Analyse Research Open Data with ROOT's RDataFrame

ATLAS Open Data Tutorial

27 November 2025

Eirik Gramstad

First: Get the notebook

Notebook on github

```
git fetch --all
git checkout RDFNotebook
13-TeV-examples/rdataframe/RDataFrameOnResearchOD.ipynb
```

ROOT in Swan

- Do nothing, and carry on

```
1 isSwan = True
2 isColab = False
3 isBinder = False
4 isGalaxy = False
```

Execute the first 6 cells **now!**
(may take some time if you haven't used ROOT in Swan before)

Get ROOT in Google Colab

```
1 isSwan = False
2 isColab = True
3 isBinder = False
4 isGalaxy = False
```

1. Get the precompiled ROOT for Google Colab

```
[1] ✓ 30s  !wget -q --show-progress https://github.com/MohamedElashri/ROOT/releases/download/root-v6.32.04-python3.12/root_v6.32.04_Ubuntu_Python3.12.zip
           root_v6.32.04_Ubunt 100%[===================>] 279.35M  10.5MB/s    in 29s
```

2. Unzip it

```
[2] ✓ 9s  !unzip -q /content/root_v6.32.04_Ubuntu_Python3.12.zip
```

3. Get some other needed libraries

```
[3] ✓ 6s  !wget http://archive.ubuntu.com/ubuntu/pool/main/o/openssl/libssl1.1_1.1.1f-1ubuntu2_amd64.deb
          !sudo dpkg -i libssl1.1_1.1.1f-1ubuntu2_amd64.deb
          !rm -f libssl1.1_1.1.1f-1ubuntu2_amd64.deb
[4] ✓ 11s !sudo apt install xrootd-client xrootd-server python3-xrootd
```

Tip!

If you'd like to avoid having to download the zip-file all the time: save it to your google drive, mount your drive and get it from there

```
# Load the Drive helper and mount
from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive
```

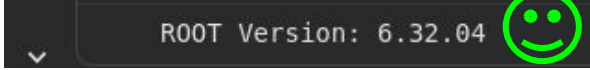
Get ROOT in Google Colab

4. Make sure ROOT is found by the system

5. Check that everything's working!

```
[9]
✓ 0s
import ROOT
print(f"ROOT Version: {ROOT.gROOT.GetVersion()}")

ROOT Version: 6.32.04
```



```
[5]
✓ 0s
import sys
import ctypes

# Step 1: Append ROOT paths to Python
sys.path.append("root_build/")
sys.path.append("root_build/bin/")
sys.path.append("root_build/include/")
sys.path.append("root_build/lib/")

# Step 2: Load the required shared libraries (.so files)
ctypes.cdll.LoadLibrary("root_build/lib/libCore.so")
ctypes.cdll.LoadLibrary("root_build/lib/libThread.so")
ctypes.cdll.LoadLibrary("root_build/lib/libTreePlayer.so")

print("ROOT Libraries Loaded Successfully!")

ROOT Libraries Loaded Successfully!
```

6. We also need *atlasopenmagic*

```
[10]
✓ 6s
# First we install atlasopenmagic into our SWAN environment
# Notice that we need --user to avoid trying to install the package in a
# read-only file system This is a problem unique to SWAN; on binder or colab you
# won't need --user, but it doesn't hurt
%pip install --user atlasopenmagic

Collecting atlasopenmagic
  Downloading atlasopenmagic-1.7.0-py3-none-any.whl.metadata (5.8 kB)
Requirement already satisfied: pyyaml in /usr/local/lib/python3.12/dist-packages (from atlasopenmagic) (6.0.3)
Requirement already satisfied: requests in /usr/local/lib/python3.12/dist-packages (from atlasopenmagic) (2.32.4)
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-packages (from atlasopenmagic) (4.67.1)
Requirement already satisfied: charset_normalizer<4,>=2 in /usr/local/lib/python3.12/dist-packages (from requests->atlasopenmagic) (3.4.4)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.12/dist-packages (from requests->atlasopenmagic) (3.11)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.12/dist-packages (from requests->atlasopenmagic) (2.5.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.12/dist-packages (from requests->atlasopenmagic) (2025.10.5)
Downloading atlasopenmagic-1.7.0-py3-none-any.whl (21 kB)
Installing collected packages: atlasopenmagic
Successfully installed atlasopenmagic-1.7.0

[11]
✓ 0s
# Now we have to do a little bit of work to make sure that atlasopenmagic is
# available in our python path This is because SWAN by default does not include
# the local package installation area in the PYTHONPATH Again, this is not
# necessary on binder or colab - there you can remove these lines if you like,
# though they don't do any harm
import sys
import os
import re
import json
sys.path += [ f'{os.environ["HOME"]}/.local/lib/python{sys.version_info.major}.{sys.version_info.minor}/site-packages' ]

import atlasopenmagic as atom
```

Execute the first 6 cells **now!**
(may take some time if you haven't used ROOT in Swan before)

