2023 JuliaHEP Workshop

Geant4.jl - New Interface to Simulation Applications

Pere Mato / CERN
9 November 2023

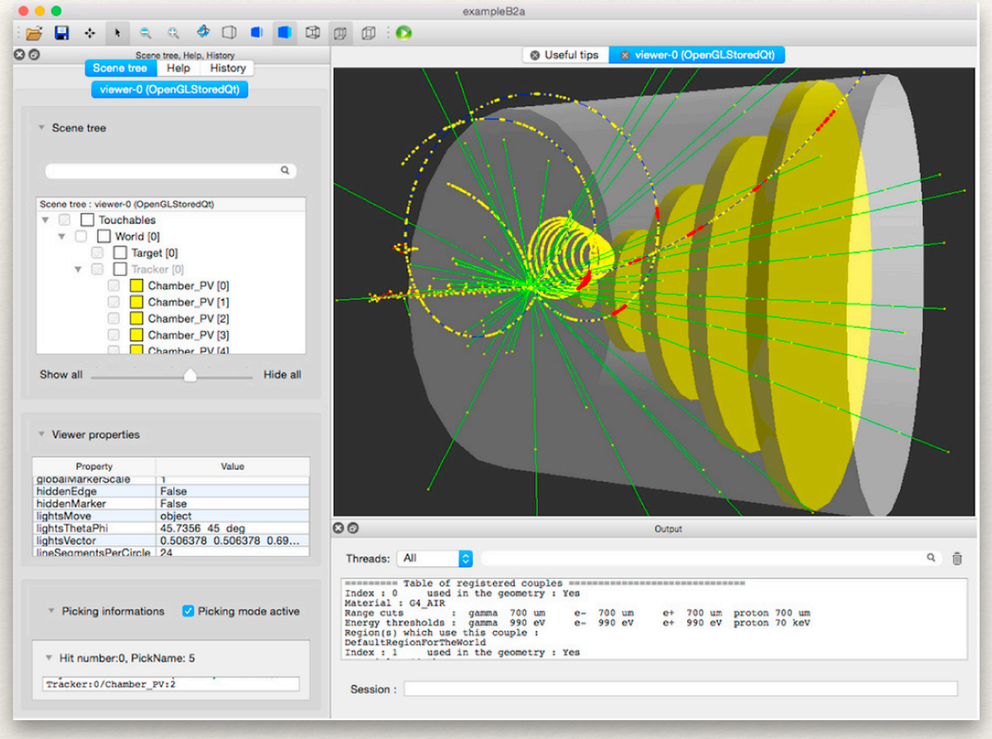
https://github.com/JuliaHEP/Geant4.jl

Motivation

- **Julia is a priori a good programming language candidate for HEP**
  - It combines **high-level expressibility** for scientific computational problems together with **high-performance** execution, avoiding the **two language problem**
- One essential aspect is to **evaluate its interoperability with existing C++ libraries** in HEP
- An excellent case for this evaluation is to use Geant4
  - It is large, complex, and not easy to re-write
  - The result of this evaluation is a ‘nice-to-have’ new functionality for Geant4

Geant4

- **Geant4** is is a software toolkit for the simulation of the passage of elementary particles through matter
  - Primarily used in the fields of high-energy physics, nuclear physics, medical physics, space science, and other areas
- Developed by the Geant4 Collaboration
  - Open source license, over 2 MLOC of C++, about 25 years



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

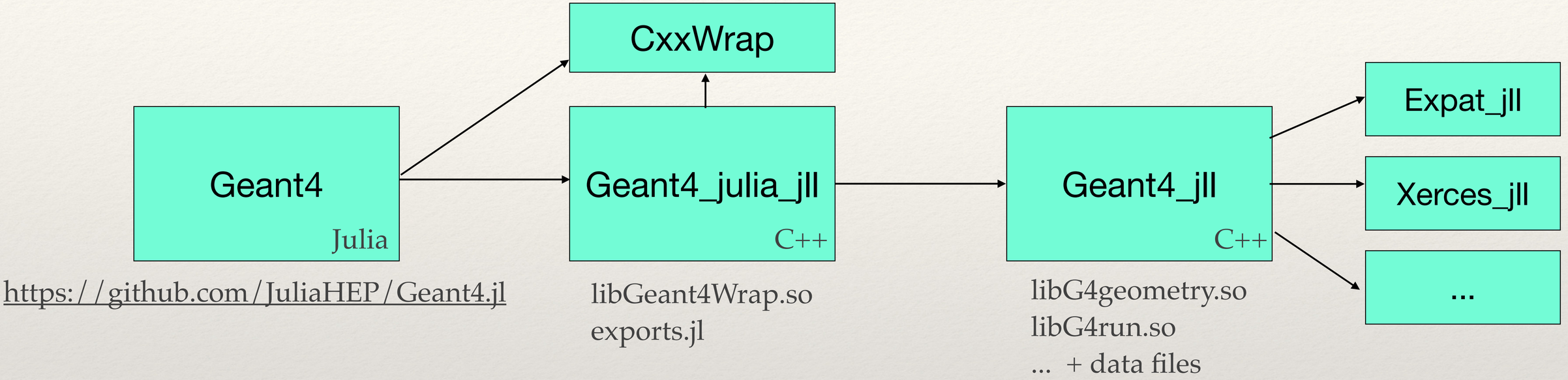
```
Generated wrapper statistics
 enums:              28
 classes/structs:    209
   templates:        0
   others:           209
 class methods:      2846
 field accessors:    19 getters and 19 setters
 global variable accessors: 10 getters and 0 setters
 global functions: 53
```

