Reaction Aware Dataframes

Derek Glazier
University of Glasgow

Electron Ion Collider UK Gathering
University of Birmingham
18-19th November 2024

Preamble

EXPERIMENTAL::

The point, for me, was to see if

RDataFrame could be used in complex analyses
For example, in analysing multi-final state event generators

This could be made simple and “user friendly”

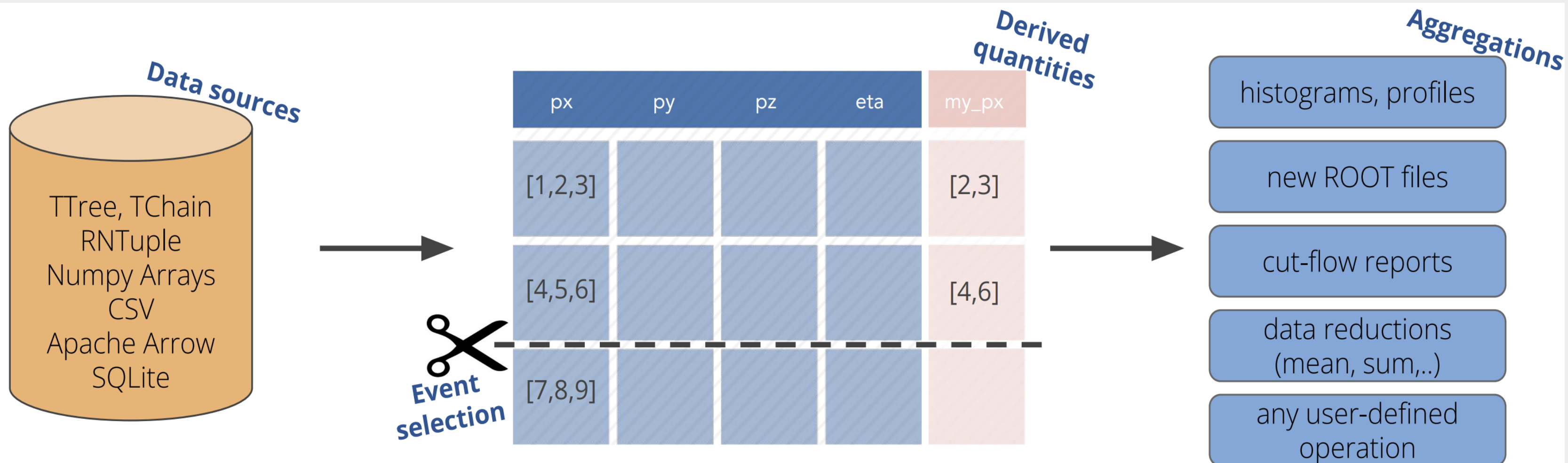
This could be extended to ePIC data

The point today is to encourage others to try out RDataFrame for ePIC analysis

Summary

- What are dataframes
- Why use a dataframe instead of a for loop in a large Root script ? Or a more Object Orientated approach
- How can I use a dataframe
  - Simple things
  - More complicated things : functions and functors
- What does “reaction aware” mean ?
- RAD with Pythia
- RAD with benchmarking

What is ROOT RDataFrame



Some aspects particular to HEP

Input datasets are much larger than memory, entries are statistically independent.

Histograms, new ROOT files as common aggregations.

Collections are ubiquitous.

The goal

Ease of use, good performance and scaling from 1 to 1000+ cores out of the box. Extensibility.

Ergonomic support for common HEP use cases (systematics, working with collections, ...).

https://indico.fnal.gov/event/23628/contributions/241029/

What is RDataFrame

Construct → Transform → Results

Use strings!

```
ROOT::RDataFrame df("mytree", {"f1.root", "f2.root"});
auto h = df.Filter("x > 0").Histo1D("x");
h->Draw(); // the event loop is run here, upon first access to one of the results
```

Use modern C++!

```
// C++11 lambda expressions and C++ functions are also supported as filter expressions
auto filtered_df = df.Filter([](float x) { return x > 0; }, {"x"});
auto hx = filtered_df.Histo1D("x");
auto hy = filtered_df.Histo1D("y");
hx->Draw(); // event loop is run here, both hx and hy are filled
```

RDataFrame Actions

```
ROOT::RDat[aFrame df(d]ataset);
auto df2 = df.Filter("x > 0")
             .Define("r2", "x*x + y*y");
auto rHist = df2.Histo1D("r2");
df2.Snapshot("newtree", "out.root");
```

- on this (ROOT, CSV, ...) dataset
- [...]events for which x > 0
- define r2 = $x^2 + y^2$
- plot r2 for events that pass the cut
- write the skimmed data and r2 to a new ROOT file

event selection

derived quantities, object selections

data aggregations

Users can inject **arbitrary code** at all steps, which makes this relatively simple API extremely versatile.