Content, framework and notebook

1. Using RDataFrame to inspect the Research Open Data
   - Simple example
2. Using RDataFrame in a simplified analysis workflow
   - Simplified example, but shows most of the ingredients needed to perform a complete analysis
3. Converting from RDataFrame to NumPy
   - Slow, and unnecessary
4. Doing machine learning directly from RDataFrame
   - Requires later versions (>6.38) of ROOT



bleeding edge

Get the latest notebook version:

[Notebook on github]

```
git fetch –all
git checkout RDFNotebook
13-TeV-examples/rdataframe/RDataFrameOnResearchOD.ipynb
```

Introduction

- What is RDataFrame?
  - a modern, high-level interface for analysis of data stored in **TTree**, **CSV** and other data formats, in **C++** or **Python**
    - multi-threading and other low-level optimisations allow users to exploit all the resources available on their machines completely transparently
  - Calculations are expressed in terms of **functional chain of actions and transformations**, **RDataFrame takes care of their execution**
- Links
  - Main documentation
  - Many nice tutorials

Introduction

- Compare with a simple example in awkward array

Awkward array (from live demo)

```
# Perform the cuts for each data entry in the tree
for data in tree.iterate(variables, library="ak"):

    photon_isTightID = data['photon_isTightID']
    data = data[cut_photon_reconstruction(photon_isTightID)]

    photon_pt = data['photon_pt']
    data = data[cut_photon_pt(photon_pt)]

    # Append data to the whole sample data list
    sample_data.append(data['mass'])
```

The RDataFrame counterpart

```
df = df.Define("photon_isTightID","AnalysisPhotonsAuxDyn.DFCommonPhotonsIsEMTight")
df = df.Filter('ROOT::VecOps::Sum(photon_isTightID)>=2',"Two tight photons")
df = df.Define("photon_pt","AnalysisPhotonsAuxDyn.pt/1000.")
df = df.Filter("Sum(photon_pt>50)>0 && Sum(photon_pt>30)>1","Pt > 50, 30 GeV")
```

No explicit **loop over events**!

RDataFrame takes care of the execution (either single threaded, multi threaded or multi processed)

No need to store the result in any (possible large) vector!

How to define an RDataFrame

- Let's start with a simple example using only one file from the Research Open Data release

```
[13]
✓ 5s
df = ROOT.RDataFrame("CollectionTree", "root://eospublic.cern.ch:1094//eos/opendata/atlas/rucio/mc20_13TeV/DAOD_PHYSLITE.37110937._000011.pool.root.1")
```

Name of the TTree

Path to root file (can also provide list of files or strings with wildcards)

- Lot's of Warning messages:
  - `Warning in <TClass::Init>: no dictionary for class <...>`
  - Caused by the fact that we **can not access any of the branches relying on the xAOD class** (needs ATLAS software). Can be **ignored** for now.

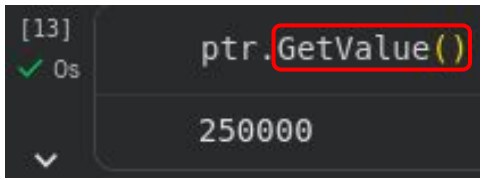
Lazy execution in RDataFrame

Lazy Executions

Most actions in RDataFrame are **lazy**, i.e. they are **not executed on the spot**, but registered with RDataFrame and executed only when a result is accessed for the first time

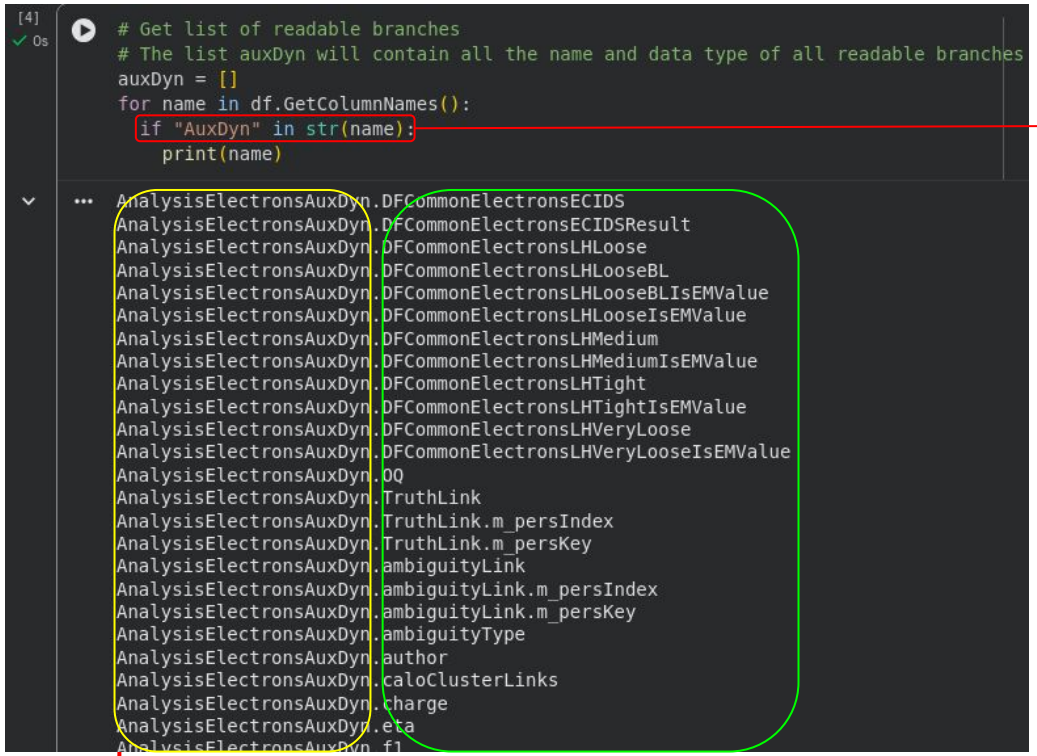
E.g. **getting the number of events** in the data frame does nothing…