Basic Interface

- The interaction to Geant4 from Julia needs to be adapted to the native Julia language
  - E.g., only structs, no real inheritance, no class methods
- Since all classes start with ‘G4’ we can `export` all types when issuing `using Geant4`
  - This is very good, no clashes
- Sometimes native Julia types are not obtained directly and require some extra step
  - e.g. G4String

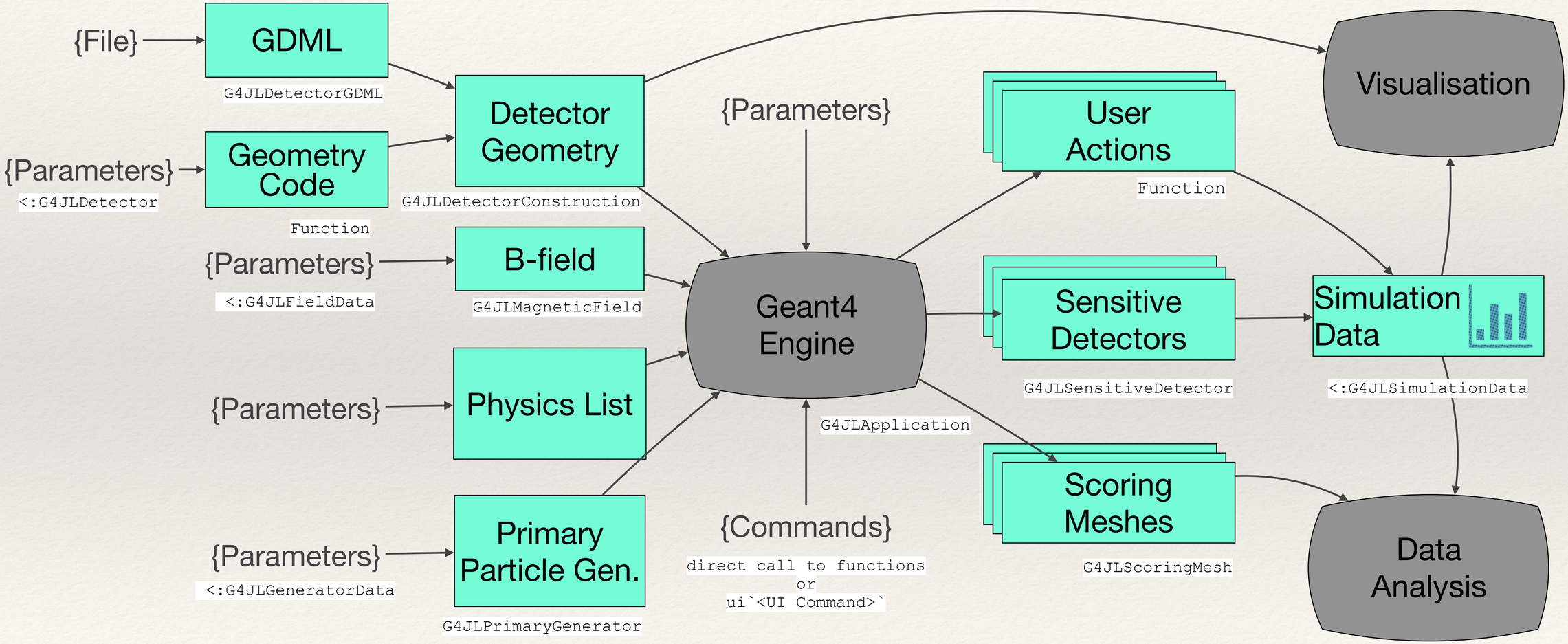
```
julia> using Geant4
julia> runManager = G4RunManager()
*************************************************************
 Geant4 version Name: geant4-11-01-patch-01 [MT]   (10-February-2023)
                       Copyright : Geant4 Collaboration
                      References : NIM A 506 (2003), 250-303
                                 : IEEE-TNS 53 (2006), 270-278
                                 : NIM A 835 (2016), 186-225
                             WWW : http://geant4.org/
*************************************************************

Geant4.G4RunManagerAllocated(Ptr{Nothing} @0x00007f9fcb6f9c50)

julia> methodswith(G4RunManager, supertypes=true)
[1] convert(t::Type{G4RunManager}, x::T) where T<:G4RunManager in Geant4 at ...
[2] AbortEvent(arg1::Union{CxxWrap.CxxWrapCore.CxxRef{<:G4RunManager},
...
[94] rndmSaveThisRun(arg1::Union{CxxWrap.CxxWrapCore.CxxRef{<:G4RunManager}, ...

julia> v = GetVersionString(runManager)
ConstCxxRef{G4String}(Ptr{G4String} @0x00007ffed34df2d8)

julia> String(v)
" Geant4 version Name: geant4-11-01-patch-01 [MT]   (10-February-2023)"
```

