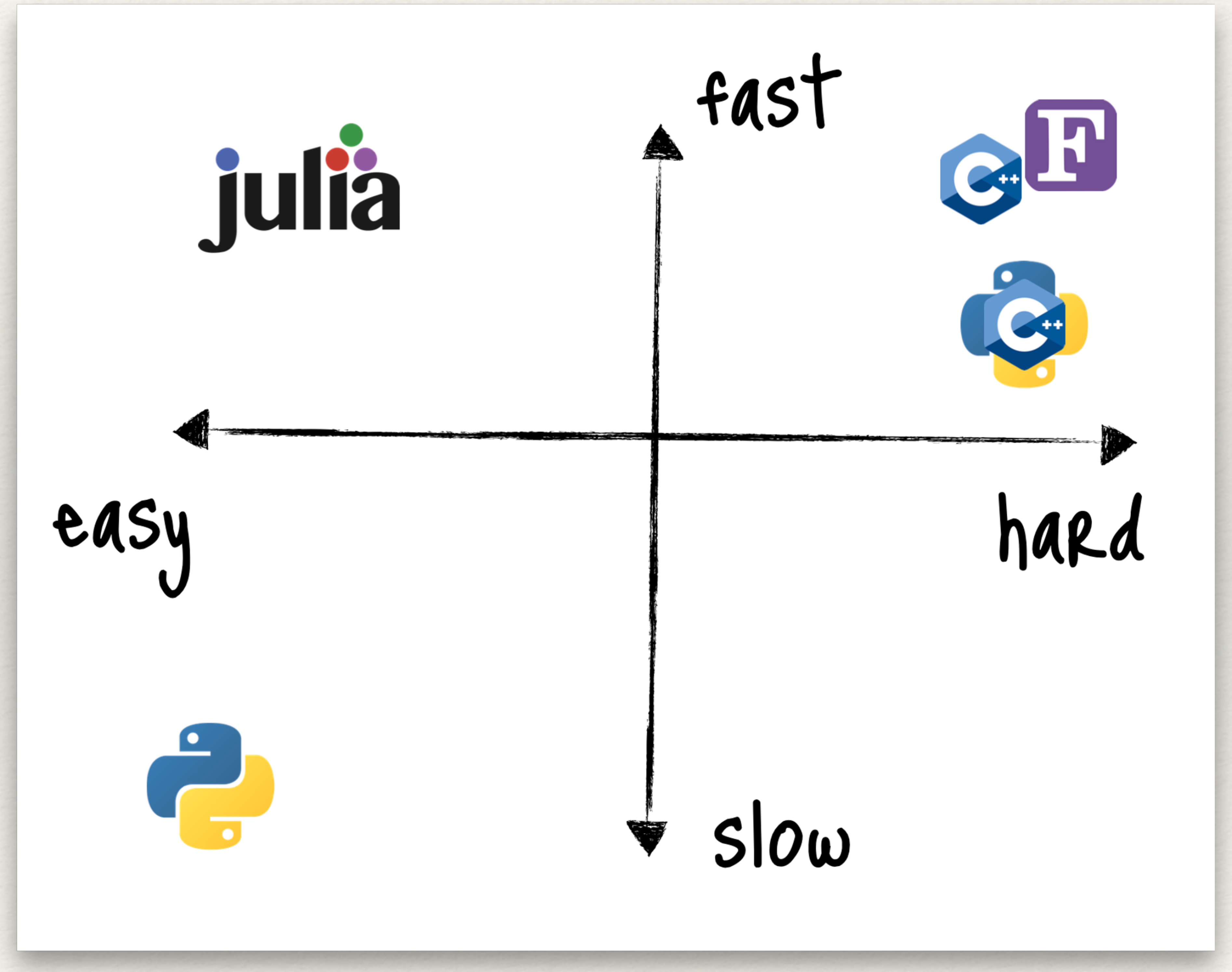
8th FCC Physics Workshop

Analysis with the Julia language

Pere Mato / CERN
14 January 2025

Why Julia?

- Invented in 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



Julia main Features

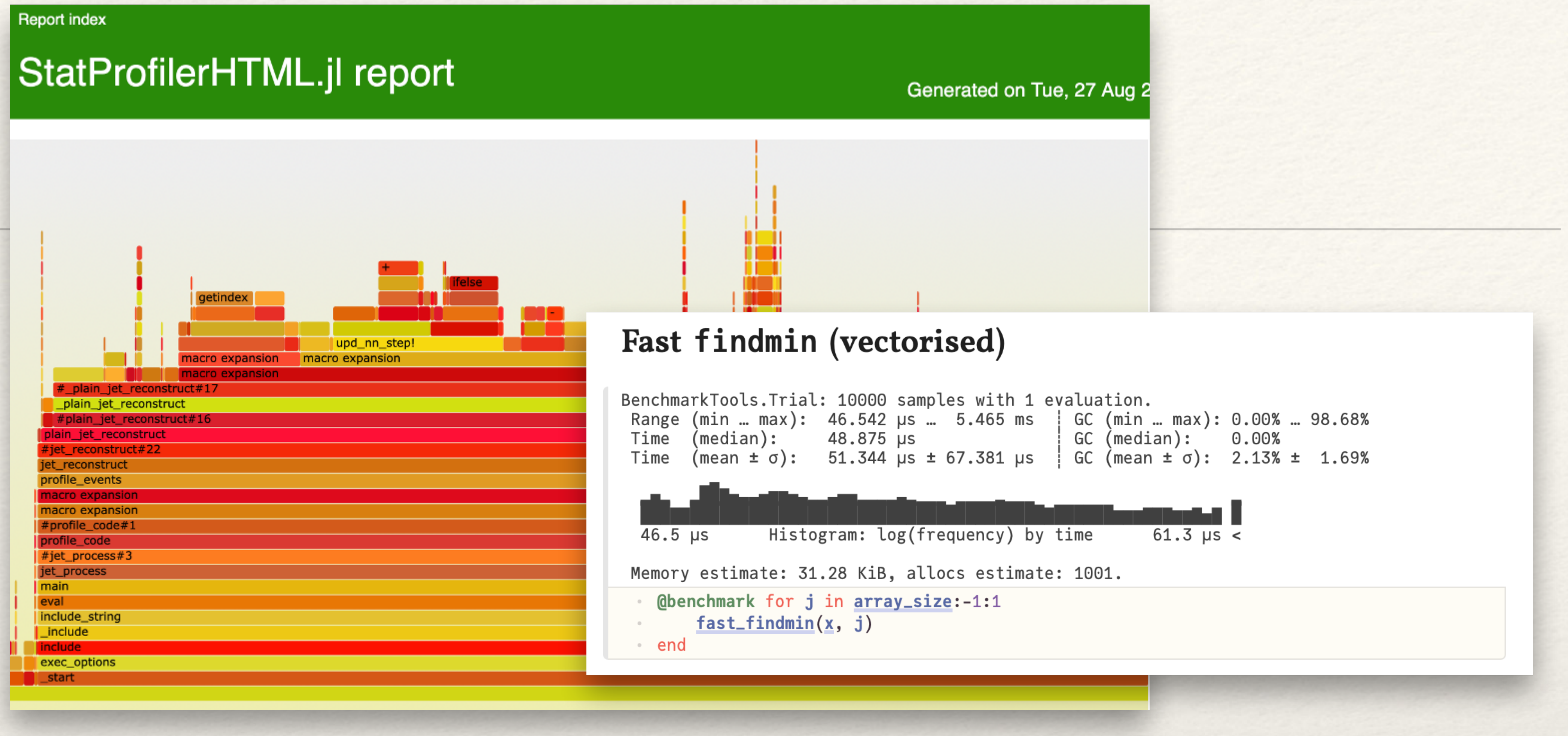
- Easy of use
  - REPL, notebooks, garbage collected, expressive maths syntax
- Fast
  - Not interpreted. Just ahead of time compiler (powered by LLVM)
  - Reflexion, meta-programing, threads, vectorization, GPU support, HPC, etc.
- Advanced type system
  - Powerful and sophisticated type expressions
- Multiple dispatch
  - This allows packages to compose packages without knowing about each other