… until you explicitly ask for the value



Getting the branches in the RDataFrame

- There are 1519 branches in the Research Open Data
  - Not all of them can be used in a simple RDataFrame analysis



```
[4]
✓ 0s
# Get list of readable branches
# The list auxDyn will contain all the name and data type of all readable branches
auxDyn = []
for name in df.GetColumnNames():
  if "AuxDyn" in str(name):
    print(name)
```

```
AnalysisElectronsAuxDyn.DFCommonElectronsECIDS
AnalysisElectronsAuxDyn.DFCommonElectronsECIDSResult
AnalysisElectronsAuxDyn.DFCommonElectronsLHLoose
AnalysisElectronsAuxDyn.DFCommonElectronsLHLooseBL
AnalysisElectronsAuxDyn.DFCommonElectronsLHLooseBLIsEMValue
AnalysisElectronsAuxDyn.DFCommonElectronsLHLooseIsEMValue
AnalysisElectronsAuxDyn.DFCommonElectronsLHMedium
AnalysisElectronsAuxDyn.DFCommonElectronsLHMediumIsEMValue
AnalysisElectronsAuxDyn.DFCommonElectronsLHTight
AnalysisElectronsAuxDyn.DFCommonElectronsLHTightIsEMValue
AnalysisElectronsAuxDyn.DFCommonElectronsLHVeryLoose
AnalysisElectronsAuxDyn.DFCommonElectronsLHVeryLooseIsEMValue
AnalysisElectronsAuxDyn.OQ
AnalysisElectronsAuxDyn.TruthLink
AnalysisElectronsAuxDyn.TruthLink.m_persIndex
AnalysisElectronsAuxDyn.TruthLink.m_persKey
AnalysisElectronsAuxDyn.ambiguityLink
AnalysisElectronsAuxDyn.ambiguityLink.m_persIndex
AnalysisElectronsAuxDyn.ambiguityLink.m_persKey
AnalysisElectronsAuxDyn.ambiguityType
AnalysisElectronsAuxDyn.author
AnalysisElectronsAuxDyn.caloClusterLinks
AnalysisElectronsAuxDyn.charge
AnalysisElectronsAuxDyn.eta
AnalysisElectronsAuxDyn.f1
```

… and many more

Can only read the branches with AuxDyn

Split into collection and variable

See more on the content in Mariana's introduction to PHYSLITE

Start doing some real work

1. Define signal electrons as electrons passing some basic criteria
   - Creates a mask, **signal_el**, telling if the electron passed criteria or

```
[14]
✓ 0s
    df = df.Define("signal_el","AnalysisElectronsAuxDyn
                                AnalysisElectron
                                Analysis
                                                    ElectronsLHTight")
```

2. Make a new vari ber of signal electrons per event
   - Simp times **signal_el** is true for every event

```
Define('n_signal_el','ROOT::VecOps::Sum(signal_el)')
```

Nothing has been executed yet...

Keep events with at least two signal electrons
   - Using the Filter command

```
[16]
✓ 0s
    df = df.Filter('n_signal_el >= 2',"At least two electrons passing cuts")
```

Data frame containing only the events passing the cut

Cut expression

Description of cut (will be useful later - see next slide)

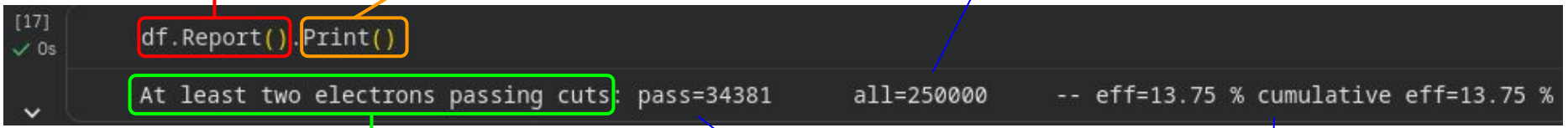
Let's look at some results

- We can generate an overview of the cuts applied so far

Report gives you an overview of the filters applied to your RDataFrame (lazy!)

