RootIO.JL

A I/O package for ROOT.JL

Yash Solanki[1] Philippe Gras[2] Pere Mato Vila[3]

[1]Indian Institute of Technology, Delhi
[2]CEA/IRFU, Université Paris-Saclay
[3]CERN, EP-SFT

Table of contents

Introduction

Motivation

ROOT TTree support in Julia

UnROOT.jl: excellent to read TTrees, but does not support writing.

UpROOT.jl (wrapper to Uproot Python package): read/write support, but performance limited by Python.

ROOT.JL: Julia interface to the C++ ROOT data analysis framework. It provides the C++ API with one-to-one mapping. **The C++ interface does not map well in Julia for TTree read and write**.

Need for an easy interface to write TTrees in Julia

Writing TTree with ROOT.jl

Write support has a complex syntax:

```
using ROOT
f = ROOT.TFile!Open("test1.root", "RECREATE")
tree = ROOT.TTree("tree", "tree")
age = Ref(0) # <- A REF
Branch(tree, "name", Ptr{Cvoid}(), "s/C") # <- POINTER
Branch(tree, "age", age)
for  (name, age[]) in [("john", 18), ("Emma", 3), ("Eleanor", 21)]
    SetBranchAddress(tree, "name", convert(Ptr{Cchar}, pointer(name)))
    GC.@preserve name age Fill(tree) # UNDERLYING USE OF PTR REQUIRES @preserve
end
Scan(tree) # Display the tree content
Write(tree)
Close(f)
```

Listing 1: Writing to a root file using ROOT.JL

It requires manipulation of references and pointers, which is not in line with the design practices of Julia programs.

RootIO.JL

RootIO.JL: Write interface for ROOT.JL

RootIO.jl provides a user-friendly high-level interface for TTree write support. Built as a layer on top of ROOT.jl.

The previous example simplifies to:

```
using ROOT, RootIO
f = ROOT.TFile!Open("test1.root", "RECREATE")
tree = TTree(f, "tree", "tree", name = ["john", "Emma", "Eleanor"],
                  age = [18, 3, 21])
Scan(tree) # Display the tree content
Write(tree)
Close(f)

```

Listing 2: Writing to a root file using RootIO.JL

RootIO includes support for most of the standard Julia primitive types, character strings (String), and vectors of elements of these types.

Supported data types

The following table summarizes the supported types:

Type	Description
String	A character string
Int8	An 8-bit signed integer
UInt8	An 8-bit unsigned integer
Int16	A 16-bit signed integer
UInt16	A 16-bit unsigned integer
Int32	A 32-bit signed integer
UInt32	A 32-bit unsigned integer
Float32	A 32-bit floating-point number
Float64	A 64-bit floating-point number
Int64	A long signed integer, stored as 64-bit
UInt64	A long unsigned integer, stored as 64-bit
Bool	A boolean
StdVector{T}[1]	A vector of elements of any of the above types stored as std::vector
Vector{T}	A vector of elements of any of the above types stored as a C-array[2]

Table 1: Supported data types and descriptions

Examples

Storing scalars

To initialize a TTree with branches, call the RootIO.TTree method and input the branch name along with the datatype stores by the branch.

```
1
2  using RootIO, ROOT
3  using Random
4
5  # Create a ROOT file
6  file = ROOT.TFile!Open("example.root", "RECREATE")
7
8  # Create the tree
9  tree = RootIO.TTree(file, "tree", "My Tree", pt = Float64, eta = Float64, phi = Float64)
10
11 # Fill the tree with random values
12 for i in 1:10
13  Fill(tree, (pt = 100*randexp(), eta = 5*randn(), phi = π*rand()))
14 end
15
16 # Display tree content
17 Scan(tree)
18
19 # Save the tree and close the file
20 RootIO.Write(tree)
21 ROOT.Close(file)
22
```

Listing 3: Storing scalars to root file using RootIO.JL

Storing object as structs

We can store rows that have a composite type (a structure):

```
using RootIO, ROOT
using Random
mutable struct Event
    nparts::Int32
    pt::StdVector{Float64}
    eta::StdVector{Float64}
    phi::StdVector{Float64}
end
Event()  = Event(0., StdVector{Float64}(), StdVector{Float64}(), StdVector{Float64}())
f = ROOT.TFile!Open("example.root", "RECREATE")
tree = RootIO.TTree(f, "tree", "My Tree", Event)
e = Event()
for i in 1:10
    e.nparts = rand(Vector{Int32}(1:10))
    e.pt = StdVector(100*randexp(e.nparts))
    e.eta = StdVector(5*randn(e.nparts))
    e.phi = StdVector( π*rand(e.nparts))
    RootIO.Fill(tree, e)
end
RootIO.Write(tree)
Close(f)

```

Listing 4: Storing structs to root file using RootIO.JL

Storing Tables and DataFrames

```
1
2  using RootIO, ROOT
3  using Random
4  using DataFrames
5
6  # Create the dataframe. Broadcasting is used to vectorize the event/row generation
7  nevents = 10
8  nparts = rand(1:10, nevents)
9
10 # We can use any container table type compliant with the `Tables.jl` interface.
11 table = DataFrame(nparts = nparts,
12                   pt   = StdVector.(100 .* randexp.(nparts)),
13                   eta = StdVector.(  5 .* randn.(nparts)),
14                   phi = StdVector.( π .* randn.(nparts)))
15
16 # Create the ROOT file
17 f = ROOT.TFile!Open("example.root", "RECREATE")
18
19 # Create the tree and fill it with the dataframe contents
20 tree = RootIO.TTree(f, "tree", "My Tree", table)
21
22 # Display tree contents
23 Scan(tree)
24
25 # Save the tree and close the file
26 RootIO.Write(tree)
27 Close(f)
28
```

Listing 5: Storing a DataFrame to root file using RootIO.JL

Conclusion and future development

Conclusion and future development

A new package, RootIO.JL, that provides an easy-to-use TTree write support.

- General repository registration submitted. In the meantime, use `add https://github.com/JuliaHEP/RootIO.jl`.

Future development roadmap

- Support for nested structs and sub-branches, which will allow writing of arbitrary compositions of objects;
- Read support for completeness;
- Support for RNTuple.

Thank you!