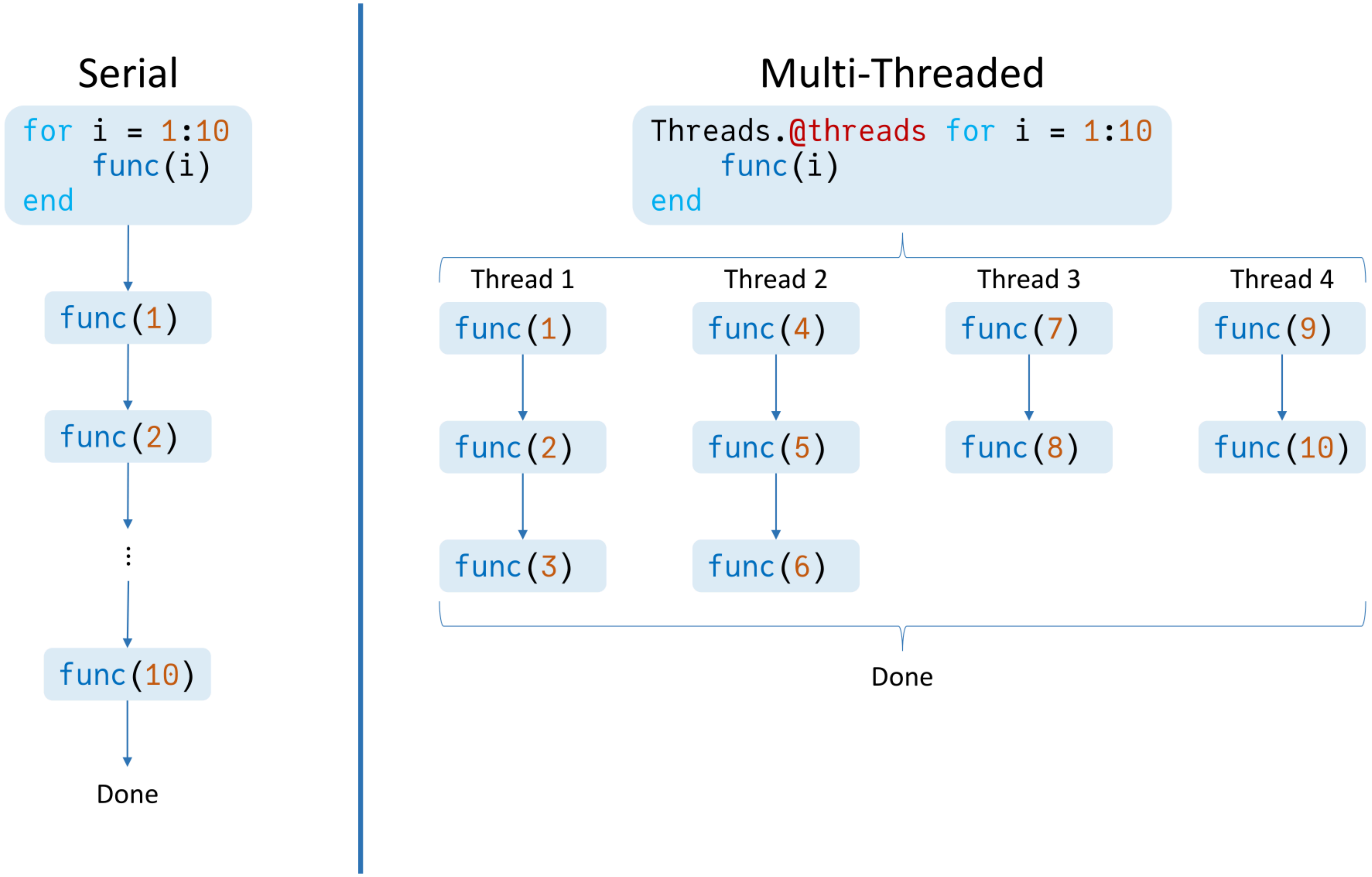


"only #JuliaLang inhabits a similar space in terms of code compactness & performance as Chapel"

Parallel computing

Native Threading support

- Support for OpenMP-like models
  - Parallelization of loops
- Support for M:N threading
  - M user threads are mapped onto N kernel threads
- Support for task migration
  - Tasks can be started, suspended, and resumed again