Plus lots of other stuff….

For my analysis I want to take reconstructed data and create a TTree of high level physics info for my reaction

EIC Tutorial with Python (I)

```
# Open input file and define branches we want to look at with uproot
events_tree = up.open(infile)["events"]
# Get particle information# Get particle information
partGenStat = events_tree["MCParticles.generatorStatus"].array()
partMomX = events_tree["MCP articles.momentum.x"].array()
partMomY = events_tree["MCParticles.momentum.y"].array()
partMomZ = events_tree["MCParticles.momentum.z"].array()
partPdg = events_tree["MCParticles.PDG"].array()

# Get reconstructed track information
trackMomX = events_tree["ReconstructedChargedParticles.momentum.x"].array()
trackMomY = events_tree["ReconstructedChargedParticles.momentum.y"].array()
trackMomZ = events_tree["ReconstructedChargedParticles.momentum.z"].array()
# Get assocations between MCParticles and ReconstructedChargedParticles
recoAssoc = events_tree["ReconstructedChargedParticleAssociations.recID"].array()
simuAssoc = events_tree["ReconstructedChargedParticleAssociations.simID"].array()
```

```
# Define histograms below
partEta = ROOT.TH1D("partEta","Eta of Thrown Charged Particles;Eta",100, -5 ,5 )
matchedPartEta = ROOT.TH1D("matchedPartEta","Eta of Thrown Charged Particles That Have Matching Track", 100, -5 ,5);
matchedPartTrackDeltaR = ROOT.TH1D("matchedPartTrackDeltaR","Delta R Between Matching Thrown and Reconstructed Charg
e Particle", 5000, 0, 5);
```

EIC Tutorial with Python (II)

```
# Add main analysis loop(s) below
for i in range(0, len(events_tree)): # Loop over all events
    for j in range(0, len(partGenStat[i])): # Loop over all thrown particles
        if partGenStat[i][j] == 1: # Select stable particles
            pdg = abs(partPdg[i][j]) # Get PDG for each stable particle
            if(pdg == 11 or pdg == 13 or pdg == 211 or pdg == 321 or pdg == 2212):
                trueMom = ROOT.TVector3(partMomX[i][j], partMomY[i][j], partMomZ[i][j])
                trueEta = trueMom.PseudoRapidity()
                truePhi = trueMom.Phi()

                partEta.Fill(trueEta)
                for k in range(0,len(simuAssoc[i])): # Loop over associations to find matching ReconstructedChargedP
article
                    if (simuAssoc[i][k] == j):
                        recMom = ROOT.TVector3(trackMomX[i][recoAssoc[i][k]], trackMomY[i][recoAssoc[i][k]], trackMo
mZ[i][recoAssoc[i][k]])
                        deltaEta = trueEta - recMom.PseudoRapidity()
                        deltaPhi = TVector2. Phi_mpi_pi(truePhi - recMom.Phi())
                        deltaR = math.sqrt((deltaEta*deltaEta) + (deltaPhi*deltaPhi))

                        matchedPartEta.Fill(trueEta)
                        matchedPartTrackDeltaR.Fill(deltaR)
```