Executes the command and prints results

Events before cut



The description of the cut we defined in Filter (previous slide)

Events passing cut

Events passing cut [in %] - both cut-by-cut and cumulative (if we had applied more cuts)

A bit more advanced calculations

- For more advanced calculations we can call C++ functions using RDataFrame's Define()-function in python
  - It uses ROOT's Cling C++ interpreter (enable just-in-time compilation of C++ in Jupyter Notebooks)
  - To be able to do this in Google Colab we need to add some paths to the environment variables

```
[18]
✓ 0s
    import os
    os.environ['ROOTSYS'] = '/content/root_build'
    os.environ['PATH'] += ':/content/root_build/bin'
    os.environ['LD_LIBRARY_PATH'] += ':/content/root_build/lib'
```

Cling

Cling is an interactive C++ interpreter. Its advantages over the standard interpreters are that it has command line prompt and uses just-in-time (JIT) compiler for compilation.

A bit more advanced calculations

- We define a simple C++ function calculating the invariant mass of to electrons using the 4-vector class in ROOT

ROOT's C++ interpreter

```
[19]
# Define the function to calculate invariant mass
ROOT.gInterpreter.Declare(
    """
#include <TMath.h>
using VecF_t = const ROOT::RVec<float>&;
float ComputeInvariantMass(VecF_t& pt, VecF_t& eta, VecF_t& phi)
{
  if(int(pt.size())<2)return -999;
  ROOT::Math::PtEtaPhiMVector p1(pt[0], eta[0], phi[0], 0.511);
  ROOT::Math::PtEtaPhiMVector p2(pt[1], eta[1], phi[1], 0.511);
  return (p1 + p2).M()/1000.;
}
"""
)
... True
```

- And we can use it to define a new variable

```
[20]
✓ 0s
df = df.Define('mee','ComputeInvariantMass(AnalysisElectronsAuxDyn.pt[signal_el],\
                                           AnalysisElectronsAuxDyn.eta[signal_el],\
                                           AnalysisElectronsAuxDyn.phi[signal_el])')
```

A tip for using C++ functions

- Put your C++ function into a function and compile from within the notebook
  - Avoids having to define every function inside the notebook
  - Easy to share across different notebooks
  - Somewhat more safe compilation

myFunctions.C

```
#include "TSystem.h"
#include <Math/Util.h>
#include <ROOT/RVec.hxx>
#include "TLorentzVector.h"

using VecF_t = const ROOT::RVec<float>&;

float ComputeInvariantMass(VecF_t& pt, VecF_t& eta, VecF_t& phi)
{
  if(int(pt.size())<2)return -999;
  ROOT::Math::PtEtaPhiMVector p1(pt[0], eta[0], phi[0], 0.511);
  ROOT::Math::PtEtaPhiMVector p2(pt[1], eta[1], phi[1], 0.511);
  return (p1 + p2).M()/1000.;
};

float ComputeInvariantMass4E(VecF_t& pt, VecF_t& eta, VecF_t& phi
{
  if(int(pt.size())<4)return -999;
  ROOT::Math::PtEtaPhiMVector p1(pt[0], eta[0], phi[0], 0.511);
  ROOT::Math::PtEtaPhiMVector p2(pt[1], eta[1], phi[1], 0.511);
  ROOT::Math::PtEtaPhiMVector p3(pt[2], eta[2], phi[2], 0.511);
  ROOT::Math::PtEtaPhiMVector p4(pt[3], eta[3], phi[3], 0.511);
  return (p1 + p2 + p3 + p4).M()/1000.;
};

```

Compile the C++ library from within your notebook

```
[ ]  ! g++ -shared -fPIC -o ./myFunctions.so ./myFunctions.C `root-config --cflags --glibs`
```

Create a histogram from the data frame

- RDataFrame has built-in functions for creating histograms
  - Histo1D(), Histo2D(), Histo3D()

```
[34]
✓ 0s
histogram = df.Histo1D( ("h_mee","Invariant mass of two electrons;m_{ee} [GeV];Events",500,0,1000) ,"mee")
```