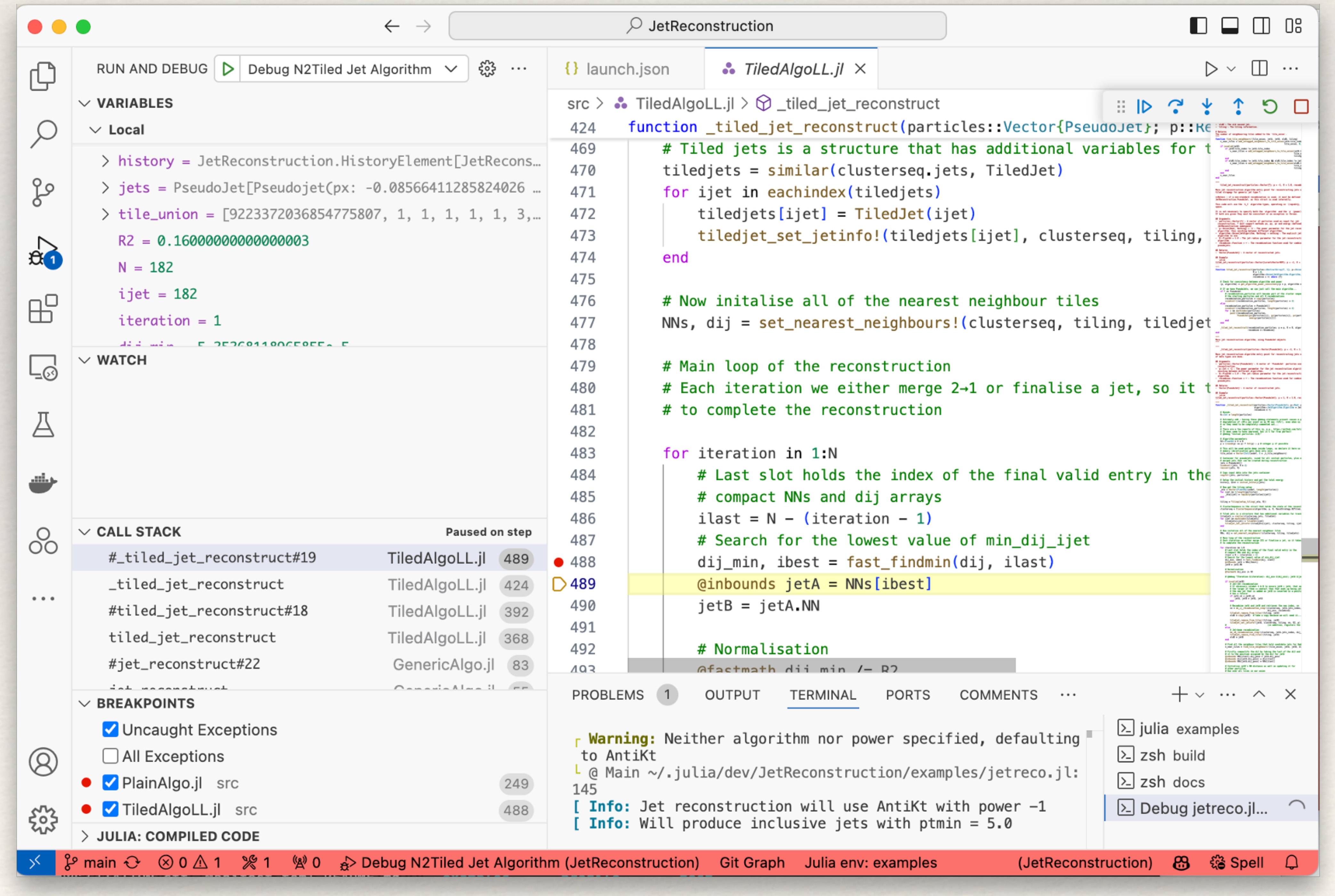
The Two Language Problem

- HEP needs a solution to 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)

Excellent Tooling

- 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





Rich Ecosystem

- More than 10k packages

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

What do we need for FCC analysis

- **Access to the Data**
  - Read access to ROOT files of EDM4hep events in EOS (XRootD protocol)
- **Analysis Tools and Algorithms**
  - Availability of a extensive ecosystem of tools (e.g. histogramming, statistics, ML) and HEP specific algorithms (e.g. jet finding, flavor tagging)
- **Plotting**
  - Data visualization specific to HEP common practices
- **Scaling Out**
  - Multi-core, accelerators (GPUs), multi-nodes, grid and cloud computing, etc.