EIC Tutorial with RDataFrame

```
  ROOT::RDataFrame df("events", infile);

  // Define new dataframe node with additional columns
  auto df1 =  df.Define("statusFilter",   "MCParticles.generatorStatus == 1"     )
                .Define("absPDG",         "abs(MCParticles.PDG)"                 )
                .Define("pdgFilter",      "absPDG == 11 || absPDG == 13 || absPDG == 211 || absPDG == 321 || absPDG =
= 2212")
                .Define("particleFilter","statusFilter && pdgFilter"             )
                .Define("filtMCParts",    "MCParticles[particleFilter]"          )
                .Define("assoFilter",     "Take(particleFilter,ReconstructedChargedParticleAssociations.simID)") // I
ncase any of the associated particles happen to not be charged
                .Define("assoMCParts",    "Take(MCParticles,ReconstructedChargedParticleAssociations.simID)[assoFilte
r]")
                .Define("assoRecParts",   "Take(ReconstructedChargedParticles,ReconstructedChargedParticleAssociation
s.recID)[assoFilter]")
                .Define("filtMCEta",      getEta<MCP>   , {"filtMCParts"} )
                .Define("filtMCPhi",      getPhi<MCP>   , {"filtMCParts"} )
                .Define("accoMCEta",      getEta<MCP>   , {"assoMCParts"} )
                .Define("accoMCPhi",      getPhi<MCP>   , {"assoMCParts"} )
                .Define("assoRecEta",     getEta<RecoP> , {"assoRecParts"})
                .Define("assoRecPhi",     getPhi<RecoP> , {"assoRecParts"})
                .Define("deltaR",         "ROOT::VecOps::DeltaR(assoRecEta, accoMCEta, assoRecPhi, accoMCPhi)");

  // Define histograms
  auto partEta                = df1.Histo1D({"partEta","Eta of Thrown Charged Particles;Eta",100,-5.,5.},"filtMCEt
a");
  auto matchedPartEta         = df1.Histo1D({"matchedPartEta","Eta of Thrown Charged Particles That Have Matching Tr
ack",100,-5.,5.},"accoMCEta");
```

ROOT::RVec<>

Basically wrapper for std::vector<>

ROOT gives some additional utiltities

- implicit looping (like numpy arrays)

RAD gives some more

https://github.com/dglazier/rad/blob/master/include/RVecHelpers.h

ePIC data in RDataFrame will be contained in RVecs

These will be the arguments given to functions

```
//calculate magnitude for all entries in vecs
RVec<double> ThreeVectorMag(const RVec<double> &x,
                            const RVec<double> &y, const RVec<double> &z){
    return sqrt(x * x + y * y + z * z);
}
```

Note best to pass in const references (&)

Keeps data safe and prevents unecessary copy

Don't use TLorentzVector

https://root.cern.ch/doc/master/classTLorentzVector.html

Attention

TLorentzVector is a legacy class. It is slower and worse for serialization than the recommended superior alternative ROOT::Math::LorentzVector. ROOT provides specialisations of the ROOT::Math::LorentzVector template which offer superior runtime performance, i.e.:

- ROOT::Math::PtEtaPhiMVector based on pt (rho),eta,phi and M (t) coordinates in double precision
- ROOT::Math::PtEtaPhiEVector based on pt (rho),eta,phi and E (t) coordinates in double precision
- ROOT::Math::PxPyPzMVector based on px,py,pz and M (mass) coordinates in double precision
- ROOT::Math::PxPyPzEVector based on px,py,pz and E (energy) coordinates in double precision
- ROOT::Math::XYZTVector based on x,y,z,t coordinates (cartesian) in double precision (same as PxPyPzEVector)
- ROOT::Math::XYZTVectorF based on x,y,z,t coordinates (cartesian) in float precision (same as PxPyPzEVector but float)

More details can be found in the documentation of the Physics Vectors package.

Main user difference using external functions not class methods

```
auto cmBoost = cm.BoostToCM();
PxPyPzMVector  CMTar=boost(tar,cmBoost);
```

Note on usage

```
//standard rdataframe usage :
auto df1 = df.Define("Mass0","MassCalc(px[0],py[0],pz[0],e[0])")
             .Define("Mass1","MassCalc(px[1],py[1],pz[1],e[1])");

//equivalent to :
auto df0 = df.Define("Mass0","MassCalc(px[0],py[0],pz[0],e[0])");
auto df1 = df0.Define("Mass1","MassCalc(px[1],py[1],pz[1],e[1])");

//and in rad :
rad.Define("Mass0","MassCalc(px[0],py[0],pz[0],e[0])");
rad.Define("Mass1","MassCalc(px[1],py[1],pz[1],e[1])");

//or if rdf interface defined :
rad::rdf::Mass(rad,"Mass0","{0}");
rad::rdf::Mass(rad,"Mass1","{1}");
```