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)

Rethinking the Application Interface

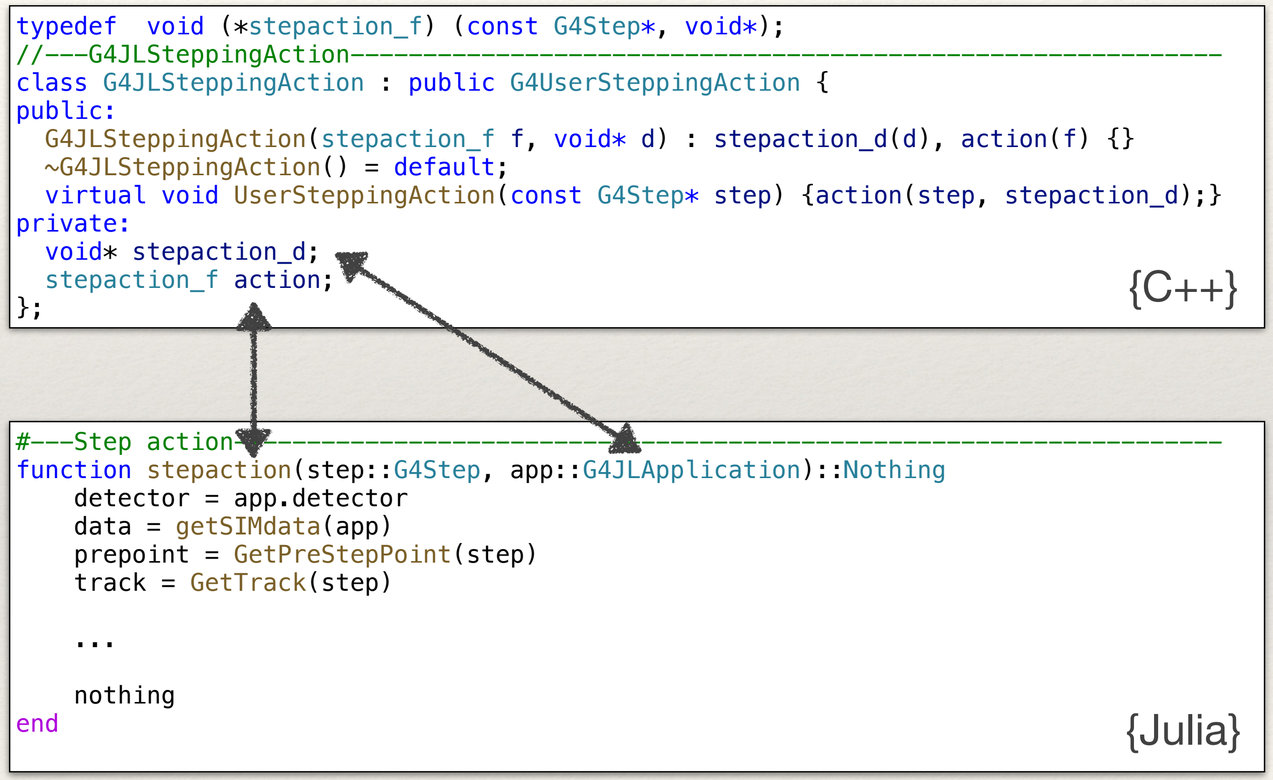


Application Interface: Wish List

- 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
  - Typically by inheriting from a virtual base classes (e.g. `G4UserSteppingAction`, `G4VSensitiveDetector`)
- CxxWrap.jl provides a convenient way to call Julia from C++ using `@safe_cfunction` (argument checking wrapper of `@cfunction`)
  - Needed to bridge the Geant4 inheritance way with thin classes calling specific **closures** (user function + state)
  - Julia native closures not supported for ARM architectures (developed a workaround)
- Pending issue: small memory allocation ~40 bytes each callback



Sensitive Detectors

- A **sensitive detector** is defined with a custom user `struct` to collect hit information and three user defined functions (`initialize`, `endOfEvent` and `processHits`)
- Functions receive its own "worker thread" copy of the data
- Associations to the corresponding G4LogicalVolume are declared at instantiation of `G4JLApplication`

```
#---Define Tracker Hit-----
struct TrackerHit
  trackID::Int32
  chamberNb::Int32
  edep::Float64
  pos::Point3{Float64}
end
```

```
#-------------------------------------------------------------------
#---Define Sensitive Detector--------------------------------------
#-------------------------------------------------------------------
#---SD collected data-----------------------------------------------
struct B2aSDData <: G4JLSDData
  trackerHits::Vector{TrackerHit}
  B2aSDData() = new([])
end

#---Initialize method-----------------------------------------------
function _initialize(::G4HCofThisEvent, data::B2aSDData)::Nothing
  empty!(data.trackerHits)
  return
end

#---End of Event method---------------------------------------------
function _endOfEvent(::G4HCofThisEvent, data::B2aSDData)::Nothing
  # write the event hits to permanent storage
  return
end

#---Process Hit method----------------------------------------------
function _processHits(step::G4Step, ::G4TouchableHistory, data::B2aSDData)::Bool
  edep = step |> GetTotalEnergyDeposit
  edep <  0. && return false
  pos = step |> GetPostStepPoint |> GetPosition
  push!(data.trackerHits,
       TrackerHit(step |> GetTrack |> GetTrackID,
                  step |> GetPreStepPoint |> GetTouchable |> GetCopyNumber,
                  edep,
                  Point3{Float64}(x(pos),y(pos),z(pos))))
  return true
end

#---Create SD instance----------------------------------------------
chamber_SD = G4JLSensitiveDetector("Chamber_SD", B2aSDData();
                                   processhits_method=_processHits,
                                   initialize_method=_initialize,
                                   endofevent_method=_endOfEvent)
```