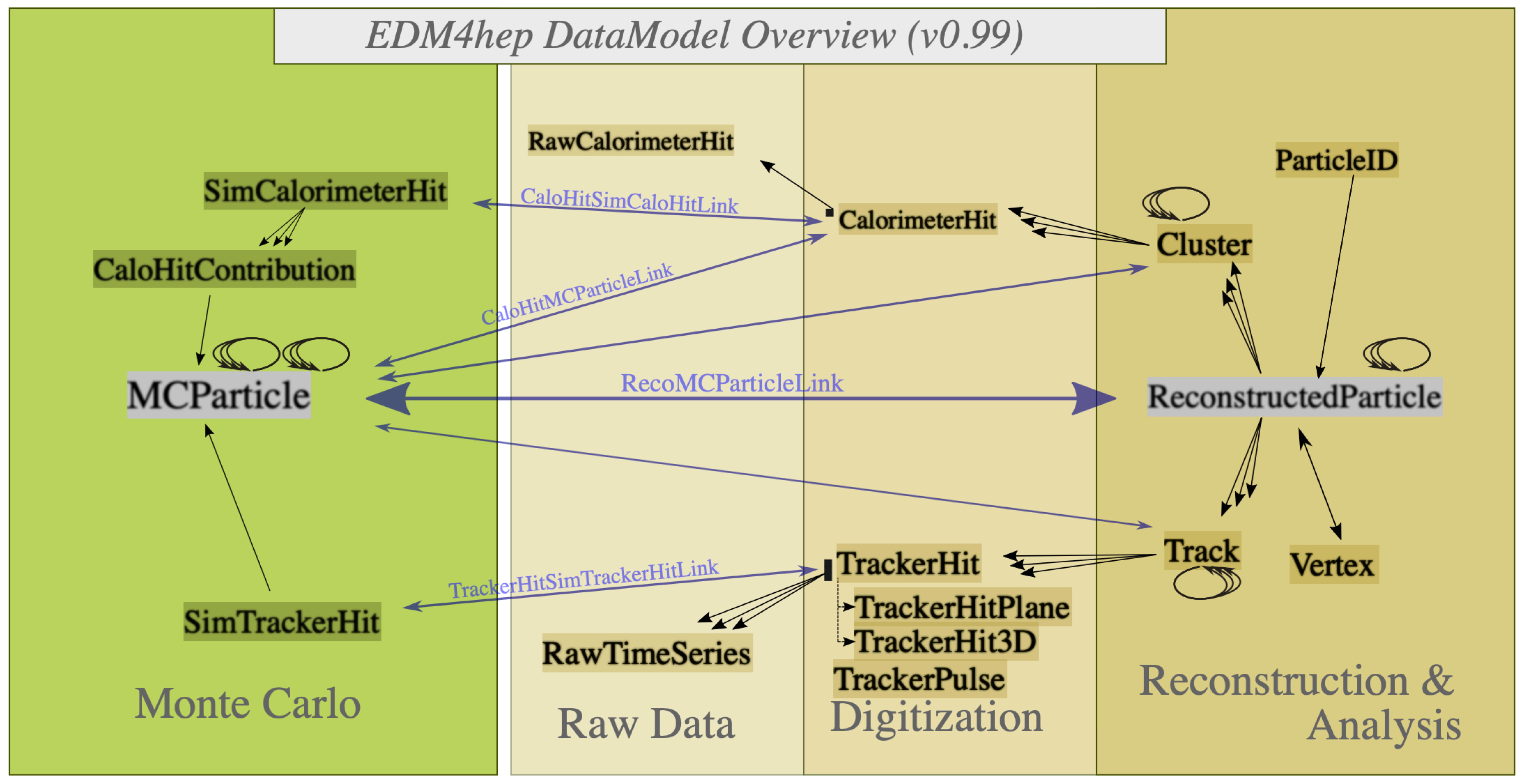
Reading EDM4hep files

https://github.com/peremato/EDM4hep.jl

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

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

The EDM4hep Data Model



- Covering the simulation / digitization / reconstruction / analysis domains

EDM4hep.jl

- Generate Julia ‘friendly’ data structures for the EDM4hep data model
  - Using the same YAML file used by PODIO to generate C++ code
- Be able to read event data files (in ROOT format) written by C++ programs from Julia
  - Using the UnROOT.jl package, which itself makes use of the XRootD.jl package (wrapper for the XRootD package) to read from remote files
- Later, be able also to write RNTuple files from Julia

Main Design Features

- All entities are **immutable structs** for better performance, SoA, GPUs, etc.
  - POD with basic types and structs, including the relationships (one-to-one and one-to-many)
  - Object attributes cannot be changed, new instances can be created with Accessors.jl
- Constructors have **keyword arguments** with reasonable default values
- New objects are by default not registered, they are "**free floating**". Explicit registration or setting relationships will register them to containers.
- Note that operations like **register**, **setting relationships** will automatically create a new instances. The typical pattern is to overwrite the user variable with the new instance, e.g.:

```
p1 = MCParticle(...)
p1, d1 = add_daugther(p1, MCParticle(...))
```

- Reading EDM4hep containers from ROOT will result in highly efficient **StructArrays**
  - Very efficient access by column and the same time provide convenient views as object instances

PODIO Generation

- Written small Julia script to generate Julia structs from YAML file
  - Added a **`ObjectID`** to each object to control its registration state
  - Relations implemented with **`ObjectID`** and **`Relation`** structs with just indices (isbits() = POD)
- Two files: **`genComponents.jl`**, **`genDatatypes.jl`** generated that can be complemented with utility methods

```julia
"""
struct MCParticle

    Description: The Monte Carlo particle - based on the lcio::MCParticle.
    Author: F.Gaede, DESY
"""
struct MCParticle <: POD
    index::ObjectID{MCParticle}      # ObjectID of itself
   #---Data Members
    PDG::Int32                       # PDG code of the particle
    generatorStatus::Int32           # status of the particle as defined by the ...
    simulatorStatus::Int32           # status of the particle from the simulation ...
    charge::Float32                  # particle charge
    time::Float32                    # creation time of the particle in [ns] wrt. ...
    mass::Float64                    # mass of the particle in [GeV]
    vertex::Vector3d                 # production vertex of the particle in [mm].
    endpoint::Vector3d               # endpoint of the particle in [mm]
    momentum::Vector3f               # particle 3-momentum at the production vertex..
    momentumAtEndpoint::Vector3f     # particle 3-momentum at the endpoint in [GeV]
    spin::Vector3f                   # spin (helicity) vector of the particle.
    colorFlow::Vector2i              # color flow as defined by the generator

    #---OneToManyRelations
    parents::Relation{MCParticle,1}  # The parents of this particle.
    daughters::Relation{MCParticle,2}  # The daughters this particle.
end
```

```julia
"""
struct SimTrackerHit

    Description: Simulated tracker hit
    Author: F.Gaede, DESY
"""
struct SimTrackerHit <: POD
    index::ObjectID{SimTrackerHit}   # ObjectID of itself
    #---Data Members
    cellID::UInt64                   # ID of the sensor that created this hit
    EDep::Float32                    # energy deposited in the hit [GeV].
    time::Float32                    # proper time of the hit in the lab frame in ...
    pathLength::Float32              # path length of the particle in the sensiti ...
    quality::Int32                   # quality bit flag.
    position::Vector3d               # the hit position in [mm].
    momentum::Vector3f               # the 3-momentum of the particle at the hits ...
    #---OneToOneRelations
    mcparticle_idx::ObjectID{MCParticle}  # MCParticle that caused the hit.
end
```

