VIEWS24 Workshop

Geant4.jl - Particle Transportation in Julia

Pere Mato / CERN
27 April 2024

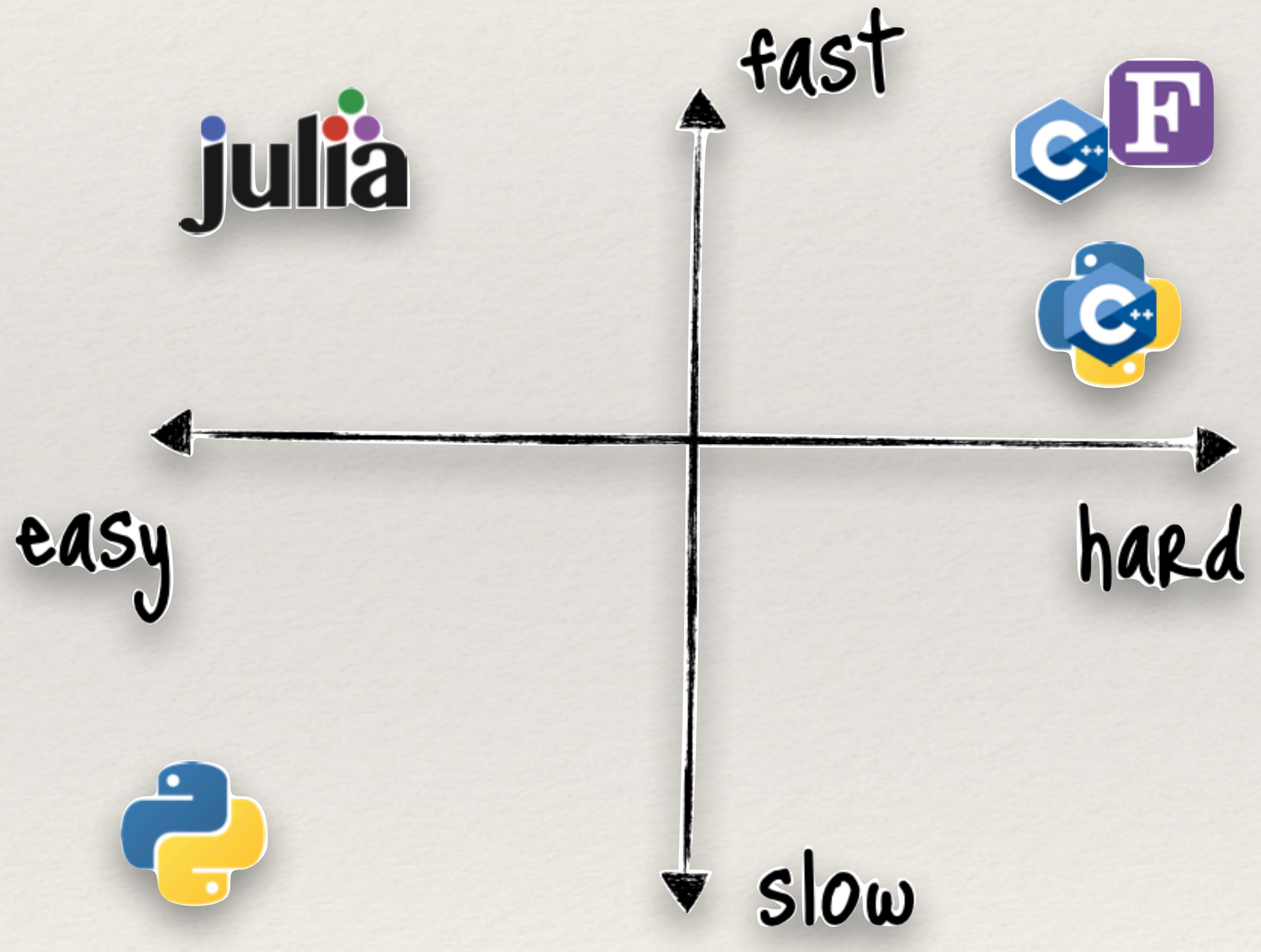
https://github.com/JuliaHEP/Geant4.jl

Why a new programming language?

