EDM4hep.jl: Analysing EDM4hep files with Julia

Pere Mato[1,*]

[1]CERN, Switzerland

Abstract. EDM4hep aims to establish a standard event data model for the storage and exchange of event data in HEP experiments, thereby fostering collaboration across various experiments and analysis frameworks. The Julia package EDM4hep.jl is capable of generating Julia-friendly structures for the EDM4hep data model and reading event data files in ROOT format (either TTree or RNTuple) that are written by C++ programs, utilising the UnROOT.jl package. This paper explores the motivations behind the primary design choices of this package, such as the exclusive use of structure of arrays (SoA) to access the stored collections, which then empowers users to develop ergonomic data analyses using Julia's high-level concepts and functionality, while maintaining performance comparable to C++ programs. Several examples are given to illustrate how efficient data analysis can be achieved using high-level objects, eliminating the need to resort to flat n-tuples.

1 Introduction

The EDM4hep [1] data model is a set of data structures designed to be applicable across all future collider projects, accommodating diverse collision environments and various detector technologies currently under discussion. It serves as the foundational vocabulary for assembling large and complex data processing applications from modular algorithms. By providing a common data model, EDM4hep has the potential to foster collaboration among different experiments and analysis frameworks.

EDM4hep is built upon the PODIO toolkit [2, 3], which enables the automatic generation of C++ data structures with built-in navigation utilities and supports multiple input/output formats. The data structures are defined using a high-level YAML specification, ensuring adherence to the plain-old-data (POD) paradigm for efficient I/O operations across various back-end systems.

The Julia programming language has emerged as a promising candidate for scientific computing in the High Energy Physics (HEP) community [4]. However, its widespread adoption depends on the availability of HEP-specific foundational libraries that can support experiment-specific software. One of the most critical functionalities for HEP workflows is efficient access to data stored in ROOT files [5], the predominant format used by HEP experiments. The UnROOT.jl package [6] is a pure Julia package that was developed to enable lazy access to ROOT files, ensuring that only the required data is read. Remote file access is provided via the XRootD protocol [7] through the XRootD.jl [8] package.

*e-mail: pere.mato@cern.ch

Building on this foundation, the EDM4hep.jl package extends Julia's capabilities by providing seamless access to EDM4hep-formatted data. This article details the key design choices behind EDM4hep.jl, with a focus on optimising usability and efficiency while leveraging Julia's expressive syntax. We then demonstrate its performance in a multi-threaded data analysis scenario, followed by an evaluation of the results.

2 Design choices

From the YAML description file, we generate Julia types as immutable structures, as shown in Listing 1. Care has been taken to ensure that the resulting types have a fixed length (referred to as `isbits` in Julia terminology). This design choice allows for efficient use of these types in various environments, including GPU programming, and enables flexible data organization in both arrays of structures and structures of arrays.

A key challenge is encoding one-to-many relationships, and vector members within a fixed-length structure. This is achieved using `first` and `last` indices that refer to a separate array of `ObjectID`s which store references to the related objects, or directly to the values in the case of vector members. The parametric types `Relation{ED,N}` and `PVector{T,N}` have been introduced to support these PODIO constructions. In addition, to facilitate object creation, constructors are provided with keyword arguments that include reasonable default values.

Choosing immutable types adds some complexity when constructing the model in memory, as modifying relationships requires creating new instances. However, this approach significantly improves read performance, which is the primary use case. For instance, when adding parent relationships between `MCParticle` instances, new instances must be created and reassigned to the current variables, as demonstrated in Listing 2.

Listing 1. Example of generated Julia type for MCParticle

```julia
"""
struct MCParticle
Description: The Monte Carlo particle - based on the lcio::MCParticle.
"""
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 ...
  simulatorStatus::Int32            # status of the particle from the  ...
  charge::Float32                   # particle charge
  time::Float32                     # creation time of the particle  ...
  mass::Float64                     # mass of the particle in [GeV]
  vertex::Vector3d                  # production vertex of the particle
  endpoint::Vector3d                # endpoint of the particle in [mm]
  momentum::Vector3f                # particle 3-momentum at the production
  momentumAtEndpoint::Vector3f      # particle 3-momentum at the endpoint
  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
```

Listing 2. Building the model in memory and establishing a relation between two MCParticles

```
p1 = MCParticle(PDG=2212, mass=0.938, momentum=(0.0, 0.0, 7000.0))
p2 = MCParticle(PDG=2212, mass=0.938, momentum=(0.0, 0.0, -7000.0))
p3 = MCParticle(PDG=1, mass=0.0, momentum=(0.750, -1.569, 32.191))
p3, p1 = add_parent(p3, p1)
...
```

Reading EDM4hep files is handled using the UnROOT.jl package, which provides seamless support for both `TTree` and `RNTuple` formats, as well as multiple versions of PODIO. The data stored in these files consists exclusively of collections of specific data types (e.g., `ReconstructedParticles`, `Vertices`), each identified by a unique `collection name` and `collection ID`.