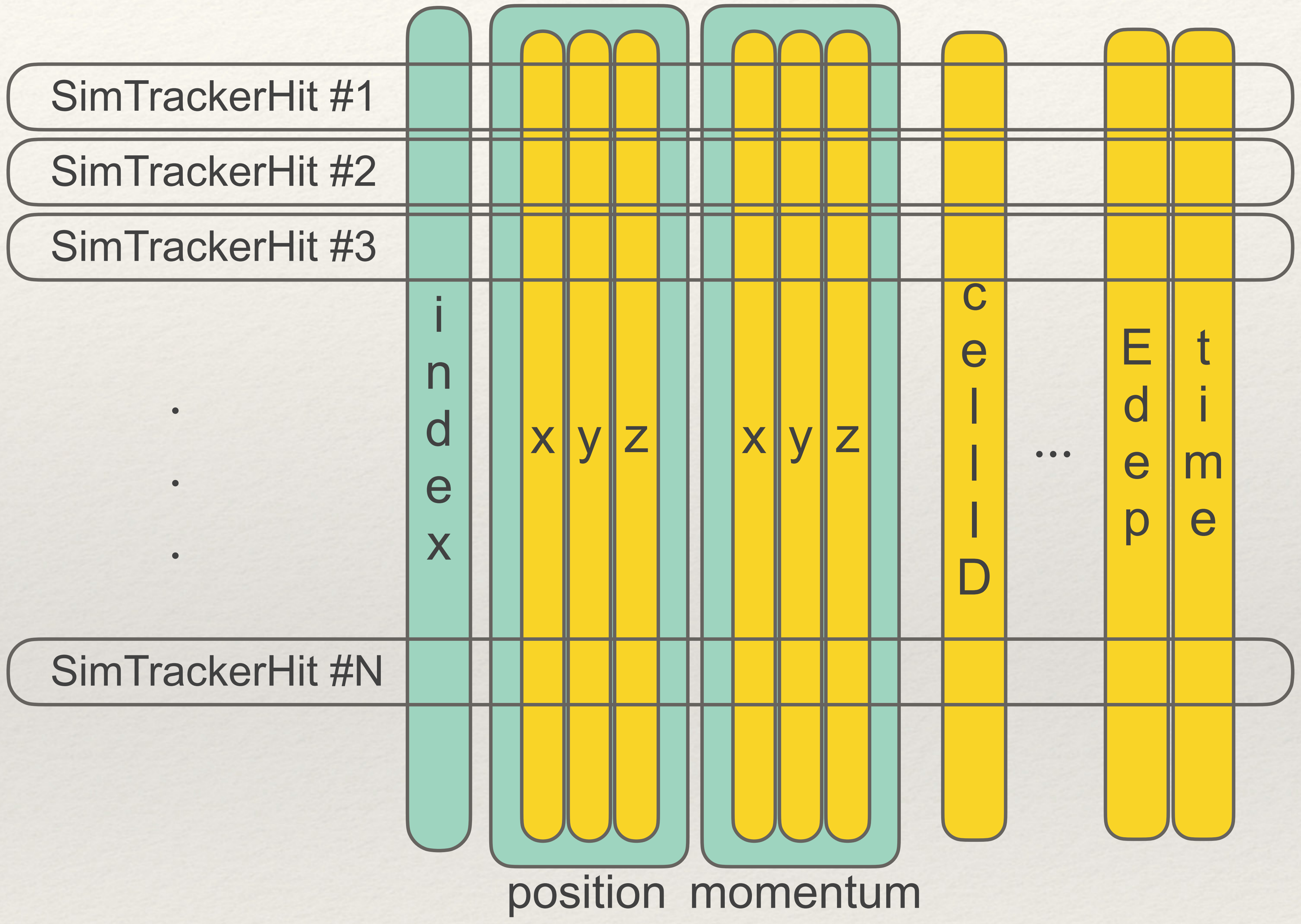
ROOT I/O

- Using **UnROOT.jl** package (equivalent to UpROOT in Python)
- Supports (transparently) TTree and RNTuple formats and several versions of PODIO storage (versions 16.x and 17.x)
  - data files consist exclusively of 'collections-of-datatypes' (e.g. ReconstructedParticles, Vertices, etc.)
- The goal is to obtain a **`StructArray{DataType}`** of each collection for each event
  - The exercise consists in mapping the schema in the ROOT file to the actual Julia datatype (using the Julia introspection or generated code)
- **XRootD.jl**
  - Wrapper to C++ XRootD providing the **File** (remote file access) and **FileSystem** (files and directories operations) interface

Creating SoAs from EDM4hep types

- UnROOT.jl provides the leaf arrays (in a lazy manner) and they are "mapped" to form SoA of a DataType
- Opens the possibility of schema evolution
  - filling empty attributes, type change, re-shaping, etc.

```
using StructArrays

# Create a struct array
hits = StructArray{SimTrackerHit}(Tuple(<TLeaf>...))


# Access elements
println(hits[1])  # Output: SimTrackerHit(....)
```



Reading from a ROOT (TTree) File

```
using EDM4hep
using EDM4hep.RootIO

cd(@__DIR__)

f = "ttbar_edm4hep_digi.root"

reader = RootIO.Reader(f)
events = RootIO.get(reader, "events")

evt = events[1];

hits = RootIO.get(reader, evt, "InnerTrackerBarrelCollection")
mcps = RootIO.get(reader, evt, "MCParticle")

for hit in hits
    println("Hit $(hit.index) is related to MCParticle $(hit.mcparticle.index) with name $(hit.mcparticle.name)")
end

#---Loop over events-------------------------------------------------------------------------
for (n,e) in enumerate(events)
    ps =  RootIO.get(reader, e, "MCParticle")
    println("Event #$(n) has $(length(ps)) MCParticles with a charge sum of $(sum(ps.charge))")
end
```

```
Hit #1 is related to MCParticle #65 with name pi+
Hit #2 is related to MCParticle #65 with name pi+
Hit #3 is related to MCParticle #65 with name pi+
Hit #4 is related to MCParticle #65 with name pi+
Hit #5 is related to MCParticle #66 with name pi-
Hit #6 is related to MCParticle #66 with name pi-
Hit #7 is related to MCParticle #66 with name pi-
Hit #8 is related to MCParticle #49 with name pi+
Hit #9 is related to MCParticle #49 with name pi+
Hit #10 is related to MCParticle #49 with name pi+
Hit #11 is related to MCParticle #27 with name K-
Hit #12 is related to MCParticle #27 with name K-
Hit #13 is related to MCParticle #27 with name K-
Hit #14 is related to MCParticle #95 with name e-
Hit #15 is related to MCParticle #95 with name e-
...
```

~ 1500 times faster than Python

StructArray provides an Ergonomic Interface

- Storage in memory consists of a set of column arrays
  - very fast access by column
- Materialize, when requested, object instances (usually on the stack) to be able to call user object methods (multiple dispatch)
  - achieving a user friendly access

```
julia> mcps = <get all MCParticle collection>
julia> typeof(mcps)
StructVector{MCParticle, ...}

julia> typeof(mcps[1])
MCParticle

julia> typeof(mcps.charge)
SubArray{Float32, 1, Vector{Float32},
Tuple{UnitRange{Int64}}, true}

julia> length(mcps.charge)
211

julia> mcps[1:2].momentum
2-element StructArray(::Vector{Float32}, ::Vector{Float32},
::Vector{Float32}) with eltype Vector3f:
 (0.5000167,0.0,50.0)
 (0.5000167,0.0,-50.0)

julia> sum(mcps[1:2].momentum)
(1.0000334,0.0,0.0)
```

StructArray provides an Efficient Interface

- Example applying some transformation to a collection (e.g. unBoost crossing angle to the collection of Reconstructed Particles)
    - Avoiding the explicit loop you can get a factor 15 in this example

```
function unBoostCrossingAngle(in, angle)
    ta = tan(angle)
    e = in.energy
    pₓ = in.momentum.x
    e′  = e * sqrt(1 + ta^2) + pₓ * ta
    pₓ′ = pₓ * sqrt(1 + ta^2) + e * ta
    return @set (@set in.momentum.x = pₓ′).energy = e′
end
```

vector of energies

vector of px

set the full column

```
function unBoostCrossingAngle_loop(in, angle)
    result = StructArray(ReconstructedParticle[])
    ta = tan(angle)
    for p in in
        e = p.energy
        pₓ = p.momentum.x
        e′ = e * sqrt(1 + ta*ta) + pₓ * ta
        pₓ′ = pₓ * sqrt(1 + ta*ta) + e * ta
        push!(result, @set (@set p.momentum.x=pₓ′).energy = e′)
    end
    return result
end
```

```
julia> rps = RootIO.get(reader, evt, "PandoraPFOs");

julia> @btime unBoostCrossingAngle($rps, -0.015rad);
  316.449 ns (12 allocations: 2.81 KiB)

julia> @btime unBoostCrossingAngle_loop($rps, -0.015rad);
  4.806 μs (68 allocations: 36.97 KiB)
```

Package EDM4hep.jl is ready for use!

- Install Julia

```
curl -fsSL https://install.julialang.org | sh
```

- Install EDM4hep

```
julia -e ‘import Pkg; Pkg.add(“EDM4hep”)’
```

```
julia> using EDM4hep
julia> using EDM4hep.RootIO
julia> file = "root://eospublic.cern.ch//eos/experiment/fcc/ee/generation/DelphesEvents/winter2023/IDEA/
p8_ee_ZZ_ecm240/events_000189367.root"
julia> reader = RootIO.Reader(file)
```

Atribute	Value
File Name(s)	root://eospublic.cern.ch//eos/experiment/fcc/prod/fcc/ee/test_spring2024/240gev/Hbb/CLD_o2_v05/rec/00016562/000/Hbb_rec_16562_1.root
# of events	100
IO Format	TTree
PODIO version	0.99.0
ROOT version	6.28.10

```
julia> events = RootIO.get(reader, "events");
julia> evt = events[1];
julia> recps = RootIO.get(reader, evt, "PandoraPFOs");
julia> recps.energy[1:5]
5-element Vector{Float32}:
```

Analysis Tools and Algorithms

https://github.com/Moelf/FHist.jl

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

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

Event Loop and Analysis functions

- The event loop can be explicit
  - No performance penalty
  - Event selection cuts are very visible and natural

```
for evt in events
    nevents += 1

    # get collection of ReconstructedParticles
    recps = RootIO.get(reader, evt, "PandoraPFOs")

    muons_all = filter(x -> abs(x.type) == 13, recps)
    muons_sel = filter(x -> norm(x.momentum) > 20GeV, muons_all)
    ...
    # CUT 1: at least a lepton with at least 1 isolated one
    length(muons_sel) >= 1 && length(muons_iso) > 0 || continue
    data.μ1 += 1
    ...
end
```