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!
- **Foreign thread adoption** introduced with v1.9 has been essential
  - care has been taken to avoid dead locks caused by GC (jl_gc_safe_enter/jl_gc_safe_leave)
- User actions, sensitive detector code, etc. will be run naturally on different threads
  - To avoid race conditions, better if each thread uses/updates its own copy of the data
  - Data is cloned for each thread and summed at the end of the run

User Actions

- **User actions** are native Julia functions (called by C++)
- In addition to the G4 standard arguments they get a reference to the `G4JLApplication` to get all simulation context (no globals!)
- All worker threads are calling the same user action functions (user needs to prevent data racing)

```julia
#---Begin Event Action-----------------------------------------------------------
function beginevent(evt::G4Event, app::G4JLApplication)::Nothing
    data = getSIMdata(app)
    fill!(data.fEnergyDeposit, 0.0)
    fill!(data.fTrackLengthCh, 0.0)
    return
end
```

its own thread local copy of the simulation data

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

    # Return if step in not in the world volume
    prepoint |> GetPhysicalVolume |> GetLogicalVolume |> GetMaterial
             == detector.fWorldMaterial && return nothing

    particle = GetDefinition(track)
    charge  = GetPDGCharge(particle)
    stepl = 0.
    if charge != 0.
        stepl = GetStepLength(step)
        data.fChargedStep += 1
    else
        data.fNeutralStep += 1
    end
    edep = GetTotalEnergyDeposit(step) * GetWeight(track)
    absorNum  = GetCopyNumber(GetTouchable(prepoint), 0)
    layerNum  = GetCopyNumber(GetTouchable(prepoint), 1) + 1

    data.fEnergyDeposit[absorNum] += edep
    data.fTrackLengthCh[absorNum] += stepl

    push!(data.fEdepHistos[absorNum], layerNum, edep)
    return
end
```

its own thread local copy of the simulation data

Detector Geometry

- Main detector parameters encapsulated in a user `struct` inheriting from `G4JLDetector`
- A user defined Julia function will be called at the right moment to construct the geometry, receiving the detector parameters
  - For the time being the same API
  - Object ownership is delicate and in most cases hidden
- Define the geometry with a GDML file is also possible

{Parameters}

```
mutable struct TestEm3Detector <: G4JLDetector
    # main input parameters
    const fNbOfAbsor::Int64
    const fNbOfLayers::Int64
    const fCalorSizeYZ::Float64
    const fAbsorThickness::Vector{Float64}

    # mutable (computed) detector data
    fLayerThickness::Float64
    fCalorThickness::Float64
    fWorldSizeYZ::Float64
    fWorldSizeX::Float64
    ...
    function TestEm3Detector(;nbOfLayers = 50,
                              calorSizeYZ = 40cm,
                              absorThickness = [2.3mm, 5.7mm],
                              absorMaterial = ["G4_Pb", "G4_lAr"])
        self = new(length(absorThickness), nbOfLayers, calorSizeYZ, absorThickness)
        ...
    end
end
```

{Construct Function}

```
function TestEm3Construct(det::TestEm3Detector)

    (; fNbOfAbsor, fNbOfLayers, fCalorSizeYZ, fAbsorThickness, fLayerThickness,
       fCalorThickness, fWorldSizeYZ, fWorldSizeX, fWorldMaterial) = det

    println("Building Geometry now!!!")

    #---World---------------------------------------------------------------
    det.fSolidWorld = G4Box("World", fWorldSizeX/2,fWorldSizeYZ/2,fWorldSizeYZ/2)
    det.fLogicWorld = G4LogicalVolume(det.fSolidWorld, fWorldMaterial, "World")
    det.fPhysiWorld = G4PVPlacement(nothing,              # no rotation
                                    G4ThreeVector(),      # at (0,0,0)
                                    det.fLogicWorld,      # its fLogical volume
                                    "World",              # its name
                                    nothing,              # its mother  volume
                                    false,                # no boolean operation
                                    0)                    #copy number
    ...
end

Geant4.getConstructor(::TestEm3Detector)::Function = TestEm3Construct
```

Magnetic Field

- The user provides a data struct inheriting from `G4JLFieldData` and the callback function `getfield!(...)` that will be called each time the field needs to be evaluated
- Also can use pre-defined fields such as `G4JLUniformMagField`

{Parameters}

{Function}

```
#---Field Parameters-------------------------------------
mutable struct CustomFieldData <: G4JLFieldData
    field::G4ThreeVector
end