Variable to plot

Title; x-axis label; y-axis label

Number of bins, start, stop

Name of histogram

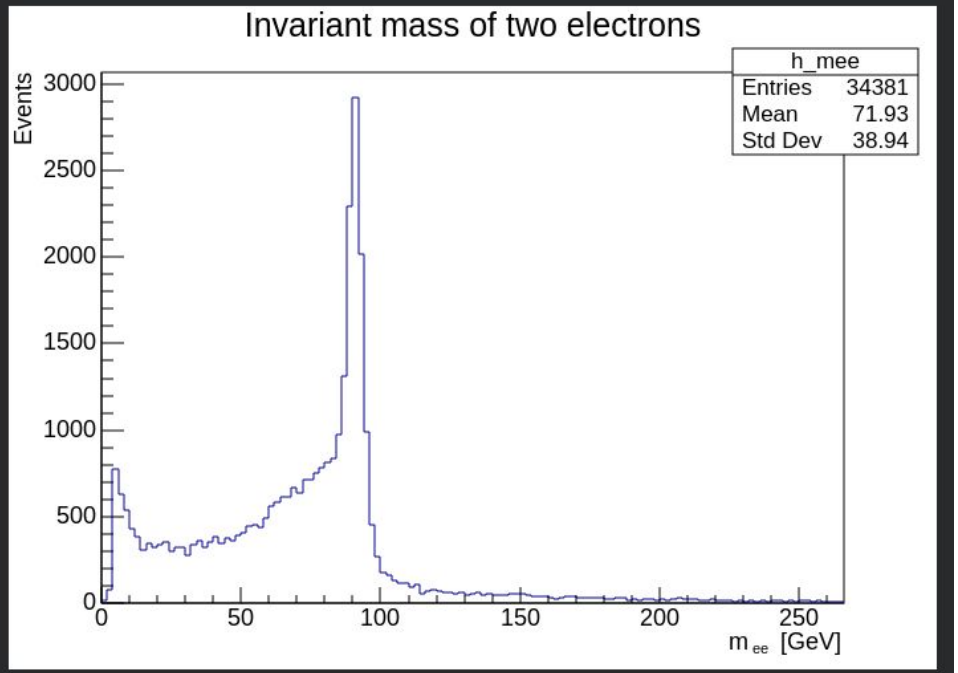
- Again, this is lazy until we…

Create a histogram from the data frame

- … we e.g. plot the histogram

~6 minutes!

```
[25]
✓ 6m
%jsroot on
c = ROOT.TCanvas("c","c",1)
c.Draw()
histogram.Draw()
```



Let’s build a more complete analysis

- We need to
  - Retrieve scale factors to scale each MC sample to the integrated luminosity of the data set (i.e. need cross-sections, filter efficiencies, k-factors, sum of event weights)
    - All of this can be retrieved from *atlasopenmagic*
  - Categorize the PHYSLITE files into the different background processes (ttbar, diboson, Z+jets, etc…) for nice plots
  - Parallelize the execution to make analysis more efficient
  - Define a set of C++ functions and compile and call on them from the Jupyter Notebook

Let's start with defining the samples

- For this we use *atlasopenmagic* - as we have done several times during this tutorial

```
atom.available_releases()
# Set the release to the Research Open Data release
atom.set_release('2024r-pp')
```

```
Fetching metadata for release: 2024r-pp...
Available releases:
======================================
2016e-8tev           2016 Open Data for education release of 8 TeV proton-proton collisions (https://opendata.cern.ch/record/3860).
2020e-13tev          2020 Open Data for education release of 13 TeV proton-proton collisions (https://cern.ch/2r7xt).
2024r-pp             2024 Open Data for research release for proton-proton collisions (https://opendata.cern.record/80020).
2024r-hi             2024 Open Data for research release for heavy-ion collisions (https://opendata.cern.ch/record/80035).
2025e-13tev-beta     2025 Open Data for education and outreach beta release for 13 TeV proton-proton collisions (https://opendata.cern.ch/record/93910).
2025r-evgen-13tev    2025 Open Data for research release for event generation at 13 TeV (https://opendata.cern.ch/record/160000).
2025r-evgen-13p6tev  2025 Open Data for research release for event generation at 13.6 TeV (https://opendata.cern.ch/record/160000).
Fetching datasets: 100%|██████████| 374/374 [00:02<00:00, 125.60datasets/s]
✓ Successfully cached 374 datasets.
Active release: 2024r-pp. (Datasets path: REMOTE)
```

```
# Define the input data sets we'd like to use (a selection of samples used for the H->ZZ analysis)
defs = {
    'Zjets' :{'dids': [700320,700321,700322], 'color': "#6b59d3" },
    'ZZ':     {'dids': [700600],'color': "#ff0000" },# red
    'Higgs':  {'dids': [345060, 346228, 346310],'color': "#00cdff" },# light blue
    'Data':{'dids':['data']}
}

# Unfortunately Google Colab is too slow
# so in interest of time let's only look at Higgs
if isColab:
    del defs['Zjets']
    del defs['ZZ']
    del defs['Data']

# In swan we can access the samples directly from /eos/ while in colab we must use the root (or https) protocols
samples   = atom.build_dataset(defs, protocol='root' if isColab else 'eos', cache=True)
```