Returns Rnode (df1) with define applied

*** I prefer this**

Rad keeps the last defined RNode as a datamember

Can be accessed with rad.CurrFrame()

```
auto df0 = epic.CurrFrame();
auto hW = df0.Histo1D({"W","W",100,0,20.},"tru_W");
```

PODIO Data Format

Plain-Old-Data I/O, aka PODIO
avoid deep-object hierarchies

To both improve runtime performance and simplify the implementation

Support for inter-object relations

For Example HepMC3 root file shows :

```
particles.pid : Int_t pid[particles_]
particles.status : Int_t status[particles_]
particles.mass : Double_t mass[particles_]
particles.momentum.m_v1 : Double_t m_v1[particles_]
particles.momentum.m_v2 : Double_t m_v1[particles_]
...
```

ePIC Data Format

O(4000) branches !

```
ReconstructedParticles.energy : Float_t energy[ReconstructedParticles_]
ReconstructedParticles.momentum.x : Float_t x[ReconstructedParticles_]
ReconstructedParticles.charge : Float_t charge[ReconstructedParticles_]
ReconstructedParticles.mass : Float_t mass[ReconstructedParticles_]

ReconstructedParticleAssociations.recID : UInt_t recID[ReconstructedParticleAs_]
ReconstructedParticleAssociations.simID : UInt_t simID[ReconstructedParticleAs_]

MCParticles.momentum.x : Float_t x[MCParticles_]
MCParticles.charge : Float_t charge[MCParticles_]
MCParticles.mass : Double_t mass[MCParticles_]
```

And many more ...

Injecting algorithms

Functions and functors

If your calculation depends on event data use a function
These can be defined in some header or in a C++ lambda
Event data is passed in as arguments

If your calculation also depends on some parameters use a functor

A functor is an instance of a class with a function and parameters kept as data members

ePIC truth matching

Optional analysis strategy :

Use ReconstructedParticles and MCParticles branches
Map these to “rec” and “tru” types of variables
- Filter tru indices for gen_stat == 1 (removes a lot!)
- Loop over tru, look for rec and link to tru index
- Filter/reorder all other columns to sync with new tru

Can now operate on real matched reaction when reconstructed particles exist
Calculations automatically performed for both rec and tru
Resolutions can be determined for all variables

Installation

Header only
Compilation at run-time via standard ROOT scripting(cling)

Download from github (CURRENTLY NOT STABLE)

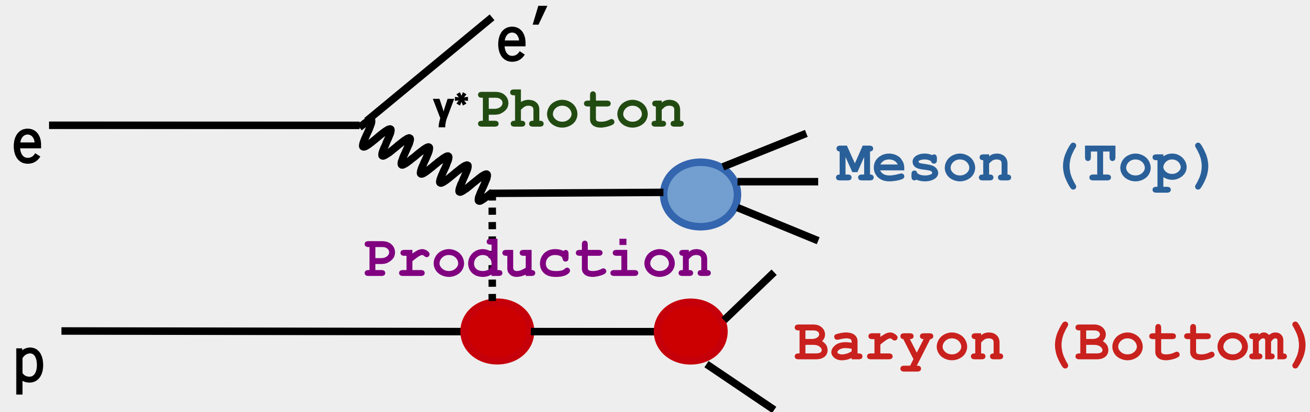
```
git clone https://github.com/dglazier/rad
```

Add the include directory to you ROOT_INCLUDE_PATH

```
setenv ROOT_INCLUDE_PATH /to/where/is/rad/include
```