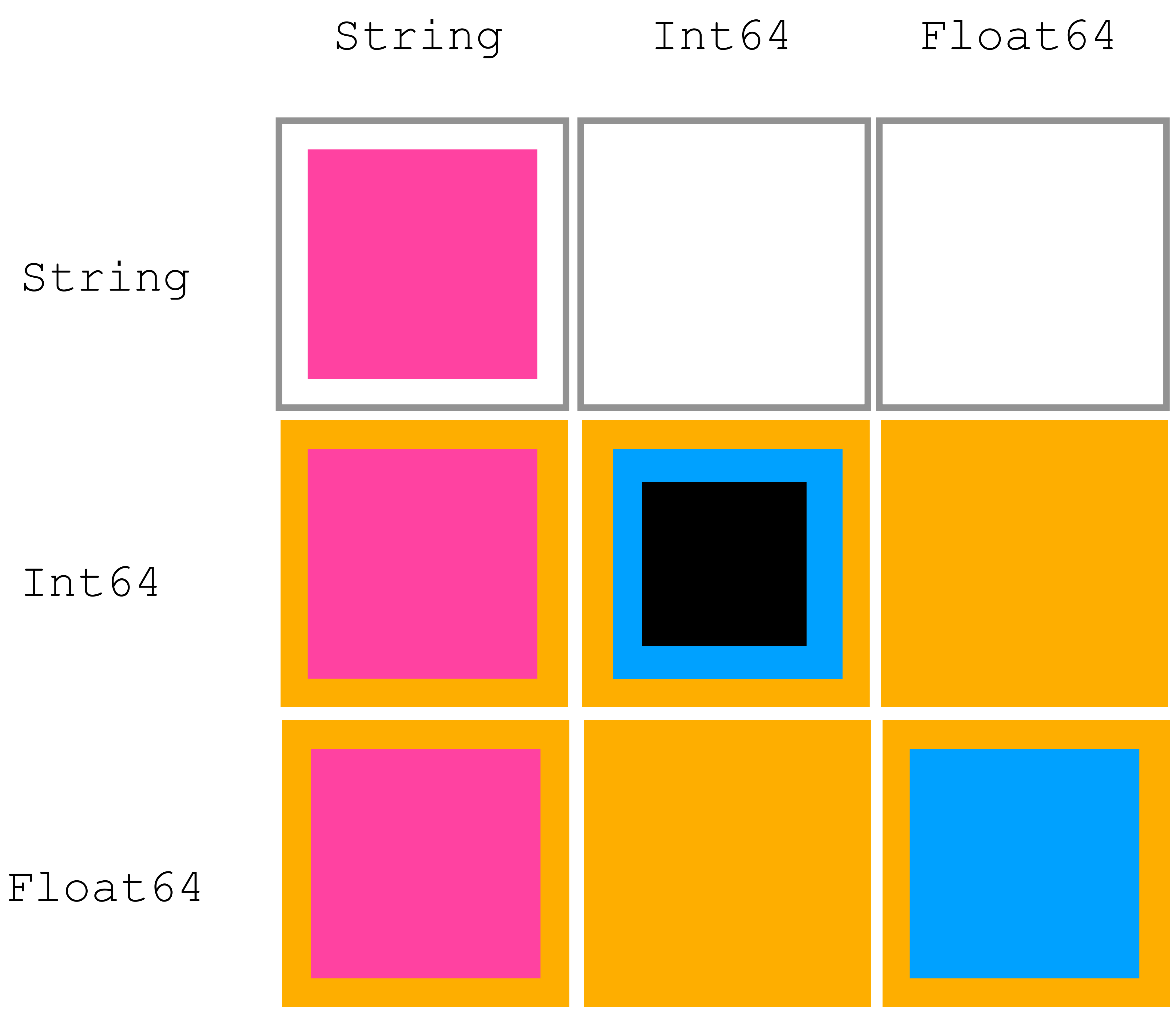


Taken from [https://blog.glcs.io/parallel-processing]

Multiple dispatch
Function and methods

`Float64<:AbstractFloat<:Real<:Number<:Any`

```
f(::Any, ::Number)
f(::T, ::T) where {T<:Number}
f(::Int64, ::Int64)
f(::String, ::Any)
```



Reproduced from [https://scientificcoder.com/the-art-of-multiple-dispatch]

Multiple dispatch II

Expressiveness

Dispatch degree	Syntax	Dispatched on	Selection power
None	`f(x,y,z)`	`{ }`	`1`
Single	`x.f(y,z)`	`{x}`	`|X|`
Multiple	`f(x::X,y::Y,z::Z)`	`{x,y,z}`	`|X|·|Y|·|Z|`