- We a need a solution for the **Two Language Problem**
  - **C++** is fast but complex (and every day becoming more complex)
  - **Python** is nice and easy but very slow (mitigated if you avoid loops)
- The community has developed ways to deal with these two languages but we pay a price
  - Interoperability is not always smooth (e.g. garbage collection side effects)
  - Awkward constructions (e.g. the C++ strings in the PyRDF)

Why Julia?

- The Julia language was launched in 2012 (v1.0 in 2018) - New, but not immature!
- Modern imperative language, multi-paradigm with reflection and object orientation
- Robust **built-in tooling** (learning from earlier languages)
  - Outstanding integrated **package manager** and build system
  - Module system with excellent **code reuse**
  - Modern tooling, with built in **debuggers** and **profilers**
  - **Interactive** - REPL and full notebook support (it's the "Ju" in Jupyter)
- Julia has been built from the ground up to be **very fast**
  - JIT compilation via LLVM to native machine code
  - Performance is comparable to C and C++ (as a baseline, see microbenchmarks)



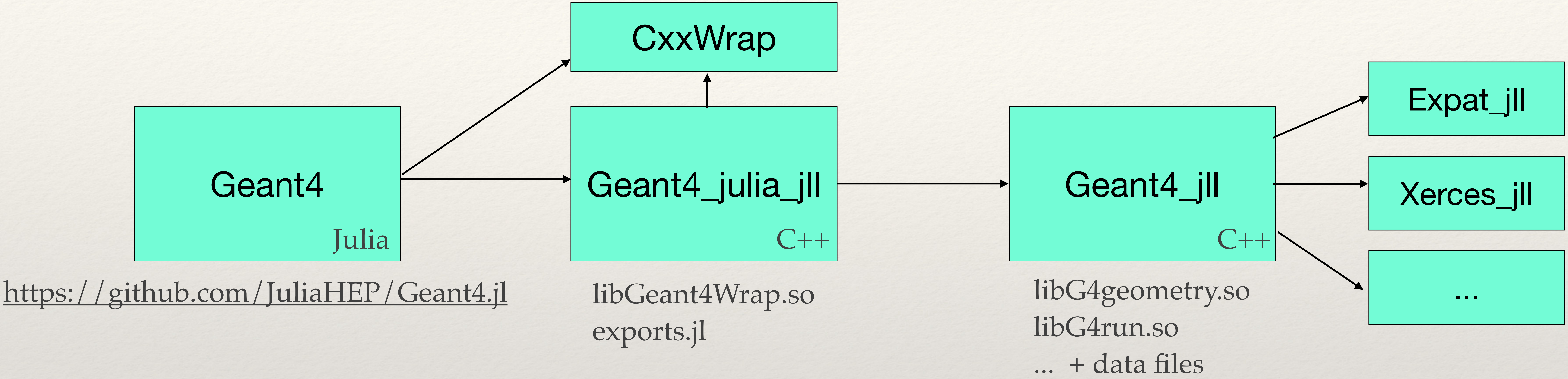
Julia Wrappers to Geant4

- Similarly to Python, to call C++ from Julia you need to write (better generate) wrappers for each method you want to offer to Julia
- Using the **CxxWrap.jl** package
  - The user needs to write small code (in C++) to wrap each class and method (similar to pybind11 or Boost.Python)
- The package **WrapIt** developed by Ph. Gras makes use of LLVM libraries to generate the wrappers automatically 😀

```
Generated wrapper statistics
 enums:              28
 classes/structs:    209
   templates:        0
   others:           209
 class methods:      2846
 field accessors:    19 getters and 19 setters
 global variable accessors: 10 getters and 0 setters
 global functions: 53
```