Structure : Some classes to configure RAD dataframes
Some functions/functors to calculate stuff

Reaction Aware



To analyse a reaction in general we need to :

- Identify final state particles (Filter)
- Associate them with Top and Bottom vertices (Define)
- Calculate Photon Kinematics
- Calculate Production Kinematics
- Calculate Top/Bottom Intermediate states
- Calculate Top/Bottom Decay Kinematics

How is it Reaction Aware

The method is to identify the indices of each particle in the data vector

These are then stored in a reaction Map
This can change event-to-event
Given an index helper functions create 4-vectors etc.
Can then be used in standarised kinematic calculations

```
//Analysing HepMC or MCMatched data
epic.setParticleIndex("idxPi",2);

//Analysing reconstrcuted, with no matching
epic.setParticleIndex("idxPi",
                      rad::indice::useNthOccurance(1,211),
                      {"ReconstructedParticles.PDG"});
```

I know my pion is 2nd
In particle list

Can use any
complicated algorithm

From indices to 4-vectors

User code :

```
epic.setParticleIndex("idxEl",0,11);
epic.setParticleIndex("idxPo",1,-11);
epic.setParticleIndex("idxPi",2,211);
epic.setParticleIndex("idxN",3,2112);
```

Index known as MCmatched

```
epic.setMesonParticles({"idxEl","idxPo","idxPi"});
```

Inside RAD function definition :

```
//Get meson (Top) 4-vector
auto meso=FourVector(react[names::MesonsIdx()],px,py,pz,m);
```

Returns 4-vector sum of e−, e+, π+

react → the reaction map, stores particle indices
MesonsIdx is the position in react where meson indices are stored

Inside RAD Kinematic Function

```
///\brief return 4 momentum transfer squared of "in particles" - "out particles" on top vertex

  //Get photon 4-vector
  auto phot=PhotoFourVector(react,px,py,pz,m);
  //Get meson (Top) 4-vector
  auto meso=FourVector(react[names::MesonsIdx()],px,py,pz,m);
  //subtract
  auto psum = phot-meso;
  //return t
  return - (psum.M2());
}
```

Uses react for beam and scattered e- indices

Vectors of momentum components

react → the reaction map, stores particle indices

MesonsIdx is the position in react where meson indices are stored

Inside RAD Kinematic Function

```
///\brief return 4 momentum transfer squared of "in particles" - "out particles" on top vertex
template<typename Tp, typename Tm>
Tp TTop(const config::RVecIndexMap& react,
        const RVec<Tp> &px, const RVec<Tp> &py, const RVec<Tp> &pz, const RVec<Tm> &m){
  //Get photon 4-vector
  auto phot=PhotoFourVector(react,px,py,pz,m);
  //Get meson (Top) 4-vector
  auto meso=FourVector(react[names::MesonsIdx()],px,py,pz,m);
  //subtract
  auto psum = phot-meso;
  //return t
  return - (psum.M2());
}
```

Vectors of momentum components

React → the reaction map, stores particle indices

MesonsIdx is the position in react where meson indices are stored

Particle Creator

As well as reconstructed particles, we also need to manipulate intermediate or missing particle 4-vectors

ParticleCreator adds these momentum components to the reconstructed components vectors and generates index for the new particle

Sum – sum 4-vector of given particles
Diff – subtract 4-vector of given indices
Beam – define a fixed 4-vector for beam particle
Miss – subtract given particles from sum of beams

User code configure stage

```
rad::config::ePICReaction epic{"events","jpac_z3900_10x100.root"};
epic.SetBeamsFromMC();

//epic.AliasColumnsAndMC();
epic.AliasColumnsAndMatchWithMC();

//Assign particles names and indices
epic.setScatElectronIndex(4);          e-'
epic.setParticleIndex("idxEl",0,11);   e-
epic.setParticleIndex("idxPo",1,-11);  e+    Final state particles
epic.setParticleIndex("idxPi",2,211);  π
epic.setParticleIndex("idxN",3,2112);  n

//Create some intermediate particles                decaying states
epic.Particles().Sum("idxJ",{"idxEl","idxPo"});    J/ψ → e- + e+
epic.Particles().Sum("idxZ",{"idxPi","idxJ"});     Z → J/ψ + π
                    New name{Particles to sum}
//Group particles into top and bottom vertices
epic.setMesonParticles({"idxEl","idxPo","idxPi"});
//can also add missing particles Missing particle : n = beams- e'-Z
epic.Particles().Miss("idxCalcN",{rad::names::ScatEle().data(),"idxZ"});
epic.setBaryonParticles({"idxCalcN"});

//must call this after all particles are configured
epic.makeParticleMap();
```