- Analysis functions are simple to write using the EDM4hep types directly

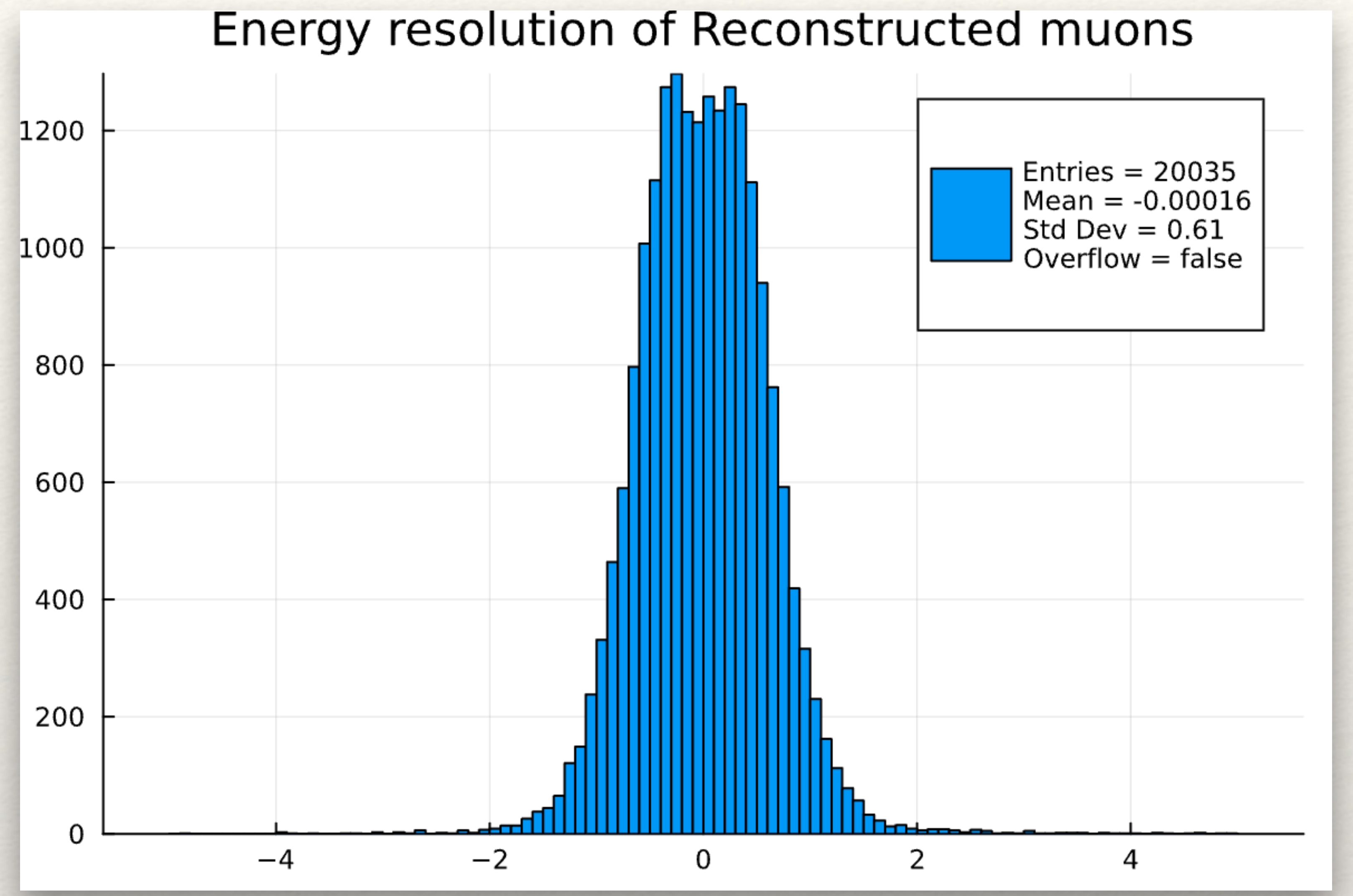
```
function missingEnergy(ecm, rps, p_cutoff)
    p = -sum(r.momentum for r in rps if pₜ(r) >= p_cutoff)
    e =  sum(r.energy for r in rps if pₜ(r) >= p_cutoff)
    ReconstructedParticle(momentum=(p.x, p.y, p.z), energy=ecm-e)
end
```

- Easy to add utility functions to the types

```
pₜ(p::ReconstructedParticle) = √(p.momentum.x^2 + p.momentum.y^2)
θ(p::ReconstructedParticle) = atan(√(p.momentum.x^2+p.momentum.y^2), p.momentum.z)
ϕ(p::ReconstructedParticle) = atan(p.momentum.y, p.momentum.x)
```

Example: μ energy resolution

```
hresolu = H1D("Resolution [GeV]", 100, -5., 5., unit=:GeV)

get_recps = create_getter(reader, "PandoraPFOs"; selection=[:type, ...])
get_mcps  = create_getter(reader, "MCParticles"; selection=[:PDG, ...])
get_trks  = create_getter(reader, "SiTracks_Refitted"; selection=[:type])
get_links = create_getter(reader, "SiTracksMCTruthLink")

for evt in events
    # Select muons
    recps = unBoostCrossingAngle(get_recps(evt), -0.015rad)
    muons_all = filter(x -> abs(x.type) == 13, recps)            # select muons
    muons_sel = filter(x -> norm(x.momentum) > 20GeV, muons_all) # select p > 20 GeV

    # Energy resolution of Reconstructed muons
    mcps = unBoostCrossingAngle(get_mcps(evt), -0.015rad) # MC particles
    trks = get_trks(evt)                                  # Tracks
    links = get_links(evt)                                # Links Tracks<->MC part
    for muon in muons_sel
        for trk in muon.tracks
            nl = findfirst(x -> x.rec == trk, links)      # find the link index
            isnothing(nl) && continue
            push!(hresolu, muon.energy - links[nl].sim.energy)
        end
    end
end
plot(hresolu.hist, title=hresolu.title, cgrad=:plasma)
```



- Complete code to compare reconstructed energy for selected muons with truth

Multi-threaded Analysis

- Developed mini-framework to ensure thread safety
  - The user defines a **data structure** and an **analysis function**
- Each thread works on a subset of events using its own copy of the output data
- At the end, the results are 'summed' automatically

```
@with_kw mutable struct MyData <: AbstractAnalysisData
    nevents::Int64 =  0 # events processed
    μ1::Int64 = 0       # events with 1 muon
    μ2::Int64 = 0       # events with 2 muons
    mμμ::Int64 = 0      # resonance mass cut
    pμμ::Int64 = 0      # ...
    ...
end
```

```
function myanalysis!(data::MyData, reader, events)
  for evt in events
    data.nevents += 1

    recps = RootIO.get(reader, evt, "PandoraPFOs")
    recps = unBoostCrossingAngle(recps, -0.015rad)
    muons_all = filter(x -> abs(x.type) == 13, recps)
    muons_sel = filter(x -> norm(x.momentum) > 20GeV, muons_all)
    isos = coneIsolation(0.01, 0.5, muons_sel, recps)
    muons_iso = [x for (x,iso) in zip(muons_sel, isos) if iso < 0.25]

    # CUT 1: at least a lepton with at least 1 isolated one
    length(muons_sel) >= 1 && length(muons_iso) > 0 || continue
    data.μ1 += 1

    # CUT 2 :at least 2 OS leptons, and build the resonance
    length(muons_sel) >= 2 &&
      sum(muons_sel.charge) < length(muons_sel)|| continue
    data.μ2 += 1
    Zs = resonanceBuilder(91GeV, muons_sel)
    ...
 end
  return data
end
```

```
events = RootIO.get(reader, “events")
mydata = MyData()
do_analysis!(mydata, myanalysis!, reader, events;  mt=true)
```

