The Journal of Open Source Software

UnROOT: an I/O library for the CERN ROOT file format written in Julia

Tamás Gál [1,2], **Jerry (Jiahong) Ling** [3], **and Nick Amin** [4]

1 Erlangen Centre for Astroparticle Physics **2** Friedrich-Alexander-Universität Erlangen-Nürnberg **3** Harvard University **4** University of California, Santa Barbara

DOI: 10.21105/joss.04452

Software

- Review
- Repository
- Archive

Editor: Daniel S. Katz

Reviewers:

- @PerilousApricot
- @jpata

Submitted: 03 June 2022
Published: 18 August 2022

Summary

`UnROOT.jl` is a pure Julia implementation of CERN's ROOT (Brun & Rademakers, 1997) file I/O (`.root`) software, which is fast and memory-efficient, and composes well with Julia's high-performance iteration, array, and multi-threading interfaces.

Statement of Need

The High-Energy Physics (HEP) community, especially in data analysis, has been facing the two-language problem for a long time. Often, physicists would start prototyping with a `Python` front-end which glues to a `C/C++/Fortran` back-end. Soon they would hit a task that could be challenging to express in columnar (i.e., "vectorized") style, a type of problem that is normally tackled with libraries like `numpy` (Harris et al., 2020) or pandas (The pandas development team, 2020). This usually would lead to them either writing `C++` kernels and interfacing them with `Python`, or porting the prototype to `C++` and starting to maintain two code bases including the wrapper code. Specific to HEP, `AwkwardArray` (Pivarski et al., 2018) can be seen as a compromise between the two solutions, where the user writes in a special columnar style that has some flexibility for addressing the jaggedness of HEP data.

All traditional options represent increasing engineering effort for authors and users, often in multiple programming languages. Many steps of this process are critical, such as identifying bottlenecks and creating an architecture that is simultaneously performant and maintainable while still being user-friendly and logically structured. Using a `Python` front-end and dancing across language barriers also hinders the ability to parallelize tasks down to event level, as the existing usage often relies on chunk or even file level parallelization. Finally, newer techniques such as automatic differentiation also work more smoothly without language barriers, allowing physicists to develop algorithms. With Julia's active auto diff community [1], we expect `UnROOT.jl` to be one of the cornerstones for physicists.

`UnROOT.jl` attempts to solve all of the above by choosing Julia, a high-performance language with a simple and expressive syntax (Bezanson et al., 2017). Julia is designed to solve the two-language problem in general. This has also been studied for HEP specifically (Stanitzki & Strube, 2021). Analysis software written in Julia can freely escape to a `for-loop` whenever vectorized-style processing is not flexible enough, without any performance degradation. At the same time, `UnROOT.jl` transparently supports multi-threading and multi-processing by providing data structures that are a subtype of `AbstractArray`, the built-in abstract type for array-like objects, which allows easy interfacing with array-routines from other packages, thanks to multiple dispatch, one of the main features of Julia.

[1] https://juliadiff.org/

Features and Functionality

The ROOT data format is flexible and mostly self-descriptive. Users can define their own data structures (C++ classes) that derive from ROOT classes and serialise them into directories, trees, and branches. The information about the deserialisation is written to the output file (therefore, it's self-descriptive) but there are some basic structures and constants needed to bootstrap the parsing process. One of the biggest advantages of the ROOT data format is the ability to store jagged structures like nested arrays of structs with different sub-array lengths. In high-energy physics, such structures are preferred to represent, for example, particle interactions and detector responses as signals from different hardware components, combined into a tree of events.

UnROOT.jl understands the core structure of ROOT files, and is able to decompress and deserialize instances of the commonly used TH1, TH2, TDirectory, TTree, etc. ROOT classes. All basic C++ types for TTree branches are supported as well, including their nested variants. Additionally, UnROOT.jl provides a way to hook into the deserialisation process of custom types where the automatic parsing fails. At the time of this article, UnROOT is already being used successfully in the data analysis of the KM3NeT neutrino telescope. And just like RDataFrame, it can be directly used on "NTuple" TTree such as the NANOAOD format used by the CMS collaboration (Ehatäht, 2020).

Opening and loading a TTree lazily, i.e., without reading the whole data into memory, is simple:

```
julia> using UnROOT

julia> f = ROOTFile("test/samples/NanoAODv5_sample.root")
ROOTFile with 2 entries and 21 streamers.
test/samples/NanoAODv5_sample.root
   Events
      "run"
      "luminosityBlock"
      "event"
      "HTXS_Higgs_pt"
      "HTXS_Higgs_y"
      ...

julia> mytree = LazyTree(f, "Events", ["Electron_dxy", "nMuon", r"Muon_(pt|eta)$"])
 Row   Electron_dxy     nMuon   Muon_eta         Muon_pt
       Vector{Float32}  UInt32  Vector{Float32}  Vector{Float32}

 1     [0.000371]       0       []               []
 2     [-0.00982]       2       [0.53, 0.229]    [19.9, 15.3]
 3     []               0       []               []
 4     [-0.00157]       0       []               []
       ...
```

As seen in the above example, the entries in the columns are multi-dimensional and jagged. The LazyTree object acts as a table that suports sequential or parallel iteration, selections, and filtering based on ranges or masks, and operations on whole columns:

```
for event in mytree
    # ... Operate on event
end

Threads.@threads for event in mytree # multi-threading
    # ... Operate on event
end
```

```
mytree.Muon_pt # a column as a lazy vector of vectors
```

The `LazyTree` is designed as `<: AbstractArray`, which makes it compose well with the rest of the Julia ecosystem. For example, syntactic loop fusion [2] and Query-style tabular manipulations provided by packages like `Query.jl` [3] without any additional code support just work out-of-the-box.