#---Callback functional called by Geant4----------------
function getfield!(field::G4ThreeVector, pos::G4ThreeVector,
                   data::CustomFieldData)::Nothing
    assign(field, data.field)
    return
end

#---Instantiate a the field object----------------------
field = G4JLMagneticField("Uniform",
                          CustomFieldData(fieldvector);
                          getfield_method=getfield!)
```

Primary Particle Generator

- The user can use the predefined `G4JLGunGenerator` generator, or define his/her own generator by defining a `struct` for the parameters and two functions: `init` and `generate` that will be called at the adequate moment
- Implicit multiple instances will be created in case on MT

{Parameters}

```
mutable struct MedicalBeamData <: G4JLGeneratorData
  particleName::String
  particlePtr::CxxPtr{G4ParticleDefinition}
  energy::Float64
  ssd::Float64
  fieldXY::Float64
  surfaceZ::Float64
end
```

{Functions}

```
function MedicalBeam(;particle="e-", energy=10MeV, ssd=100cm, fieldXY=10cm)
  data = MedicalBeamData(particle, CxxPtr{G4ParticleDefinition}(C_NULL),
                         energy, ssd, fieldXY, 0.)

  function init(data::MedicalBeamData, app::G4JLApplication)
    data.particlePtr = FindParticle(data.particleName)
    data.surfaceZ = -app.detector.phantomZ/2
  end

  function generate( evt::CxxPtr{G4Event}, data::MedicalBeamData)::Nothing
    mass = data.particlePtr |> GetPDGMass
    momemtum = √((mass + data.energy)^2 - mass^2)
    pvec = momemtum * generateBeamDir(data.ssd, data.fieldXY);
    primary = G4PrimaryParticle(data.particlePtr, pvec |> x, pvec |> y, pvec |> z )
    vertex = G4PrimaryVertex(G4ThreeVector(0, 0, data.surfaceZ - data.ssd), 0ns)
    SetPrimary(vertex, move!(primary))
    AddPrimaryVertex(evt, move!(vertex))
  end

  G4JLPrimaryGenerator("MedicalBeam", data; init_method=init,
                                            generate_method=generate)
end
```

```
# Predefined Particle Gun-------------------------------------------------
particlegun = G4JLGunGenerator(particle = "e-",
                               energy = 1GeV,
                               direction = G4ThreeVector(1,0,0),
                               position = G4ThreeVector(0,0,0))
```

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
```