If you're running on Google Colab we need to drastically reduce the data set

Load the samples into an RDataFrame

- When we later are gonna parallelize our execution it is beneficial to put all the files we want to process into one big RDataFrame
  - Remember that we will not have any memory issues doing this since ROOT is handling memory in a completely different way than e.g. is done in Pandas dataframe
  - We still need to know which process (real data, simulated ttbar, simulated diboson etc.) each file belongs to when we later want to plot histograms etc.
  - We achieve this by loading the RDataFrame using a specification file in json format

```
{
   "samples": {
      "sampleA": {
         "trees": ["tree1", "tree2"],
         "files": ["file1.root", "file2.root"],
         "metadata": {
            "lumi": 10000.0,
            "xsec": 1.0,
            "sample_category" = "data"
            }
      },
      "sampleB": {
         "trees": ["tree3", "tree4"],
         "files": ["file3.root", "file4.root"],
         "metadata": {
            "lumi": 0.5,
            "xsec": 1.5,
            "sample_category" = "MC_background"
            }
      }
   }
}
```

From RDataFrame documentation

Create the specification file from atlasopenmagic

- Looping over the dictionary used in *atlasopenmagic*
  - Creates a dictionary of files belonging to each of the DSIDs/data with the corresponding metadata

```
# Name of the json file.
outfilename = "./spec.json"
# Outermost key must be "samples"
RDF_spec = {"samples":{}}
# loping over the samples defined above
for name, info in defs.items():
    for did in info["dids"]:
        if not "Data" in name:
            # Get the metadata from atom
            metadata = {'xsec':atom.get_metadata(did,'cross_section_pb'),
                    'sumOfWeights':atom.get_metadata(did,'sumOfWeights'),
                    'genFiltEff':atom.get_metadata(did,'genFiltEff'),
                    'kFactor':atom.get_metadata(did,'kFactor'),
                    'proc':name}
            # Populates the dictionary with input samples and metadata
            RDF_spec['samples'][did] = {"trees":["CollectionTree"],
                                        "files":[item.replace("simplecache::", "") for item in atom.get_urls(str(did), protocol='root', cache=True)],
                                        "metadata":metadata}

        else:
            # In interest of time we only look at one run in data (i.e. onlt files with id 37020379 in name)
            inputlist = [item.replace("simplecache::", "") for item in atom.get_urls(str(did), protocol='root', cache=True) if '37020379' in item]
            # metadata is simple for data
            metadata = {'xsec':1.0,
                    'sumOfWeights':1.0,
                    'genFiltEff':1.0,
                    'kFactor':1.0,
                    'proc':name}
            # Populates the dictionary with input samples and metadata
            RDF_spec['samples'][did] = {"trees":["CollectionTree"], "files":inputlist, "metadata":metadata}
# Save the dictionary to json file
with open(outfilename, "w") as f:
    json.dump(RDF_spec,f)
print("Created json file %s\nCan now be loaded into a RDataFrame using:\ndf = ROOT.RDF.Experimental.FromSpec(\"spec.json\")"%outfilename)
```

Creating the metadata dictionary to add to the spec file from *atlasopenmagic*

All the files to be used for a given DSID

For data we restrict the files to one run only (i.e. run 311481 from 2016)

Look at the spec file

The DSID (i.e. corresponding to a specific process)

Name of the TTree (same in all files)

All the files belonging to this DSID

The metadata for this DSID and the category it belongs to ("Signal (m_H = 125 GeV")

A full analysis

- Multi threading is so simple (unf. colab only gives you 2 CPUs):

```
[52]
✓ 0s
ROOT.EnableImplicitMT()
```

- Then we're ready to define the RDataFrame and do some real stuff

```
[69]
✓ 1s
df = ROOT.RDF.Experimental.FromSpec("spec.json")
ROOT.RDF.Experimental.AddProgressBar(df)
```

- Stuff which only needs to be defined once we change DSID (second level in our spec dictionary)
  - Use DefinePerSample() and get info from metadata entry in json file

```
[19]
✓ 0s
# Get category
df = df.DefinePerSample("isData",f'rdfsampleinfo_.Contains("mc20_13TeV") ? 0 : 1')
```

Retrieves filename and checks if data/MC

```
# Get category
df = df.DefinePerSample("category",'rdfsampleinfo_.GetS("proc")')
# Get cross-section
df = df.DefinePerSample("xsec",'rdfsampleinfo_.GetD("xsec")')
# Get sum of event weights
df = df.DefinePerSample("sow",'rdfsampleinfo_.GetD("sumOfWeights")')
# Get generator filter efficiency
df = df.DefinePerSample("eff",'rdfsampleinfo_.GetD("genFiltEff")')
# Get k-factor
df = df.DefinePerSample("kfac",'rdfsampleinfo_.GetD("kFactor")')
# Set the luminoisty of the data set (run 311481) you've added (36.1e3 if you have loaded everything)
df = df.DefinePerSample("lumi",'343.1')
```

Retrieve information from the metadata in the spec file

S for string
D for double/float

A full analysis

- Start adding operations on every event - starting with the total weight

```
# Calculate the weight to be applied to every event (for data it is simply 1)
wgtstr = f'isData ? 1.0 : ((lumi*xsec*kfac*eff)/sow))*EventInfoAuxDyn.mcEventWeights.at(0)'
df = df.Define("wgt",wgtstr)
```