User code kinematics stage

Call predefined rad::rdf functions for kinematics

```
//masses column name, {+ve particles}, {-ve particles}
rad::rdf::MissMass(epic,"W","{scat_ele}");
rad::rdf::MissMass(epic,"MissMass","{scat_ele,idxN,idxZ}");
rad::rdf::Mass(epic,"Whad","{idxPi,idxEl,idxPo,idxN}");
rad::rdf::Mass(epic,"JMass","{idxJ}");
rad::rdf::Mass(epic,"ZMass","{idxZ}");
rad::rdf::Mass(epic,"MissNMass","{idxCalcN}");

//t distribution, column name
rad::rdf::TTop(epic,"t_gZ");
rad::rdf::TBot(epic,"t_pn");
rad::rdf::TPrimeBot(epic,"tp_pn");
rad::rdf::TPrimeTop(epic,"tp_gZ");

//CM production angles
rad::rdf::CMAngles(epic,"CM");
```

```
///////////////////////////////////////////////////////////
//Define histograms
///////////////////////////////////////////////////////////
auto df0 = epic.CurrFrame();

auto hW = df0.Histo1D({"W","W",100,0,20.},"tru_W");
auto hWhad = df0.Histo1D({"Whad","Whad",100,0,20.},"tru_Whad");
auto hMesonMass = df0.Histo1D({"MesonMass","M(e-,e+, #pi) [GeV]",100,.3,5.},"tru_ZMass");
auto hMissMass = df0.Histo1D({"MissMass","Mmiss [GeV]",1000,-10,10},"tru_MissMass");
auto hJMass = df0.Histo1D({"JMass","M(e-,e+) [GeV]",100,.3,5.},"tru_JMass");
auto htpn = df0.Histo1D({"tpn","t(p,n) [GeV^{2}]",100,-2,5},"tru_t_pn");
auto htgZ = df0.Histo1D({"tgZ","t(g,Z) [GeV^{2}]",100,-2,5},"tru_t_gZ");
auto htprimepn = df0.Histo1D({"tprimepn","t'(p,n) [GeV^{2}]",100,-2,5},"tru_tp_pn");
auto htprimegZ = df0.Histo1D({"tprimegZ","t'(p,n) [GeV^{2}]",100,-2,5},"tru_tp_gZ");
auto hthCM=df0.Histo1D({"cthCM","cos(#theta_{CM})",100,-1,1},"tru_CM_CosTheta");
auto hphCM=df0.Histo1D({"phCM","#phi_{CM})",100,-TMath::Pi(),TMath::Pi()},"tru_CM_Phi");
```

More complicated functor

```
class UndoAfterBurn
 {
 public:
   UndoAfterBurn(PxPyPzMVector p_beam,PxPyPzMVector e_beam,Float_t angle=-0.025):_crossAngle{angle}{
     //calculate and store current boosts and rotations
     RotsAndBoosts(p_beam,e_beam);
   }
   template<typename Tp, typename Tm>
   void operator()(RVec<Tp> &px,RVec<Tp> &py,RVec<Tp> &pz, const RVec<Tm> &m) const
   {
     //apply to all particles
     auto n_parts = m.size();
     for(uint i=0;i<n_parts;++i){
       undoAfterburn(i,px,py,pz,m);
     }
   }
 private:
   //Determine transformations
   void RotsAndBoosts(PxPyPzMVector p_beam,PxPyPzMVector e_beam);
   //change momentum for a particle as per prescription
   template<typename Tp, typename Tm>
   void undoAfterburn(uint idx,RVec<Tp> &px,RVec<Tp> &py,RVec<Tp> &pz, const RVec<Tm> &m) const;

   // Objects for undoing afterburn boost
   Float_t _crossAngle{-0.025}; // Crossing angle in radians
   RotationX _rotAboutX;
   RotationY _rotAboutY;
   MomVector _vBoostToCoM;
   MomVector _vBoostToHoF;
 };
```

Functors need operator()

Undo ePIC afterburner
Applied to each particle automatically

Going further : multiple reactions

Maximise efficiency : 1 data read, multiple reactions

Become less read bound

- Make the most of multi-core processing
- Just need ROOT::EnableImplicitMT(8);

Difficult to do in a single script for loop

Here relatively simple :

Prior to indicing take copies of the RAD frame

Configure each for their own reaction and observables

Lazy execute all at once!