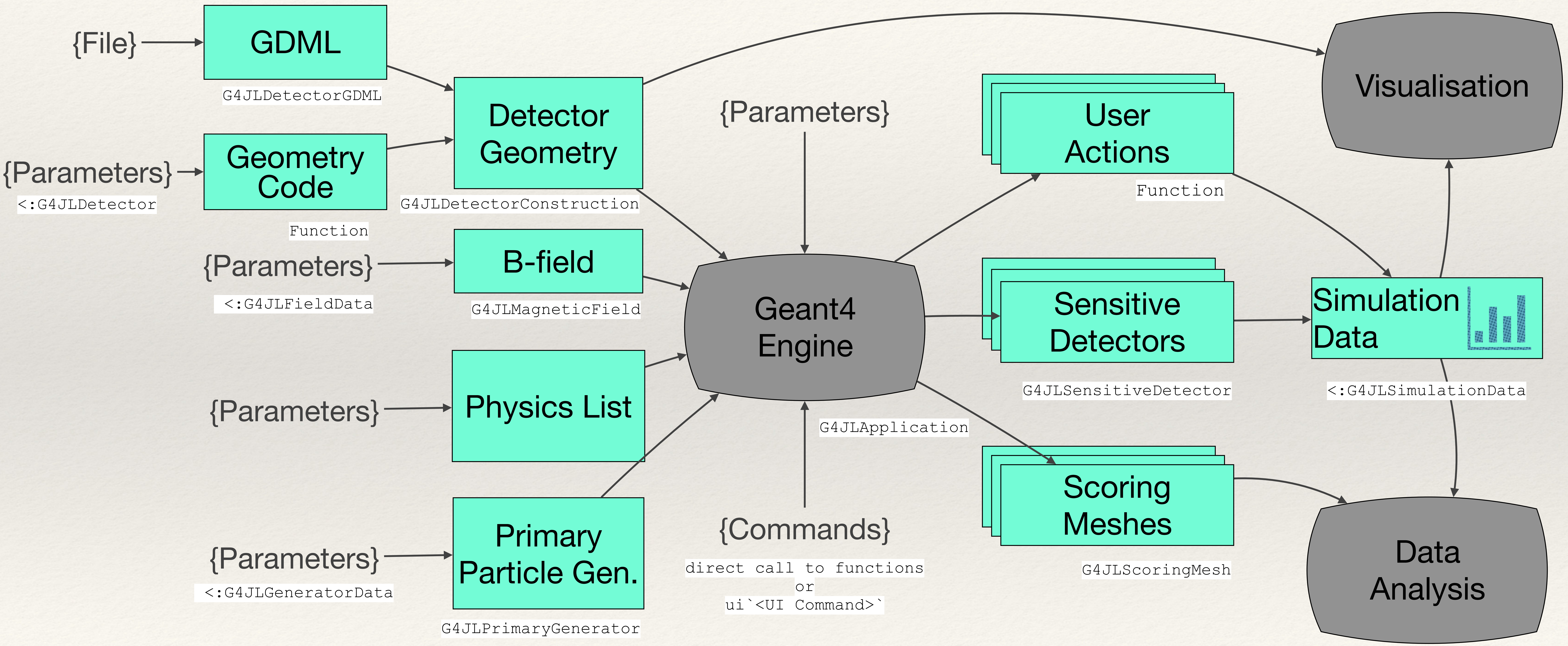
  - It helps enormously to ensure sustainability (e.g. tracking G4 versions)

Package Structure



- The package **Geant4.jl is a pure Julia package** (platform independent)
- The binary libraries (platform dependent) for Geant4 and the wrapper library are downloadable artifacts of Julia **_jll packages** produced by the **BinaryBuilder** package, and stored at the Julia infrastructure (GitHub)
- Binaries available for **Linux, MacOS and Windows,** and all common hardware architectures

Rethinking the Application Interface



Application Interface: Wish List

- The idea is to exploit the Julia language to provide a simple and ergonomic user interface
    - **Minimalistic**. Define only what you really need for the simulation application. Avoid any boilerplate code.
    - **Do the necessary at the right time**. Hide the application state and calling sequence
    - **Interactive**. Using the Julia REPL, as well as support for Jupyter and Pluto notebooks
    - **Transparent MT**. As much as possible hide behind the scenes, the handling of Multi-Threading (e.g. per-thread calls and thread-local instances)
    - **Integrated simulation and analysis**. In the same application the simulation data can be analyzed and presented

Callbacks

- "User custom code" are callbacks in the G4 toolkit
  - E.g. detector constructor, user actions and sensitive detectors
  - Typically by inheriting from a virtual base classes (e.g. `G4UserSteppingAction`, `G4VSensitiveDetector`)
- CxxWrap.jl provides a convenient way to call Julia from C++
  - The callbacks are therefore "normal" Julia functions

```julia
#---Step action------------------------------------------
function stepaction(step::G4Step, app::G4JLApplication)::Nothing
    data = getSIMdata(app)
    prepoint = GetPreStepPoint(step)
    track = GetTrack(step)

    ...

    nothing
end
```