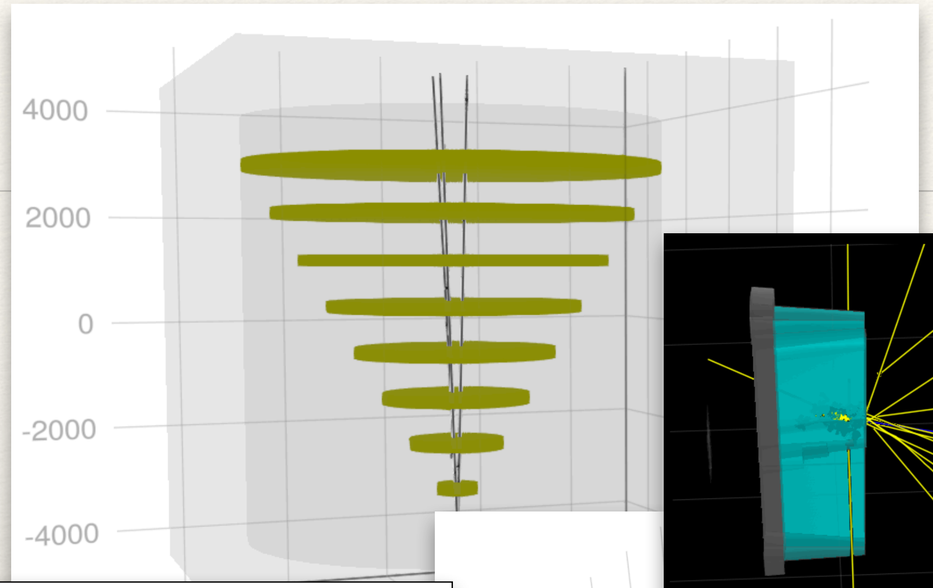
Simulation Application

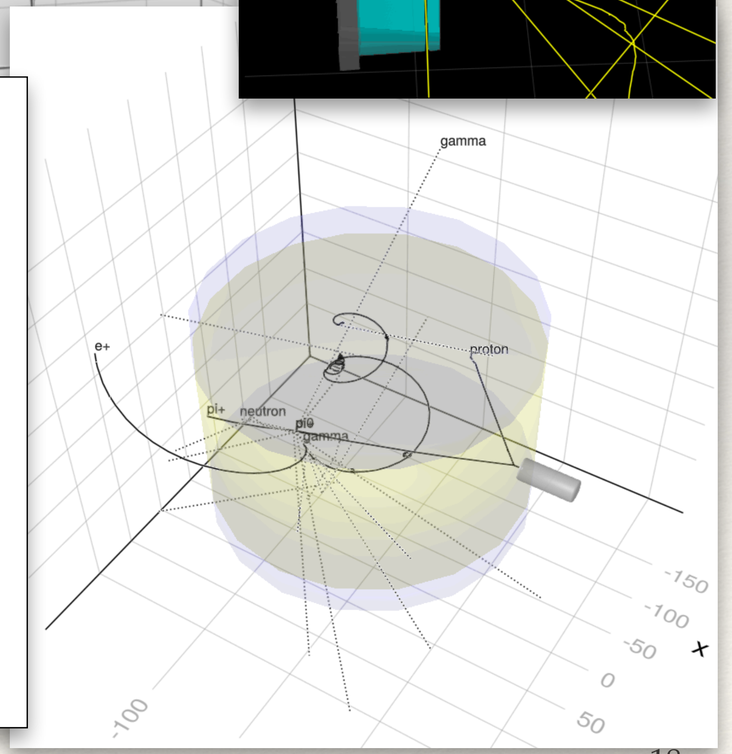
- Finally, the user can create a `G4JLApplication` with all the elements of the simulation application (detector geometry, primary generator, physics list, user actions, etc.)
- Geant4 requires a strict order of instantiation/configuration/initialization and this is guaranteed by Geant4.jl interface
- 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)
```

G4Visualization

- Implemented basic visualisation of the geometry using Makie.jl package
  - including boolean solids
- Also tested some trivial tracking visualisation
  - collected points (vector of points) in the stepping action






```
using Geant4
using G4Visualization
using GLMakie

include("../../examples/basic/B2/B2a.jl")
world = GetWorldVolume()
draw(world)

tracks = app.simdata.tracks
empty!(tracks)

beamOn(app, 1)

for t in tracks
    lines!(t)
end
```

TestEm3 example

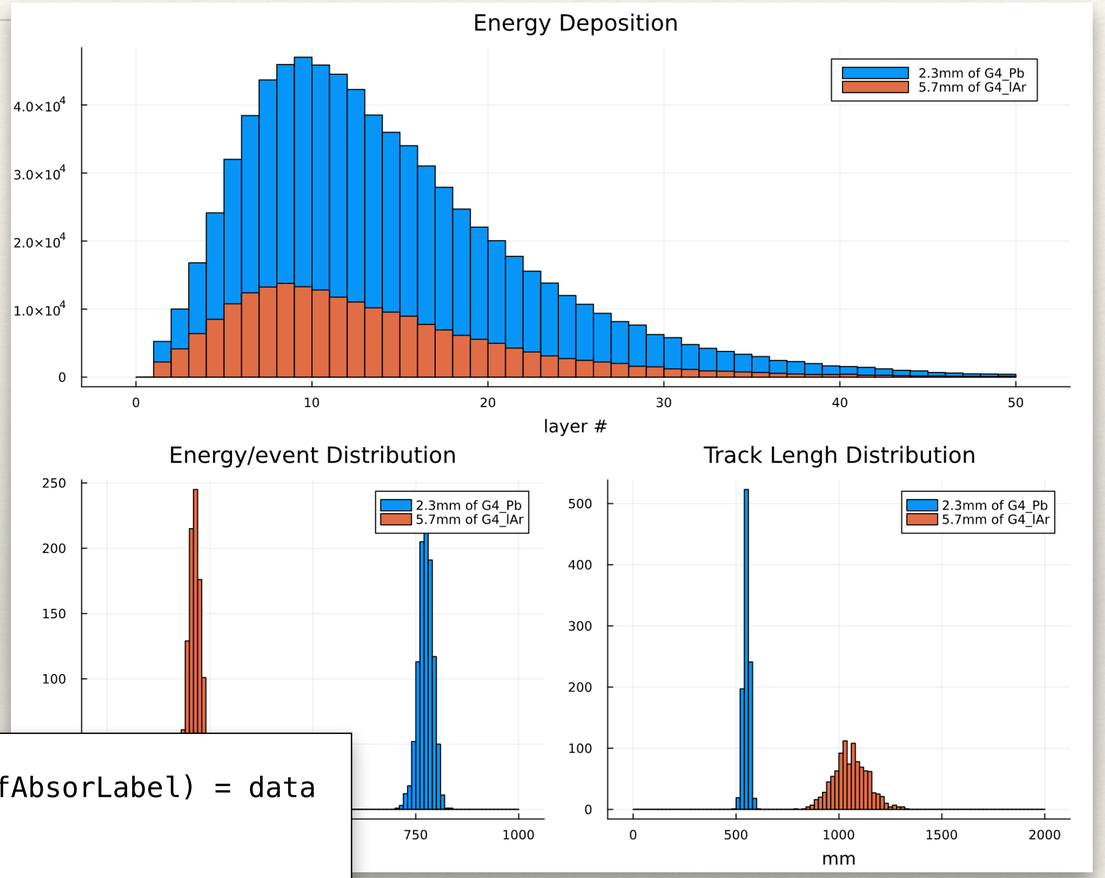
- This example works with user actions only (step, event, run, track)
  - simplified from the original
- Using histograms from FHist.jl