The goal of the reading operation is to obtain a `StructArray{DataType}` for each collection in every event. The `StructArray` type, provided by the standard Julia package StructArrays.jl [9], offers an efficient in-memory storage model, particularly well-suited for columnar operations. Reading into `StructArrays` is achieved using Julia's introspection capabilities and/or generated code based on the YAML description of the data model.

The creation of a `StructArray` is possible because the EDM4hep types are immutable and fixed size. This is shown in Figure 1. Once the PODIO collection is read as a `StructArray`, accessing and operating on columns is straightforward, as well as accessing rows, as it is shown in Listing 3

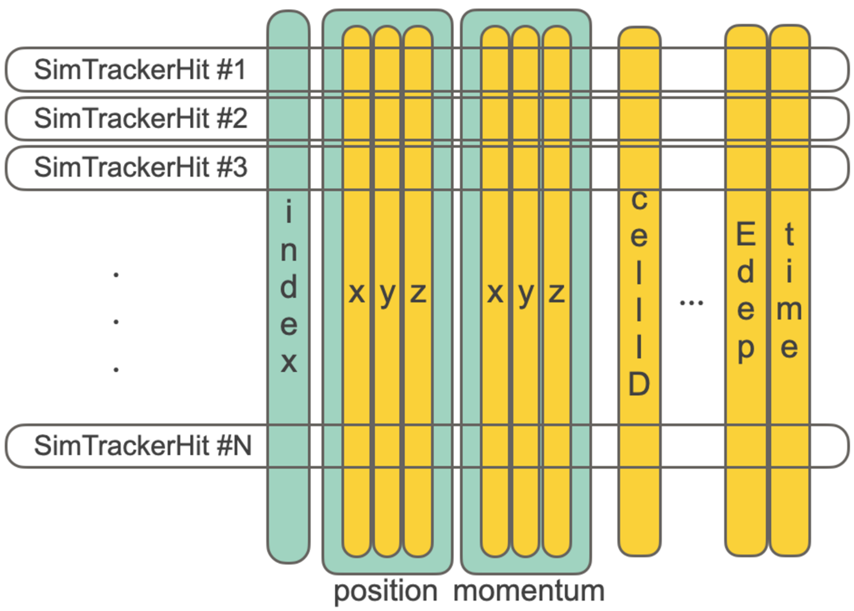

Figure 1. Mapping the columnar data in the ROOT file to construct a `StructArray{SimTrackerHit}` in memory that allows efficient columnar access, as well as by-row access by materialising the objects when required.

Listing 3. Easy access to rows and columns in containers

```
julia> mcps = <get the MCParticle collection>
julia> typeof(mcps[1])     # get the first element
```

```
MCParticle
julia> length(mcps.charge) # the column mcps.charge is a Vector{}
211
julia> mcps[1:2].momentum  # slicing
2-element StructArray(...) with eltype Vector3f:
(0.5000167,0.0,50.0)
(0.5000167,0.0,-50.0)
julia> sum(mcps[1:2].momentum) # columnar operations
(1.0000334,0.0,0.0)
```

3 Reading Interface

We have provided a simple interface for reading data files. The user creates a `Reader` object by providing a list of file names to be read, which can be either local or remote (e.g., root://eospublic.cern.ch/...). The `Reader` object is later used to lazily access the `event` and read the specific `collections` by name. Relationship intermediate collections will be read implicitly if required. Both the `event` and data collections provide an iterable and index-able interface. One-to-one relations can be followed using simple dot (.) notation. For example, in Listing 4, the relation between `SimulationHit` and `MCParticle` is simply traversed with `hit.mcparticle` returning the `MCParticle` from which to access any other property of the particle. Finally, the reading can be either sequential or multi-threaded as will be shown later.

Listing 4. Example reading the events and a couple of collections

```
using EDM4hep
using EDM4hep.RootIO
reader = RootIO.Reader("ttbar_edm4hep_digi.root")
events = RootIO.get(reader, "events")

evt = events[1];

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

#---Loop over hits----------------------------------------------------------
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 total charge $(sum(ps.charge))")
end
```

Listing 5. Output from previous listing

```
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-
...
```

4 Multi-Threaded Analysis

A mini-framework has been developed to perform some simple analysis using the EDM4hep data and ensuring that it also works in a multi-threaded environment to exploit the available processor cores to speed up the processing. The user needs to define:

- A data structure that will be used to collect the results of the analysis. This can either be simple items such as counters or histograms, or more complex data structures such as `DataFrames`. Each worker thread gets a private copy of this data structure, avoiding race conditions and eliminating the need for locks to protect the output data. At the end of the event loop processing, all thread-private instances of this data structure will be summed to produce the final result.
- An analysis function that will need to loop over the given subset of events and fill the private copy of the data structure. The user does an explicit loop over the subset of events provided. The chunking of the full set of events into subsets is done by the framework automatically using Julia's built-in functionality.