{Julia}

Multi-threading

- Geant4 can run multi-threading by distributing the simulation of events on a C++ thread pool managed by the toolkit
  - Trivial parallelization. Very good scaling!
- **MT is enabled in Geant.jl just with an argument to set number of threads**
- User actions, sensitive detector code, etc. will be run naturally on different threads
  - To avoid race conditions, better if each thread updates its own copy of the data
  - Data is cloned for each thread and summed (reduced) at the end of the run

Simulation Data

- With the ‘user actions’ and ‘sensitive detectors’ the user will collect all simulation data in a user defined struct inheriting from `G4JLSimulationData`
    - Typically it will consists of counters, histograms, temporary structs to be written step-by-step or event-by-event, etc.
- In case of MT, a function (`add!`) to reduce the contents of the data struct for each “worker thread” needs to be provided by the user

```
#---Simulation Data struct------------------------------------------------
mutable struct TestEm3SimData <: G4JLSimulationData
    #---Run data---------------------------------------------------------
    fParticle::CxxPtr{G4ParticleDefinition}
    fEkin::Float64

    fChargedStep::Int32
    fNeutralStep::Int32

    fN_gamma::Int32
    fN_elec::Int32
    fN_pos::Int32

    fEnergyDeposit::Vector{Float64}     # Energy deposit per event
    fTrackLengthCh::Vector{Float64}     # Track length per event

    fEdepEventHistos::Vector{Hist1D}
    fTrackLengthChHistos::Vector{Hist1D}
    fEdepHistos::Vector{Hist1D}
    fAbsorLabel::Vector{String}

    TestEm3SimData() = new()
end

#---add function---------------------------------------------------------
function add!(x::TestEm3SimData, y::TestEm3SimData)
    x.fChargedStep += y.fChargedStep
    x.fNeutralStep += y.fNeutralStep
    x.fN_gamma += y.fN_gamma
    x.fN_elec += y.fN_elec
    x.fN_pos += y.fN_pos
    x.fEdepEventHistos += y.fEdepEventHistos
    x.fTrackLengthChHistos += y.fTrackLengthChHistos
    x.fEdepHistos += y.fEdepHistos
end
```