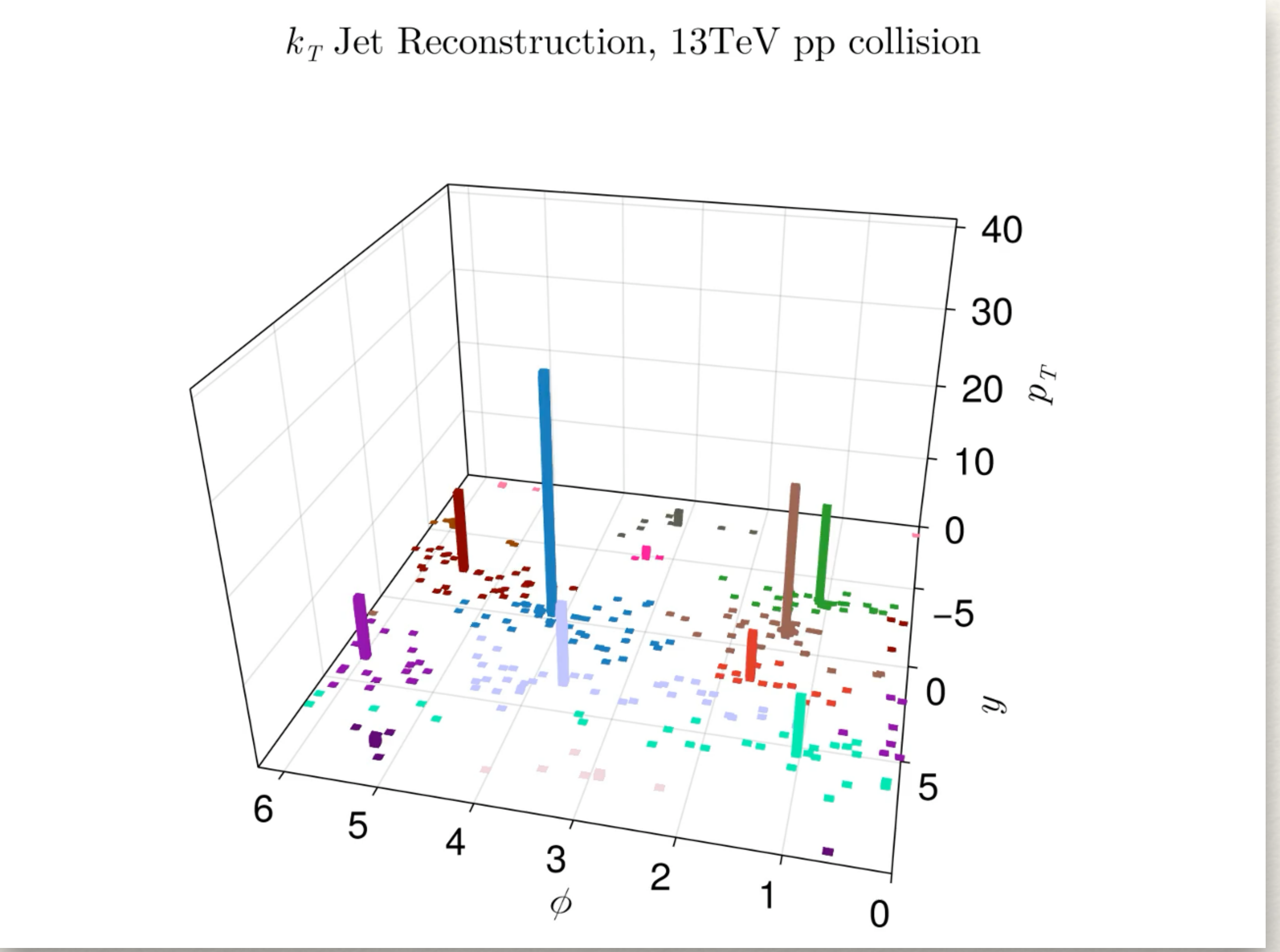
Analysis Tools: Histograms, Statistics, Minimizers, etc.

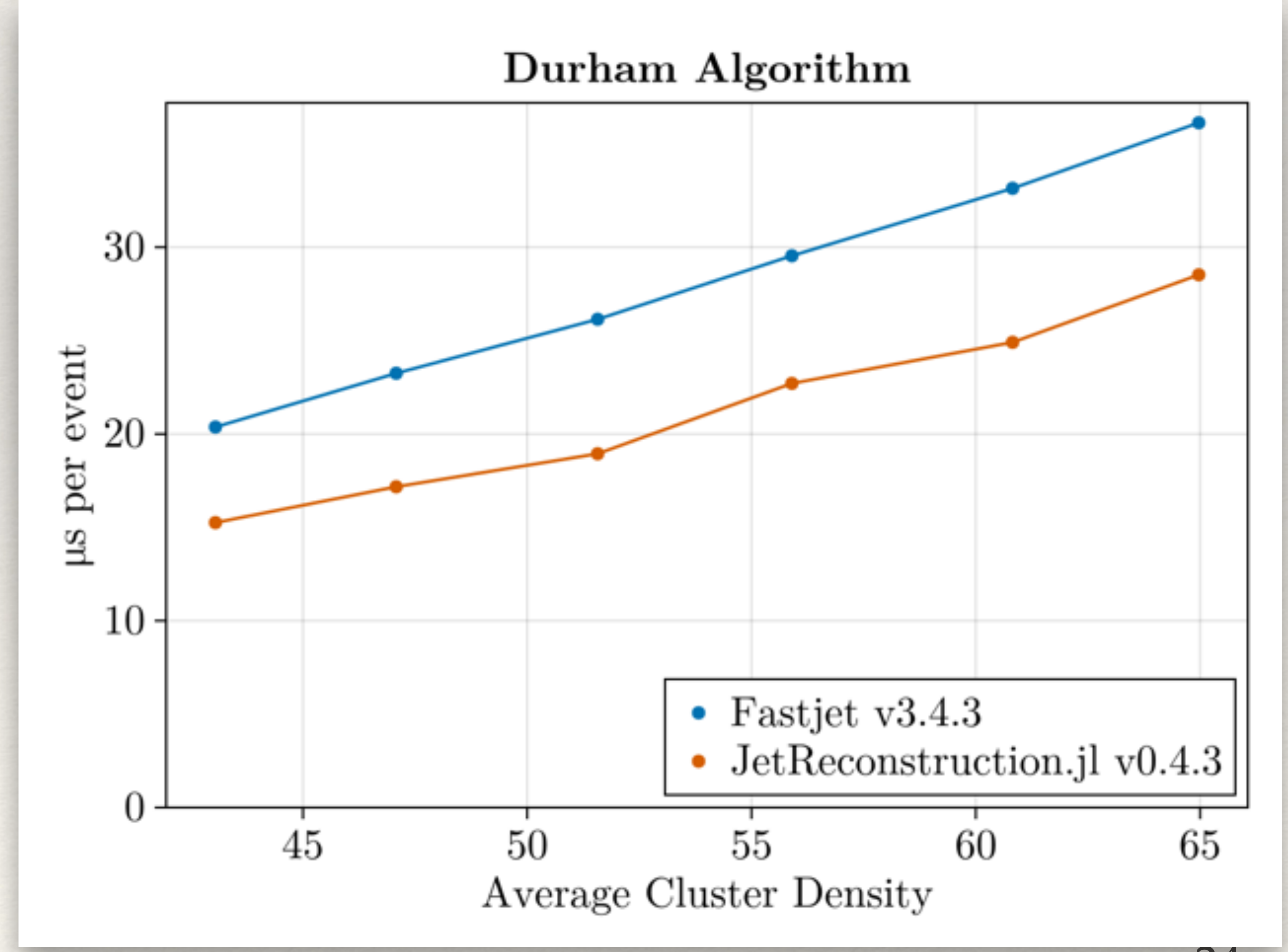
- FHist.jl - Fast, error-aware, and thread-safe 1D/2D/3D histograms
- Minuit2.jl - Starting the work to wrap C++ Minuit2
- ROOT.jl - Ongoing wrapping work to call ROOT from Julia, providing a user-friendly interface for TTrees (and RNTuple)

```julia
#Import the module.
using ROOT

# An alias for ROOT
const R = ROOT

# Create a ROOT histogram, fill random events, and fit it.
h = R.TH1D("h", "Normal distribution", 100, -5., 5.)
R.FillRandom(h, "gaus")

#Draw the histogram on screen
c = R.TCanvas()
R.Draw(h)

#Fit the histogram wih a normal distribution
R.Fit(h, "gaus")

#Save the Canvas in an image file
R.SaveAs(c, "demo_ROOT.png")

#Save the histogram and canvas demo_ROOT_out.root file.
f = R.TFile!Open("demo_ROOT_out.root", "RECREATE")
R.Write(h)
R.Write(c)
Close(f)
```