Listing 6 shows an example of an analysis in FCC using MC production data. The analysis function is straightforward since the loop over events is explicit and selection is clearly expressed using the `continue` statement and the standard `filter` function provided by Julia. To run this analysis in multi-threaded mode, the user simply needs to set the keyword argument `mt` to `true` when calling the `do_analysis!` function, which is provided in this mini-framework.

Listing 6. Simplified FCC analysis example to measure Higgs mass in the Z($\mu\mu$)H recoil measurement

```
#---User data structure to collect results---------------------------------
mutable struct MyData <: AbstractAnalysisData
    df::DataFrame
    pevts::Int64
    sevts::Int64
    MyData() = new(DataFrame(Zcand_m = Float32[], Zcand_recoil_m = Float32[],
          Zcand_q = Int32[]), 0, 0)
end

#---Analysis Function--------------------------------------------------------
function myanalysis!(data::MyData, reader, events)
  for evt in events
    data.pevts += 1                                  # count process events
    muIDs = RootIO.get(reader, evt, "Muon_objIdx") # get the ids of muons
    length(muIDs) < 2 && continue                    # skip if less than 2

    recps = RootIO.get(reader, evt, "ReconstructedParticles"; register=false)
    muons = recps[muIDs]            # use the ids to subset the reco particles

    sel_muons = filter(x -> pt(x) > 10GeV, muons)  # select the Pt of muons
    zed_leptonic = resonanceBuilder(91GeV, sel_muons)
    zed_leptonic_recoil = recoilBuilder(240GeV, zed_leptonic)

    if length(zed_leptonic) == 1  #  filter to have exactly one Z candidate
      Zcand_m        = zed_leptonic[1].mass
      Zcand_recoil_m = zed_leptonic_recoil[1].mass
      Zcand_q        = zed_leptonic[1].charge
      if 80GeV <= Zcand_m <= 100GeV   # select on mass of Z, push dataframe
        push!(data.df, (Zcand_m, Zcand_recoil_m, Zcand_q))
        data.sevts += 1               # count selected events
      end
    end
  end
```

```
  return data
end

#---Perform the analysis----------------------------------------------------
mydata = MyData()
do_analysis!(mydata, myanalysis!, reader, events; mt=true);
println("Processed events: $(mydata.pevts), selected: $(mydata.sevts)")

#---Plot the results--------------------------------------------------------
using Plots
histogram(mydata.df.Zcand_recoil_m, xlabel="GeV", title="Recoil mass plot",
    legend=false)
```

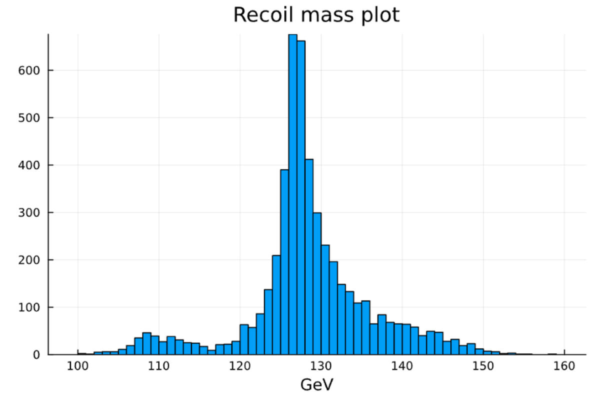

Figure 2. Mass plot of the selected Z candidates recoils obtained with the shown example

If not all attributes are required in the read types, users can define custom `getters` that retrieve only a subset of attributes, optimising the reading process, as shown in Listing 7. In this analysis, we only need the energy, momentum, charge, and mass from `ReconstructedParticle`, which represents a small fraction of the available data in the file. By selecting only the necessary attributes, we achieve a significant speedup in data access.

In future work, it may be beneficial to automatically determine the required attributes by statically analysing the user-provided analysis functions.