counters

array of histograms

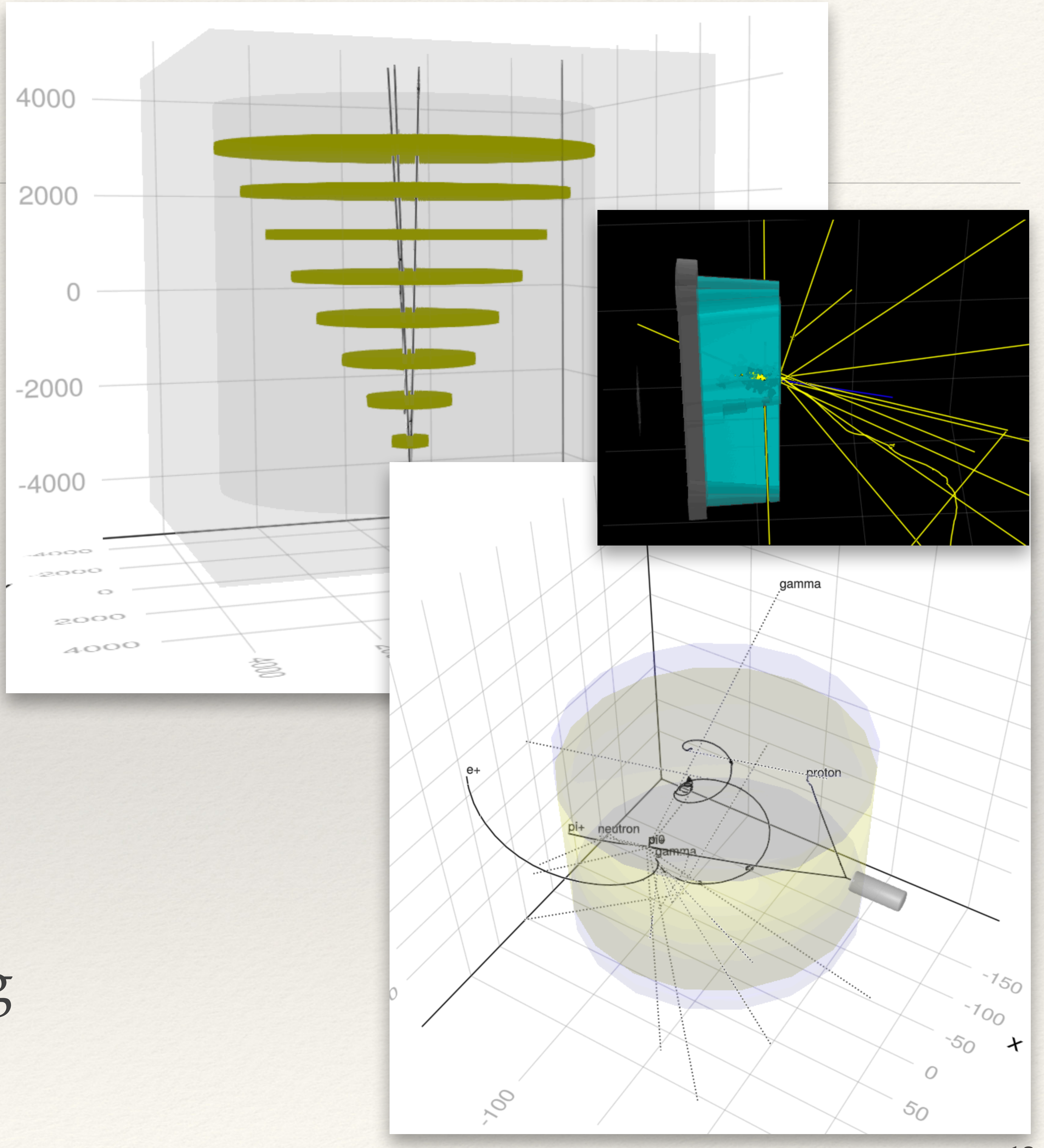
Simulation Application

- The user can create a `G4JLApplication` with all the elements of the simulation application (detector geometry, primary generator, physics list, user actions, etc.)
- Geant4 requires a strict order of instantiation/configuration/initialization and this is guaranteed by Geant4.jl interface
- In case `nthreads` > 0 (default) the `G4MTRunManager` is instantiated and simulation data as well as sensitive detector data is replicated N times