Jet Finding

- 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
- Better and more flexible ergonomic interfaces
  - Easier use of experiment specific types
- Nice integration with plotting libraries





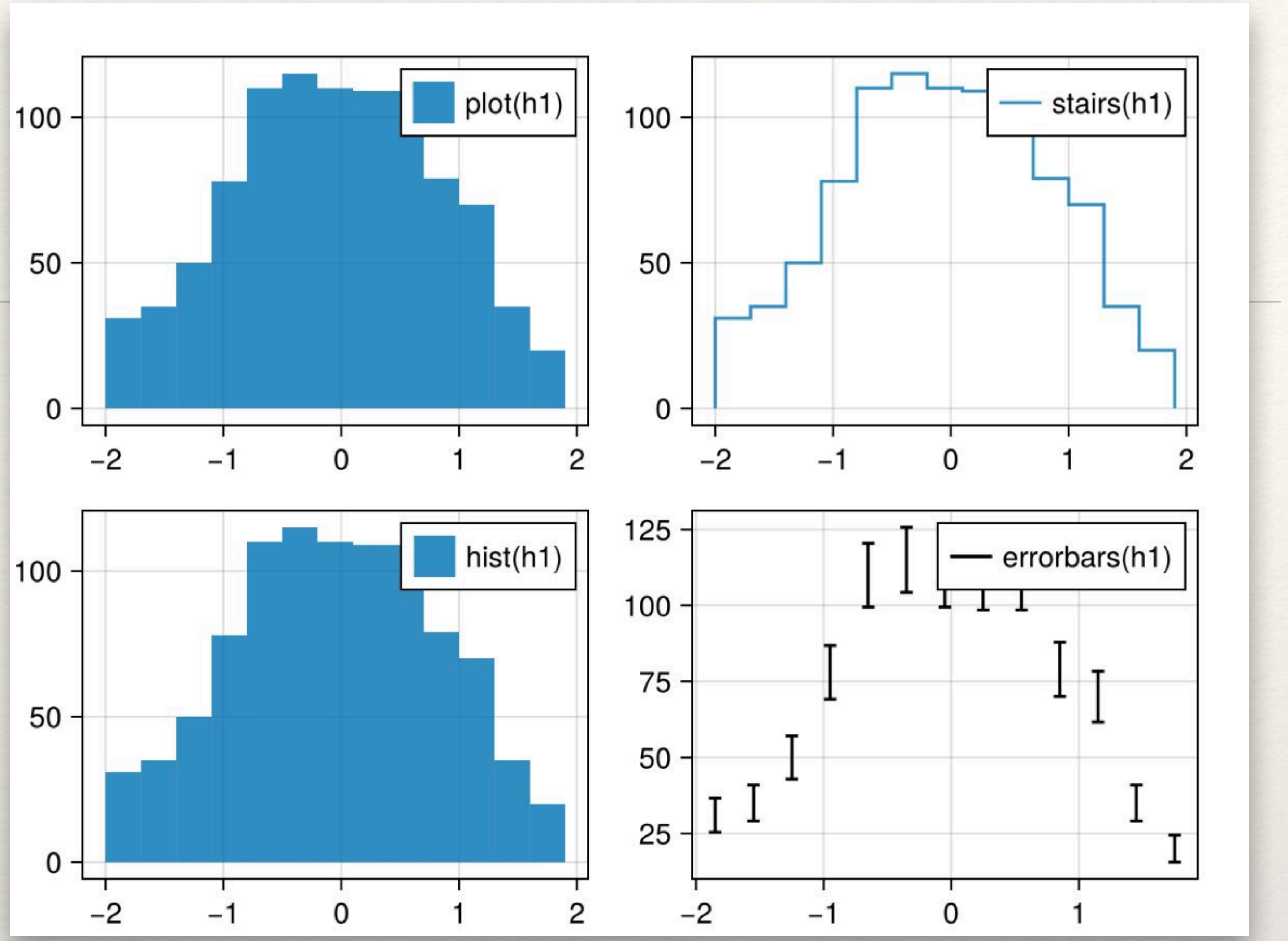
Plotting

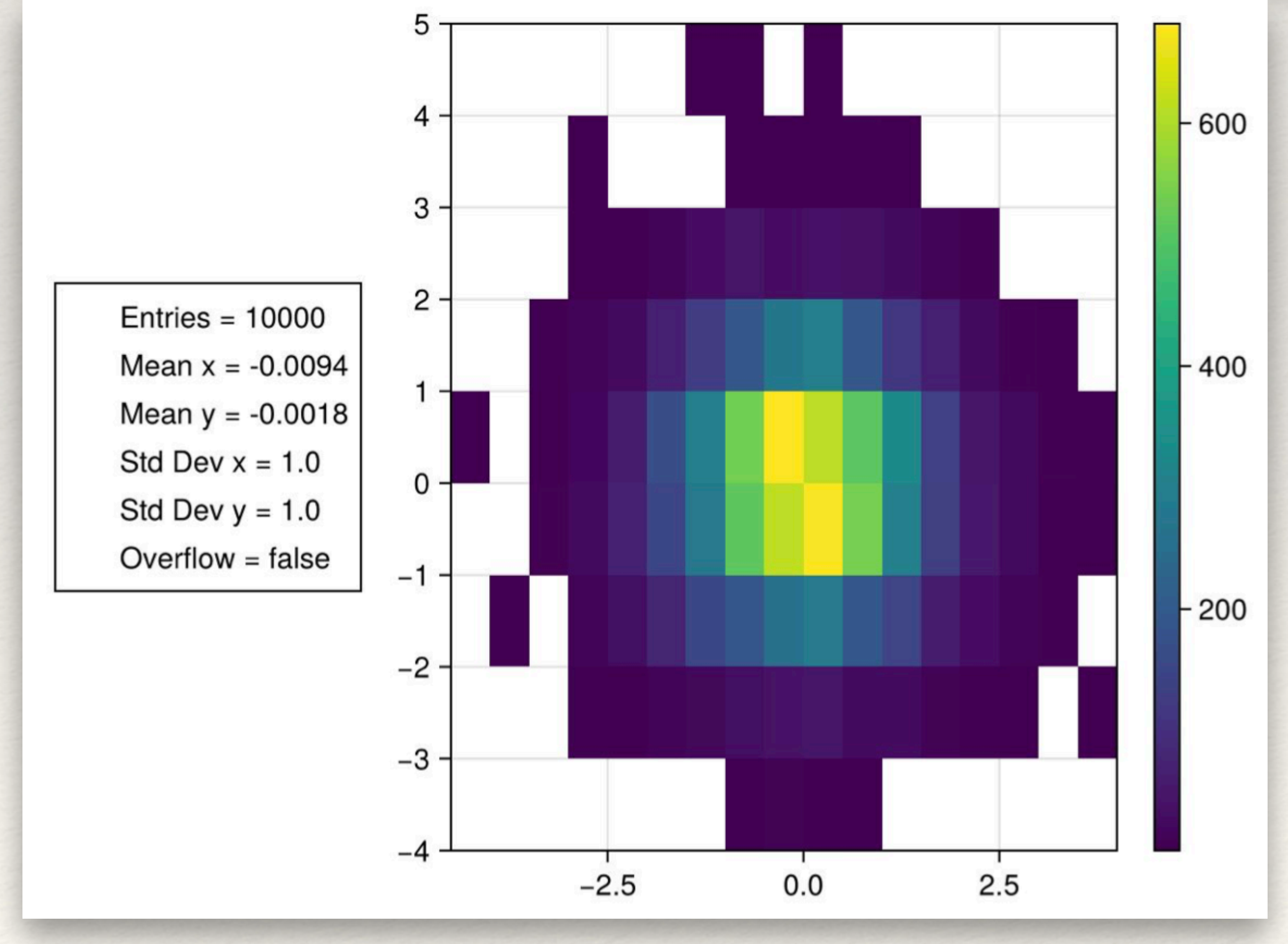
https://docs.juliaplots.org/stable/

https://docs.makie.org/dev/

Visualizations

- Plots.jl and Makie.jl are the standard visualization packages
  - Different backends (Cairo, OpenGL, etc.)
  - Makie is particularly good for 3D graphics
- They can be integrated (using the extension mechanism) very easily with FHist.jl for example