```
mutable struct TestEm3SimData <: G4JLSimulationData
  fParticle::CxxPtr{G4ParticleDefinition}
  fEkin::Float64

  fChargedStep::Int32
  fNeutralStep::Int32
  fN_gamma::Int32

  fN_elec::Int32
  fN_pos::Int32

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

  fEdepEventHistos::Vector{Hist1D64}
  fTrackLengthChHistos::Vector{Hist1D64}
  fEdepHistos::Vector{Hist1D64}

  fAbsorLabel::Vector{String}
end
```

```
#---Create the Application-------------------------------------------------------------
app = G4JLApplication(detector = TestEm3Detector(),                 # detector with parameters
                      simdata = TestEm3SimData(),                   # simulation data structure
                      nthreads = 8,                                 # number of threads
                      physics_type = FTFP_BERT,                     # what physics list to instantiate
                      generator = G4JLGunGenerator(...),            # primary generator instance
                      #----Actions-------------------------------
                      stepaction_method = stepaction,               # step action method
                      pretrackaction_method = pretrackaction,       # pre-tracking action
                      posttrackaction_method = posttrackaction,     # post-tracking action
                      beginrunaction_method = beginrun,             # begin-run action (initialise counters and histos)
                      endrunaction_method = endrun,                 # end-run action (print summary)
                      begineventaction_method = beginevent,         # begin-event action (initialise per-event data)
                      endeventaction_method = endevent)             # end-event action (fill histogram per event data)
```

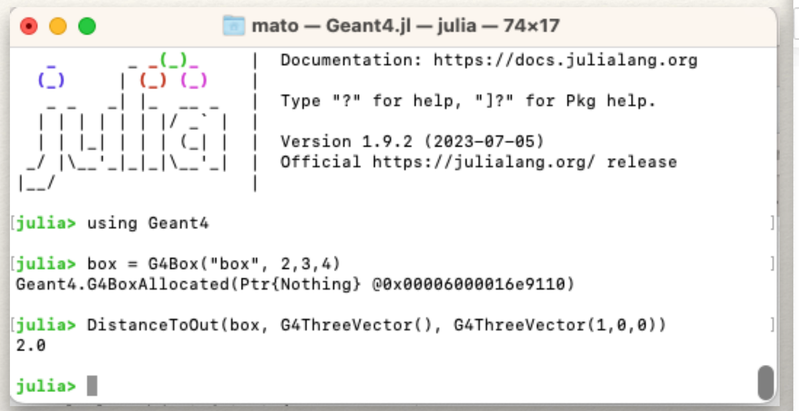
TestEm3 example - display results

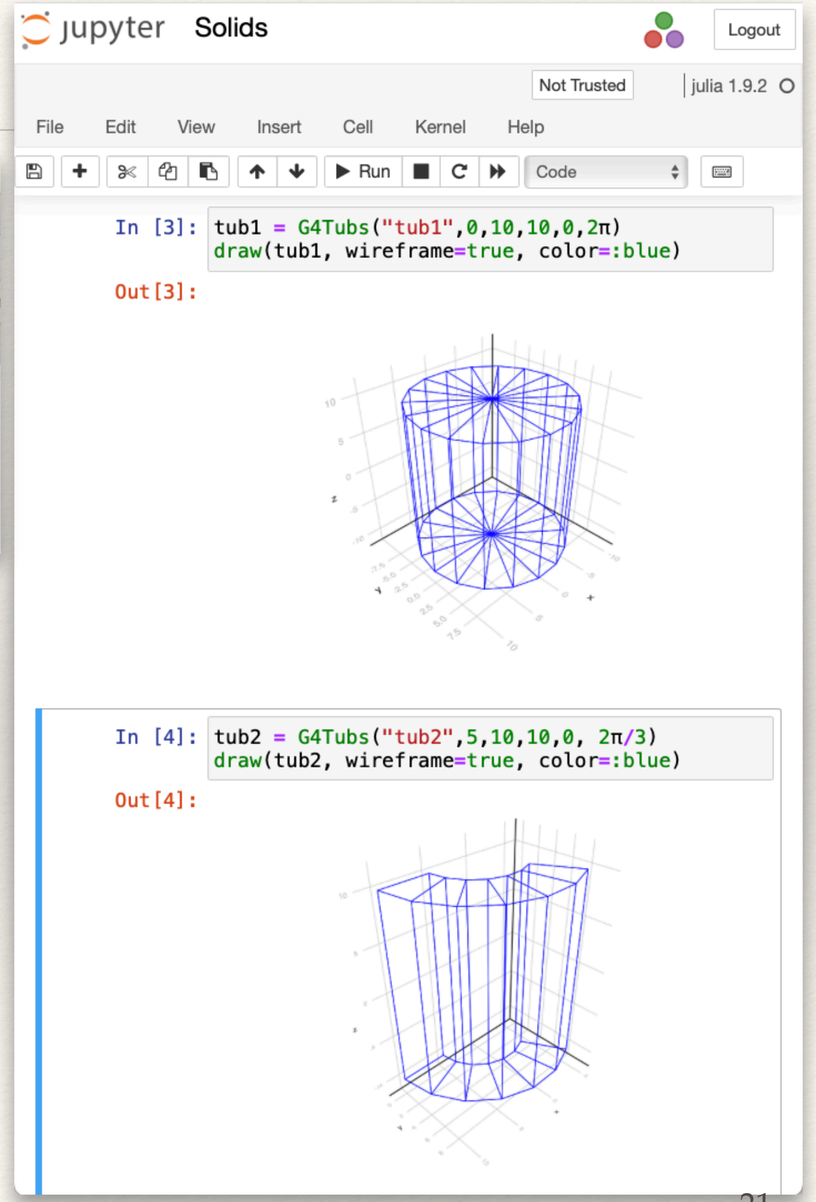
- After running the desired number of events, the simulation data structure can be passed to a plotting function or an analysis code
- Changes can be made interactively and a new run can be started
  - E.g. new detector with different parameters, new particle gun parameters, different callbacks, etc