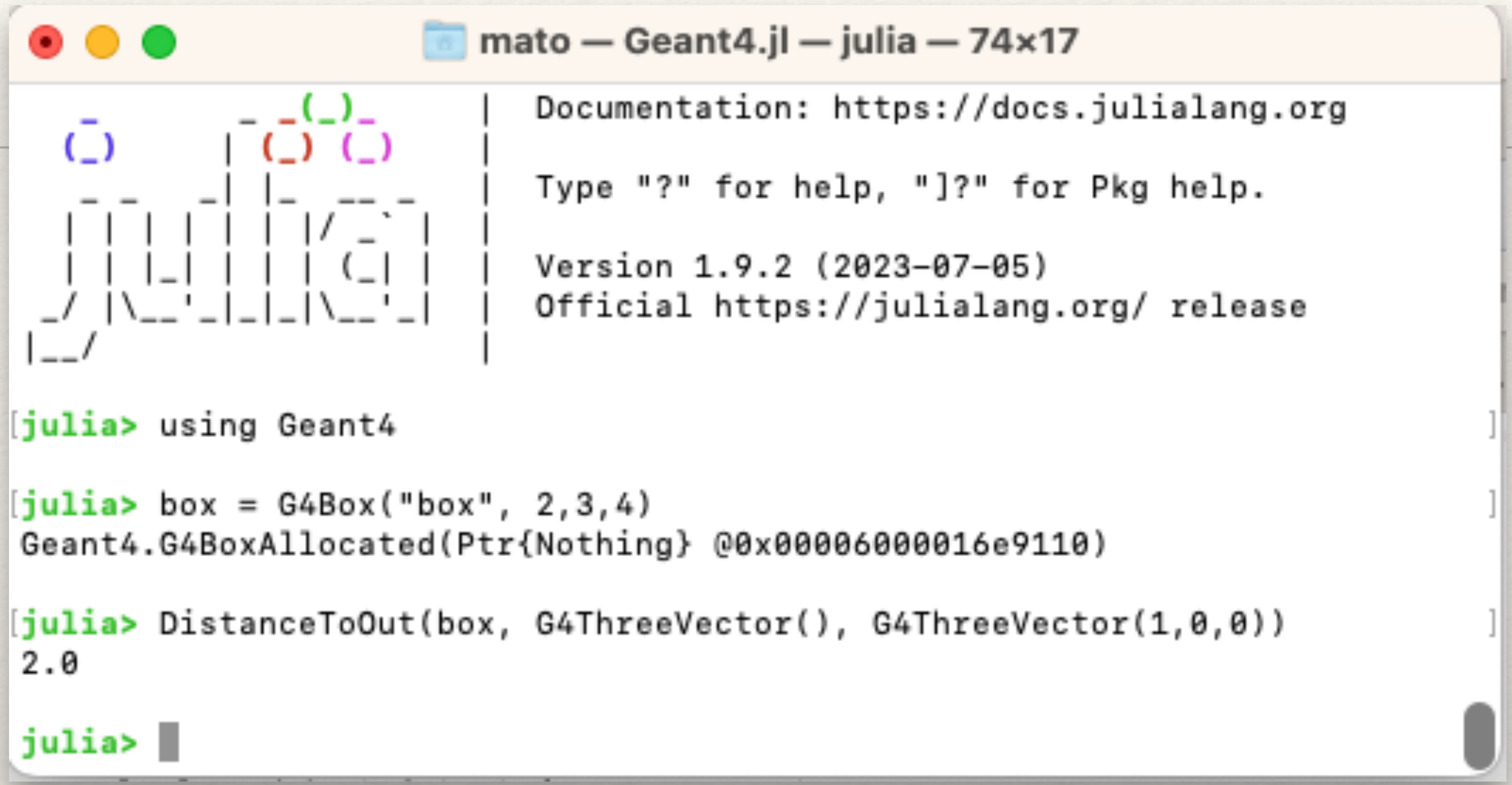
```
#---Create the Application------------------------------------------------------------------------------------
app = G4JLApplication(;detector = B2aDetector(nChambers=5),          # detector with parameters
                       physics_type = FTFP_BERT,                      # what physics list to instantiate
                       generator = G4JLParticleGun(...),              # primary particles generator
                       nthreads = 8,                                  # number of worker threads (>0 == MT)
                       endeventaction_method = endeventaction,        # end event action
                       sdetectors = ["Chamber_LV+" => chamber_SD]     # mapping of LVs to SDs (+ means multiple LVs)
                      )

#---Configure, Initialize and Run----------------------------------------------------------------------------
configure(app)
initialize(app)
beamOn(app, 1000)
```