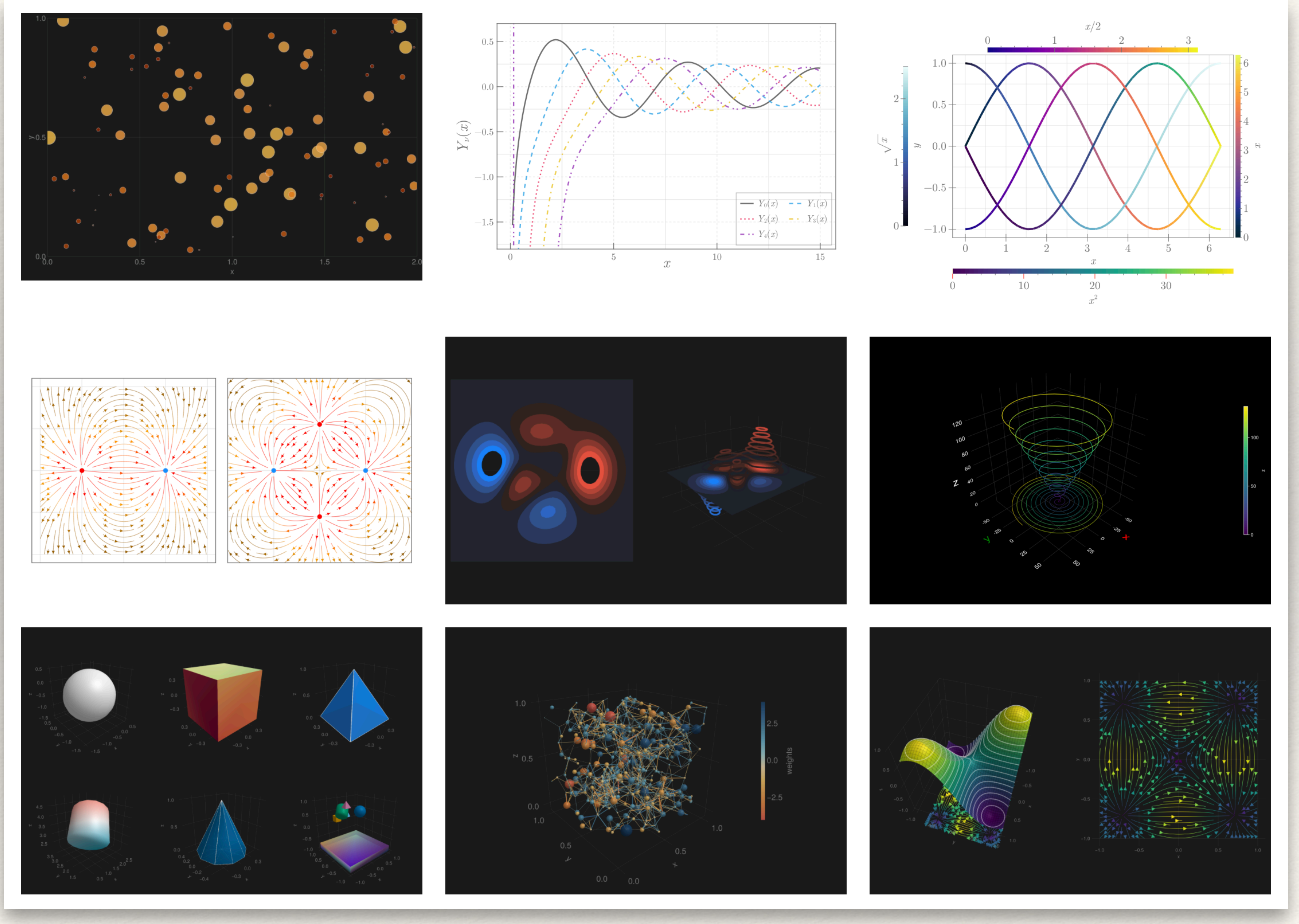





```
using FHist, Plots
h1 = Hist1D(randn(10^3); binedges = -2:0.3:2)
plot(h1)
```

Makie Data Visualization

- Makie is an interactive data visualization and plotting ecosystem
  - Available on Windows, Linux and Mac
  - Different back-ends
- With recent versions the time-to-first plot has been reduced dramatically



Using ROOT for data visualization

- While waiting to get HEP specific plotting in Julia, one possible strategy is to export final data (histos, dataframes, etc.) to ROOT to do the data presentation in there

```julia
using DataFrames
df = DataFrame(Zcand_m = Float32[],
               Zcand_recoil_m = Float32[],
               Zcand_q = Int32[],
               Zcand_recoil_θ = Float32[]), 0, 0)
for evt in events
  ...
  push!(df, (Zcand_m, Zcand_recoil_m, Zcand_q, Zcand_recoil_θ))
  ...
end
using Parquet2
Parquet2.writefile("m_H-recoil.parquet", data.df)
```

Julia

```python
import ROOT
import pandas as pd

pdf = pd.read_parquet('m_H-recoil.parquet') # engine='pyarrow'
rdf = ROOT.RDF.FromPandas(pdf)

h1 = rdf.Histo1D(("Zcand_m", "Z candidate mass
                 [GeV];N_{Events}", 100, 80, 100), "Zcand_m")
c1 = ROOT.TCanvas()
h1.Fit('gaus')
h1.Draw()
```

Python

Scaling Out

https://docs.julialang.org/en/v1/manual/multi-threading/

https://juliagpu.org/

https://github.com/JuliaParallel/Dagger.jl

MT, Parallel, GPUs, ...

- Built-in multi-threaded support (e.g. @threads, @spawn macros)
  - Good scalability with low number of cores, GC may became a limitation for many cores
- Julia is great for GPU programming
  - High-level language: higher productivity than vendor toolkits
  - Compiled language: enables native GPU programming

```
using Flux
m = Dense(10,5) |> gpu
x = rand(10) |> gpu
m(x)
```

ML

- Parallel framework — Dagger.jl
  - A framework for parallel computing across all kinds of resources, like CPUs and GPUs, and across multiple threads and multiple servers
  - Under active development, not yet production quality

Summary

- **Best-in-class Language:** Julia excels in scientific computing with high performance and ease of use, avoiding the need for multiple languages
- **EDM4hep Data:** The EDM4hep package offers efficient and ready-to-use tools for working with EDM4hep data files
- **Mature Ecosystem:** Julia's comprehensive tools and packages support advanced scientific analysis
- **HEP-Specific Needs:** Further development is still needed (e.g. low-level utilities, ROOT file writing, minimization tools, graphic recipes, etc.)
- **Ready Now:** Julia is productive and effective for analysis today.
- **Strong Community:** Active support via Slack (#HEP channel), Discourse, YouTube, and the HSF JuliaHEP activity group