No weights on data

The MC generator weights (event-by-event)

Reminder:

$$w_\sigma = \frac{\int L dt \, \sigma \, \eta \, k \, w}{\sum_i w_i}$$

- Adding selections, cuts and variables (similar to previous example)

```
# Define signal electrons
df = df.Define("signal_el","AnalysisElectronsAuxDyn.eta > -2.47 && \
                            AnalysisElectronsAuxDyn.eta <  2.47 && \
                            AnalysisElectronsAuxDyn.pt > 7000 && \
                            AnalysisElectronsAuxDyn.DFCommonElectronsLHLoose")
# Make a new variable with the number of signal electrons per event
df = df.Define('n_signal_el','ROOT::VecOps::Sum(signal_el)')
# Only select events with at leas four signal electrons
df = df.Filter('n_signal_el >= 4',"At least four electrons passing cuts")
# Calculate the invariant mass of those four electrons
df = df.Define('m4e','ComputeInvariantMass4E(AnalysisElectronsAuxDyn.pt[signal_el],\
                                             AnalysisElectronsAuxDyn.eta[signal_el],\
                                             AnalysisElectronsAuxDyn.phi[signal_el])')
```

A full analysis

- Create histograms of the 4L-invariant mass, one for each category

```
all_histograms = []
histograms = {}
for proc in defs.keys():
  histograms[proc] = df.Filter(f'category == "{proc}"').Histo1D(("h_m4e_%s"%proc,"Invariant mass of 4 electrons;m_{4e} [GeV];Events",50,0,1000),"m4e","wgt")
  all_histograms.append(histograms[proc])
```

Histo1D is lazy, so this loop will not do anything now.
Puts all histograms in a list for parallel execution

Filter out the events belonging to each category

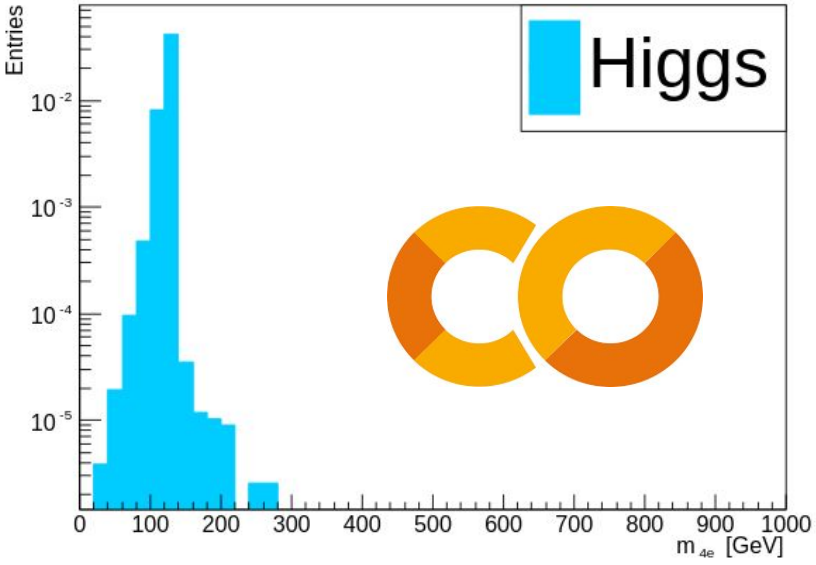
Book histograms, with a unique name for that process

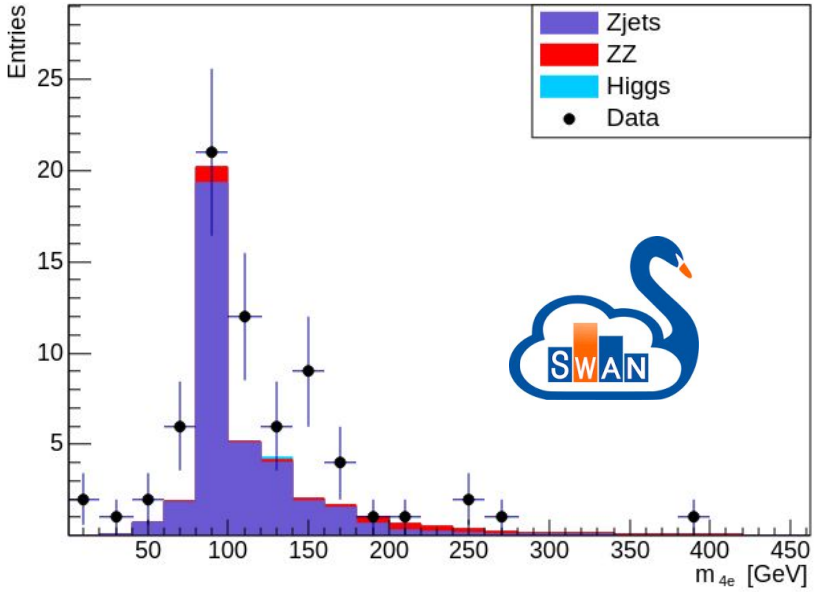
When we have defined all our histograms we send the execution graph to ROOT for processing

```
[ ]  ROOT.RDF.RunGraphs(all_histograms)
```