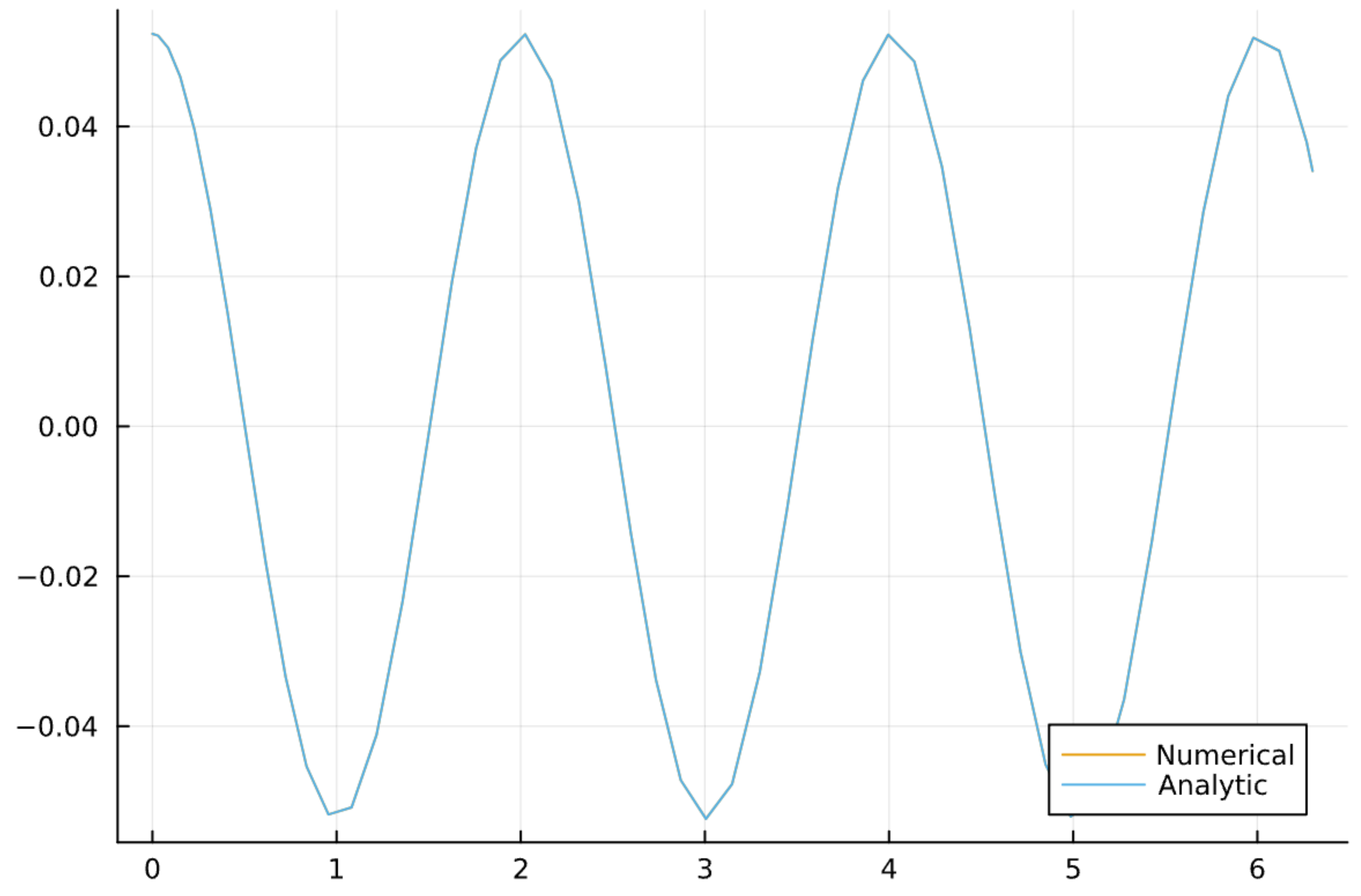
Reproduced from [S. Karpinski, “The unreasonable effectiveness of multiple dispatch”, JuliaCon2019]

Multiple dispatch III

Unreasonable effectiveness

- Allows generic code based on abstract types
- Allows arbitrary optimization
- Orthogonal development
- Solves the expression problem

```julia
using DifferentialEquations, Plots

g = 9.79          # Gravitational constants
L = 1.00          # Length of the pendulum

#Initial Conditions
u₀ = [0, π / 60]            # Initial speed and initial angle
tspan = (0.0, 6.3)

#Define the problem
function pendulum(du,u,p,t)
    θ  = u[1]
    dθ = u[2]
    du[1] = dθ
    du[2] = -(g/L)*θ
end

#Pass to solvers
prob = ODEProblem(pendulum, u₀, tspan)
sol = solve(prob, Tsit5(), reltol = 1e-6)

# Analytic solution
u = u₀[2] .* cos.(sqrt(g / L) .* sol.t)

plot(sol.t, getindex.(sol.u, 2), label = "Numerical")
plot!(sol.t, u, label = "Analytic")
```



Multiple dispatch III

Unreasonable effectiveness

- Allows generic code based on abstract types
- Allows arbitrary optimization
- Orthogonal development
- Solves the expression problem

```julia
using DifferentialEquations, Measurements, Plots

g = 9.79 ± 0.02; # Gravitational constants
L = 1.00 ± 0.01; # Length of the pendulum

#Initial Conditions
u₀ = [0 ± 0, π / 60 ± 0.01] # Initial speed and initial angle
tspan = (0.0, 6.3)

#Define the problem
function pendulum(du,u,p,t)
    θ  = u[1]
    dθ = u[2]
    du[1] = dθ
    du[2] = -(g/L)*θ
end

#Pass to solvers
prob = ODEProblem(pendulum, u₀, tspan)
sol = solve(prob, Tsit5(), reltol = 1e-6)

# Analytic solution
u = u₀[2] .* cos.(sqrt(g / L) .* sol.t)

plot(sol.t, getindex.(sol.u, 2), label = "Numerical")
plot!(sol.t, u, label = "Analytic")
```