Listing 7. Same analysis making use of specialised `getters`

```
#---Use create_getter function to construct dynamically a getter function----
get_muIDs = RootIO.create_getter(reader, "Muon_objIdx")
get_recps = RootIO.create_getter(reader, "ReconstructedParticles"; selection
    [:energy,:momentum,:charge,:mass])

#---Analysis Function-------------------------------------------------------
function myanalysis!(data::MyData, reader, events)
  for evt in events
    muIDs = get_muIDs(evt)
    length(muIDs) < 2 && continue # skip if less than 2
    recps = get_recps(evt)
    muons = recps[muIDs] # use the ids to subset the reco particles
    ...
   end
end
```

5 Results

We have not yet conducted an exhaustive performance analysis of the Julia interface for reading EDM4hep data files. In this initial version of the package, our focus has been on usability and core functionality rather than full optimization. However, some preliminary performance measurements have been obtained.

For instance, when comparing Julia's access speed to the Python bindings available in PODIO, we observe a speedup of approximately 1000×. On the other hand, when comparing it to the FCCAnalysis framework [10], which utilises ROOT RDataFrame to read files with native ROOT C++, we observe comparable performance. Both systems can process between 10,000 and 20,000 events per second for the FCC recoil analysis.

However, the current multi-threaded implementation of EDM4hep.jl exhibits poor scalability. Performance peaks at 8 cores, which is relatively low given the large number of cores available on modern high-performance servers. The most likely cause is Julia's automatic garbage collection, which introduces non-negligible serial execution overhead, limiting potential speed-up. Optimising memory allocation in the Julia implementation could help push this limit further.

6 Conclusions

The first version of the EDM4hep.jl Julia package [11] is available in the Julia General Registry and ready for use. It can be installed via the Julia package manager Pkg with the command `julia -e 'import Pkg; Pkg.add("EDM4hep")'`. The only requirement is to have Julia installed on your system.

We have demonstrated how data analysis of EDM4hep files can be streamlined using high-level objects, providing a more intuitive and structured approach compared to flat n-tuples, all within a single, consistent, and fast programming language such as Julia. The overall performance is comparable to the C++ implementation and orders of magnitude better than the plain Python bindings. The multi-threaded analysis framework allows users to exploit the full potential of modern multi-core processors, although further optimization is needed to scale to larger numbers of cores.

References

[1] F. Gaede, T. Madlener, P. Declara Fernandez, G. Ganis, B. Hegner, C. Helsens, A. Sailer, G. A.Stewart and V. Voelkl, "EDM4hep - a common event data model for HEP experiments," PoS **ICHEP2022**, 1237 doi:10.22323/1.414.1237

[2] F. Gaede, B. Hegner and P. Mato, "PODIO: An Event-Data-Model Toolkit for High Energy Physics Experiments," J. Phys. Conf. Ser. **898** (2017) no.7, 072039 doi:10.1088/1742-6596/898/7/072039

[3] J. M. Carceller, F. Gaede, G. Ganis, B. Hegner, C. Helsens, T. Madlener, A. Sailer, G. A. Stewart and V. Volkl, "Towards podio v1.0 - A first stable release of the EDM toolkit," EPJ Web Conf. **295** (2024), 06018 doi:10.1051/epjconf/202429506018 [arXiv:2312.08206 [hep-ex]].

[4] J. Eschle, T. Gál, M. Giordano, P. Gras, B. Hegner, L. Heinrich, U. Hernandez Acosta, S. Kluth, J. Ling, P. Mato, *et al.* "Potential of the Julia Programming Language for High Energy Physics Computing," Comput. Softw. Big Sci. **7** (2023) no.1, 10 doi:10.1007/s41781-023-00104-x [arXiv:2306.03675 [hep-ph]].

[5] I. Antcheva, M. Ballintijn, B. Bellenot, M. Biskup, R. Brun, N. Buncic, P. Canal, D. Casadei, O. Couet, V. Fine, *et al.* “ROOT: A C++ framework for petabyte data storage, statistical analysis and visualization,” Comput. Phys. Commun. **182** (2011), 1384-1385 doi:10.1016/j.cpc.2011.02.008

[6] J. Ling and T. Gál, “High-performance end-user analysis in pure Julia programming language,” EPJ Web Conf. **295** (2024), 06012 doi:10.1051/epjconf/202429506012

[7] The XRootD project, available at: https://xrootd.org. Accessed: 25 Feb 2025.

[8] The XRootD.jl package, available at https://github.com/JuliaHEP/XRootD.jl. Accessed: 25 Feb 2025.

[9] StructArrays.jl: Efficient implementation of struct arrays in Julia, available at https://github.com/JuliaArrays/StructArrays.jl. Accessed: 25 Feb 2025.

[10] Common analysis framework for the Future Circular Collider, available at https://github.com/HEP-FCC/FCCAnalyses. Accessed: 25 Feb 2025.

[11] Julia interface to EDM4hep, available at https://github.com/JuliaHEP/EDM4hep.jl. Accessed: Accessed: 25 Feb 2025.