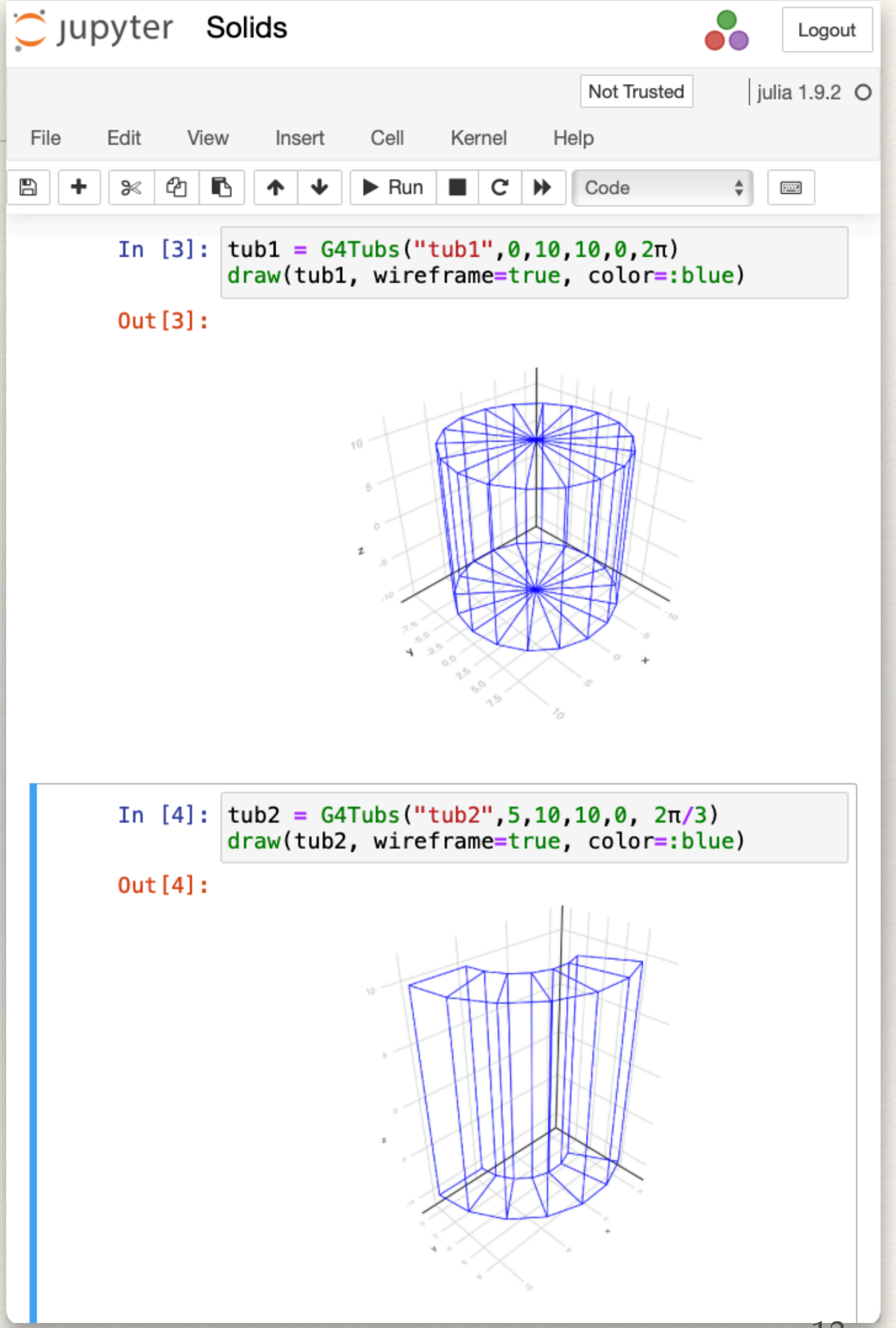
Visualization

- Implemented basic visualisation of the geometry and tracks using Makie.jl package
    - including boolean solids
    - easy for users to customize and draw basically anything
- Interactive
    - Very useful for building and debugging the application



Interactivity





- Julia comes with a powerful and modern REPL (Read-Eval-Print Loop)
  - history, line completion, help, etc.
- Very good support for notebooks (Jupyter, Pluto)
  - see examples in Geant4.jl documentation
- Both are very well integrated in IDEs such as VS Code