And after some time you get this…

```
%jsroot on
c = ROOT.TCanvas("c","c",1)
c.Draw()
c.SetLogy()
hstack.Draw("hist")
hstack.GetXaxis().SetTitle("m_{4e} [GeV]")
hstack.GetYaxis().SetTitle("Entries")
if "Data" in histograms.keys():
    histograms["Data"].Draw("ep same")
legend.Draw()
```





Convert from ROOT to NumPy

- RDataFrame also comes with functions to convert from RDF to numpy arrays/pandas

```
df = df.Define("lep1_pt","AnalysisElectronsAuxDyn.pt[signal_el].at(0)")
df = df.Define("lep2_pt","AnalysisElectronsAuxDyn.pt[signal_el].at(1)")
df = df.Define("lep3_pt","AnalysisElectronsAuxDyn.pt[signal_el].at(2)")
df = df.Define("lep4_pt","AnalysisElectronsAuxDyn.pt[signal_el].at(3)")
df = df.Define("label",f'category == "Higgs" ? 1 : 0')
```

```
numpy_col = ["lep1_pt","lep2_pt","lep3_pt","lep4_pt","m4e","label","isData","wgt","category"]
```

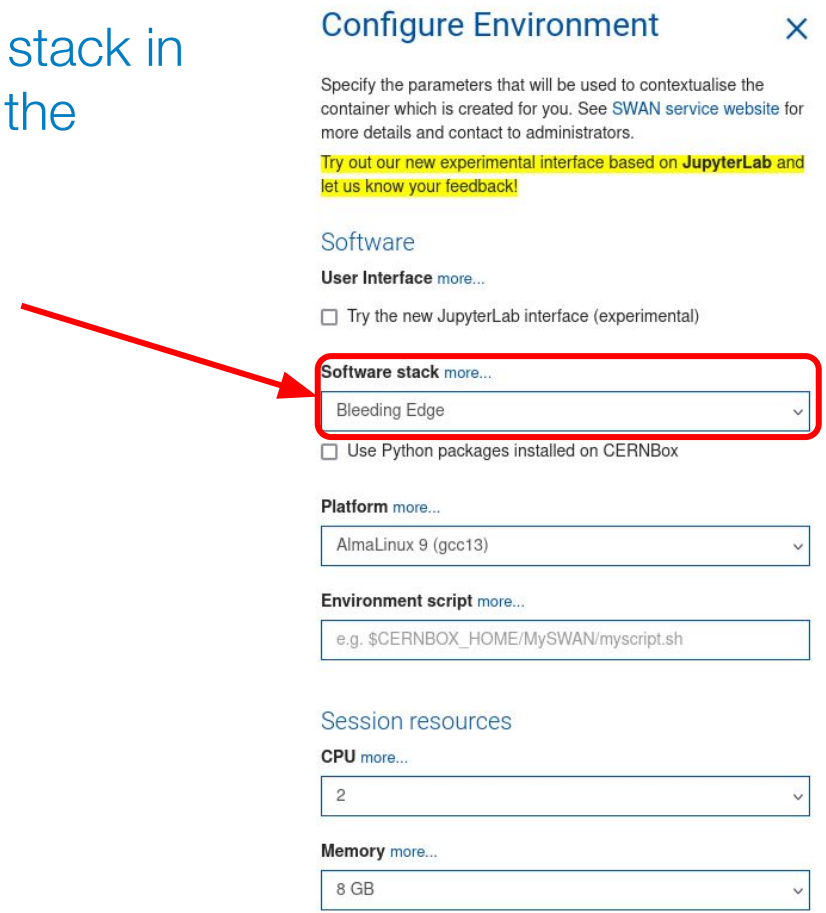
```
import pandas as pd
pandas_df = pd.DataFrame(data=df.AsNumpy(numpy_col))
```

Why bother going through numpy arrays?

- You can get pytorch tensor directly from the RDataFrame
  - This is work in development, thus you need absolute latest version of ROOT (>6.38)
- **Disclaimer**: this is really work in progress, so under constant development!
- See talk at ROOT's user workshop last week

Setup in Swan

- Setup “Bleeding Edge” Sofware stack in Swan, and go to the last part of the notebook!



Converting from RDF to pyTorch tensor

```
# Getting training and validaion sets (as pytorch tensors)
gen_train, gen_validation =  ROOT.TMVA.Experimental.CreatePyTorchGenerators(df,
                                                                            batch_size,
                                                                            chunk_size,
                                                                            block_size,
                                                                            shuffle = True,
                                                                            columns = mycolumns,
                                                                            target = target,
                                                                            drop_remainder = True,
                                                                            set_seed = 42,
                                                                            validation_split = 0.30)
```

Chunking, blocking and batching