```
function do_plot(data::TestEm3SimData)
    (;fEdepHistos, fEdepEventHistos, fTrackLengthChHistos, fAbsorLabel) = data
    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
```

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

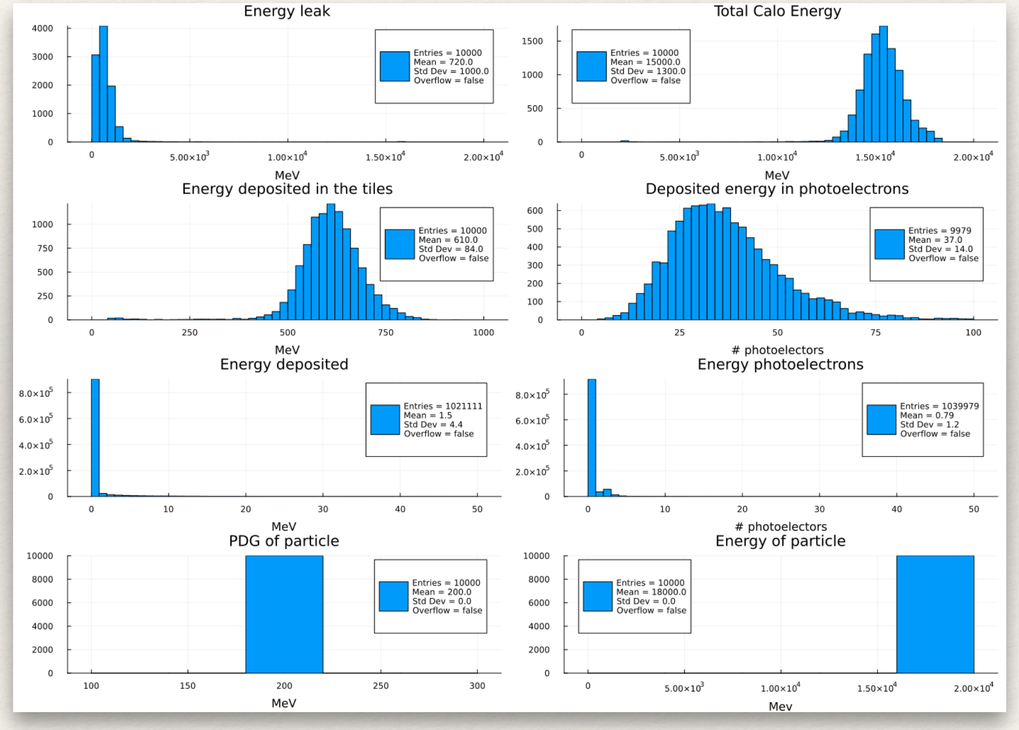
- Performance should be equivalent to the C++ application
- Julia user actions (callbacks from C++ to Julia) do not add any significant overhead and can be executed very efficiently
  - JIT and with less abstraction layers
- Julia suffers from a larger startup time (final type inference and JIT compilation)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

Adding more Complete and Realistic Examples

- The package Geant4.jl comes with a number of examples
  - extended/RE03 (scoring meshes)
  - TestEm3 (user actions with data analysis integration)
  - WaterPhantom (scoring meshes, special particle generator, plotting results)
  - HBC30 (bubble chamber with event display and online trigger)
  - Scintillator (optical photons and customised physics list)
- Recently 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)



Comparing with other Bindings

- How Geant4 bindings compare with respect the native C++?
  - Evaluated **geant4_pybind** an **Geant4.jl** with respect a number of criteria
- This is [perhaps a bit biased] summary:

	C++	PyBind	Julia
Ease of Installation	Fair	Very Good	Very Good
Friendliness of API	Bad	Fair	Good
Interactivity	Fair	Very Good	Very Good
Running Speed	Very Good	Bad	Very Good
MT support	Good	Very Bad	Very Good
Visualization/ Analysis	Fair	Good	Good

Conclusions

- The package **CxxWrap** works nicely and scales relatively well
  - Callbacks from C++ to Julia are essential for Geant4 . Measured very small overhead
  - Did not find any limitation in the C++ interfaces used by G4 (*, const *, &, const &, ...) modulo object ownership peculiarities of G4
- Julia **BinaryBuilder** is a powerful tool to streamline the installation and deployment of C++ projects and make it easier for users to get started with Julia-based applications
- **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 still in a **prototype state** with probably missing functionality but **very promising**
  - **Fully exploiting** the tooling of Julia ecosystem (CI, documentation, registration and deployment, etc.)