Performance

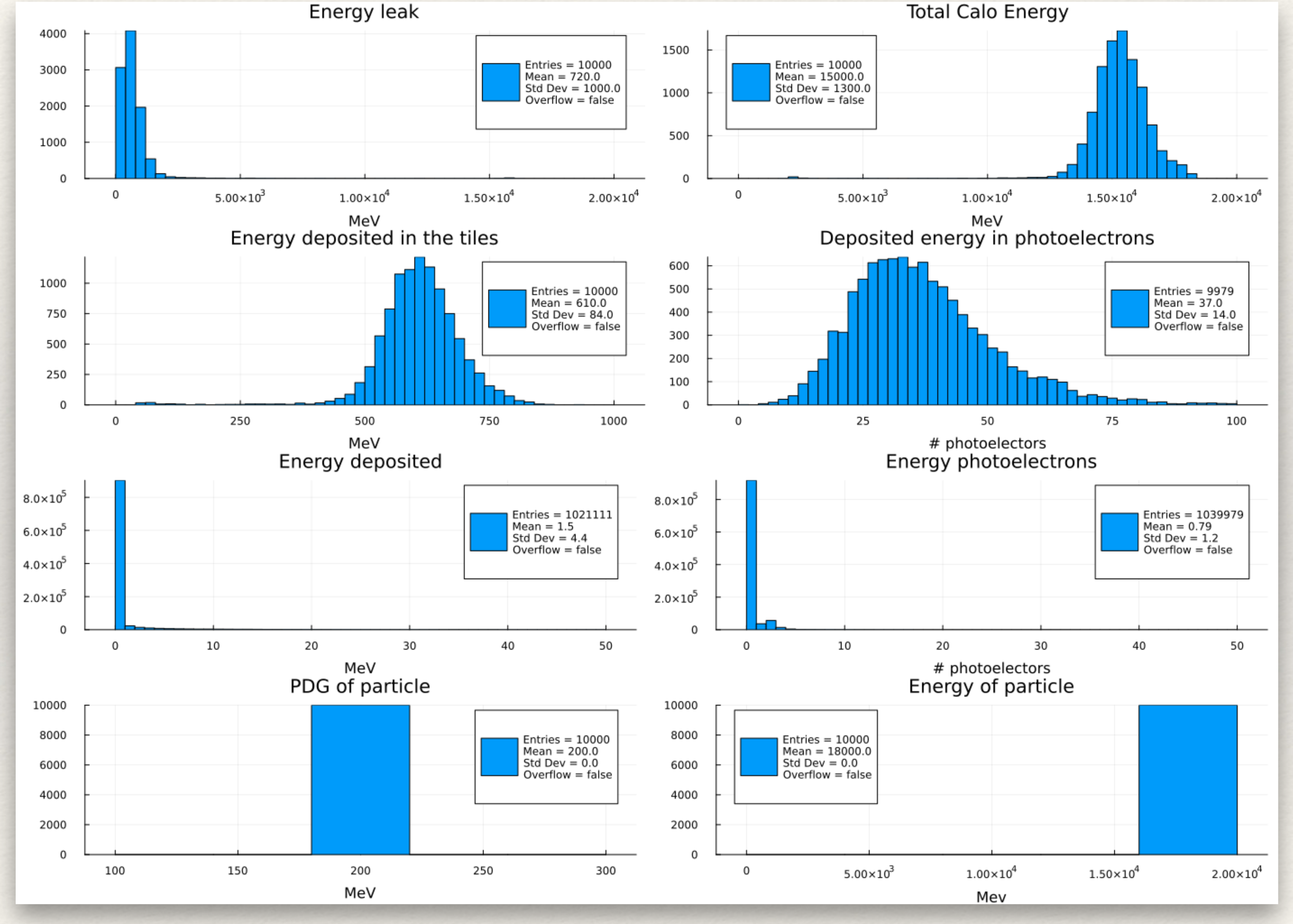
- Performance should be equivalent to the C++ application
- Julia user actions (callbacks from C++ to Julia) do not add any significant overhead and can be executed very efficiently
    - JIT and with less abstraction layers
- Julia suffers from a larger startup time (final type inference and JIT compilation)
    - big improvement since Julia version 1.9

	B2a (C++)	B2a.jl
events = 1	0.9 s	6 s
events =100k	106 s	109 s
events =100k (MT)	23 s	27 s

- Simple benchmark of B2a example
    - with protons @ 3 GeV
    - running on a Mac-mini with the M1 processor (8 cores = 4 performance and 4 efficiency)
- C++ and Julia are basically identical taking the initial overhead (serial) into account

Complete and Realistic Examples

- The package Geant4.jl comes with a number of examples
- Added ATLTileCalTB.jl converting L. Pezzotti's C++ example to validate G4 with the ATLAS TileCal test beam data
    - Sensitive detectors, user actions, signal processing, plotting results, detector and event visualisation
    - ~3000 lines (C++) versus ~1000 lines (Julia),
    - 2000 pi+ @ 18 GeV: 143 s (C++) versus 104 s (Julia)



Conclusions

- Programming in Julia is really fun
    - The built-in tooling and ecosystem is very complete. Great integration with VSCode
    - Software re-use capabilities really excel compared to C++
- **Geant4.jl** is an extremely useful add-on to the Geant4 project
    - Tutorials (very easy to setup and portable), interactive development (notebooks), connection to other powerful packages in the Julia ecosystem (visualisation, data analysis, etc.)
- Geant4.jl is in a **working state,** missing functionality can be added easily

Trying Out

- Developed a tutorial as a set of Jupyter notebooks
  - Step-by-step building a simulation application
  - Includes a number of complete examples
- You can either browse the rendered Jupyter Book or follow the instructions to run yourself the notebooks on your computer