For example, the following code will only iterate through the tree once to find all events with exactly two muons and two electrons, due to loop fusion:

```
# t <: LazyTree
findall(@. t.nMuon==2 & t.nElectron==2)
```

And query-style filtering can be done with no code addition from `UnROOT.jl`'s end thanks to Julia's composability due to multiple dispatch [4]:

```
julia> using Query, DataFrame

julia> @from evt in mytree begin
         @where length(evt.Jet_pt) > 6
         @let njets=length(evt.Jet_pt)
         @let njets40=sum(evt.Jet_pt.>40)
         @select {njets=njets, njets40, evt.MET_pt}
         @collect DataFrame
       end
```

Synthetic Benchmark against C++ ROOT

Benchmark against various C++ ROOT solution can be found in our benchmark repo[5]. Here we summarize the results:

Single-threaded composite benchmark

Language	Cold Run	Warmed Run
Julia	20.58 s	19.81 s
PyROOT RDF	40.21 s	N/A
Compiled C++ ROOT Loop	28.16 s	N/A
Compiled RDF	19.82 s	N/A

4-threaded composite benchmark

Language	Cold Run	Warmed Run
Julia	5.46 s	5.07 s
PyROOT RDF	N/A	N/A
Compiled C++ ROOT Loop	N/A	N/A

[2] https://julialang.org/blog/2017/01/moredots/
[3] https://github.com/queryverse/Query.jl
[4] https://www.youtube.com/watch?v=kc9HwsxE1OY
[5] https://github.com/Moelf/UnROOT_RDataFrame_MiniBenchmark#single-threaded-composite-benchmark

Language	Cold Run	Warmed Run
Compiled RDF	5.64 s	N/A

Usage Comparison with Existing Software

This section focuses on comparison with other existing ROOT I/O solutions in the Julia universe. However, one honorable mention is `uproot` (Pivarski et al., 2021), which is a purely Python-based ROOT I/O library that played (and still plays) an important role in the development of `UnROOT.jl` as it was at the time of this article the most complete and best documented ROOT I/O implementation.

- `UpROOT.jl` is a wrapper for the aforementioned `uproot` Python package and uses `PyCall.jl` [6] as a bridge, which means that it relies on `Python` as a glue language. In addition to that, `uproot` itself utilises the C++ library `AwkwardArray` (Pivarski et al., 2018) to efficiently deal with jagged data in `ROOT` files. Most of the features of `uproot` are available in the Julia context, but there are intrinsic performance and usability drawbacks due to the three-language architecture.
- `ROOT.jl` [7] is one of the oldest Julia `ROOT` packages. It uses C++ bindings to directly wrap the `ROOT` framework and therefore is not limited ot I/O. Unfortunately, the `Cxx.jl` [8] package that is used to generate the C++ glue code does not support Julia 1.4 or later. The multi-threaded features are also limited.

Conclusion

`UnROOT.jl` is an important package in high-energy physics and related scientific fields where the `ROOT` dataformat is established, since the ability to read and parse scientific data is certainly the first mandatory step to open the window to a programming language and its package ecosystem. `UnROOT.jl` has demonstrated tree processing speeds at the same level as the `C++ ROOT` framework in per-event iteration as well as the Python-based `uproot` library in chunked iteration.

References

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, *59*(1), 65–98. https://doi.org/10.1137/141000671

Brun, R., & Rademakers, F. (1997). ROOT: An object oriented data analysis framework. *Nucl. Instrum. Meth. A*, *389*, 81–86. https://doi.org/10.1016/S0168-9002(97)00048-X

Ehatäht, K. (2020). NANOAOD: a new compact event data format in CMS. *EPJ Web Conf.*, *245*, 06002. https://doi.org/10.1051/epjconf/202024506002

Harris, C. R., Millman, K. J., Walt, S. J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M. H. van, Brett, M., Haldane, A., Río, J. F. del, Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Pivarski, J., Osborne, I., Ifrim, I., Schreiner, H., Hollands, A., Biswas, A., Das, P., Roy Choudhury, S., & Smith, N. (2018). *Awkward array* (Version 1.9.0rc4) [Computer software]. Zenodo. https://doi.org/10.5281/zenodo.6522027

[6] https://github.com/JuliaPy/PyCall.jl
[7] https://github.com/JuliaHEP/ROOT.jl
[8] https://github.com/JuliaInterop/Cxx.jl

Pivarski, J., Schreiner, H., Smith, N., Burr, C., Kalinkin, D., Stark, G., Hartmann, N., Davis, D., O'Neil, R., Novak, A., Greiner, B., Stanislaus, B., ChristopheRappold, Deaconu, C., Cervenkov, D., Rübenach, J., Bendavid, J., Lieret, K., Peresano, M., … Held, A. (2021). *Scikit-hep/uproot4: 4.1.3* (Version 4.1.3) [Computer software]. Zenodo. https://doi.org/10.5281/zenodo.5539722

Stanitzki, M., & Strube, J. (2021). Performance of Julia for high energy physics analyses. *Computing and Software for Big Science*, *5*(1), 1–11. https://doi.org/10.1007/s41781-021-00053-3

The pandas development team. (2020). *Pandas* (latest) [Computer software]. Zenodo. https://doi.org/10.5281/zenodo.3509134