* note : currently tree snaphot is not lazy so this only works for histogramming type analysis

Automation : ReactionChannel

Utility class to do the index configuration
- just requires meson and baryon indices

Can use to analyse Pythia data, HepMC or ePIC

e.g M→ π-π+ B→ pπ0

```
ReactionChannel channel{rad,{211,-211},{2212,22,22}};
```

Multiple channels code

```
//identify a number of final states by pdg code
auto mesons={{211,-211}, {-211},    {113}, {211,111,-211} };
auto baryons={{2212},    {2212,211},{2212},{2212} };

//create base RAD object
rad::config::ePICReaction epic{"events","pythia.edm4eic.root"};

for(auto ipy = 0; ipy<mesons.size() ; ++ipy){
   //copy from base dataframe to allow lazy execution
   //on all final states
   auto rad = epic;

   //create a reaction channel which is defined in terms of the
   //meson and baryon decay products
   rad::config::ReactionChannel channel{rad,mesons[ipy],baryons[ipy]};

   //Define all kinematics we are interested in
   rad::rdf::MissMass(rad,"W","{scat_ele}");
   rad::rdf::MissMass(rad,"MissMassMeson","{scat_ele,idxMeson}");
   rad::rdf::MissMass2(rad,"MissMassMeson2","{scat_ele,idxMeson}");
   rad::rdf::MissMass2(rad,"MissMassBaryon2","{scat_ele,idxBaryon}");
```

ePIC Benchmarking

I need to make some benchmark tests for the lowQ2 Tagger

- Use the Pythia ReactionChannel Framework
- Use ePIC common benchmark format

  https://eicweb.phy.anl.gov/EIC/benchmarks/common_bench

Need to produce a json file :


```
"tests": [
    {
        "description": "Resolution benchmark for Q2",
        "name": "Q2",
        "quantity": "Resolution",
        "result": "pass",
        "target": "0.004645738386922428",
        "title": "Q2",
        "value": 0.0030971589246149523,
        "weight": 1.0
    },
```

ReactionBenchmarks

Class to automate benchmarking with RAD
- Requires truth matching

User defines histograms for observables

For each observable ReactionBenchmarks :

- generates tru and rec distributions
- generates acceptance histograms
- generates resolutions histograms
- produces plots on configurable canvases
- writes common_benchmark tests in json

Supplement : common_benchmark/benchmark_against.h
- Defines test target to be result from previous analysis

Coding ReactionBenchmarks

```
//create a copy of rad dataframe so we can apply different cuts etc
auto rad_tagger=rad;
//cut on LowQ2 Tagger acceptance region
rad_tagger.Filter("tru_theta[scat_ele]>3.12","tagger_cut");

ReactionBenchmarks incTaggHists{"IncTagger",mesons[ipy],baryons[ipy]};
//Set output directory for saving plots and tests
incTaggHists.SetOutDir(out_dir);
//define canvas formatting as 2 by 2 plots
arrange.push_back({2,2});

//Define each variable we wish to benchmark with histogram model
incTaggHists.AddVar("Q2",{"Q2","Q2",500000,0,1});
incTaggHists.AddVar("W",{"W","W",500,0,1.1*WMax});
incTaggHists.AddVar("MissPzMeson",{"MissPz","MPz",1200,-10,110});
incTaggHists.AddVarElement("scat_ele","pmag",{"epmag",
                           "Scattered Electron momentum",130,-1,12});

//Now declare all the histograms, add some cuts
incTaggHists.Declare(rad_tagger,
                     channel.CutParticleCondition("rec_pmag",">0.05")+
                     "&&rec_pmag[scat_ele]>0.1&&rec_theta[scat_ele]>3.1");

//assign to my vector of reaction benchmarks
allIncHists.push_back( incTaggHists );
```

Take rad dataframe
From slide 28

Create benchmark
Store final state pdgs

Must use previously defined variables

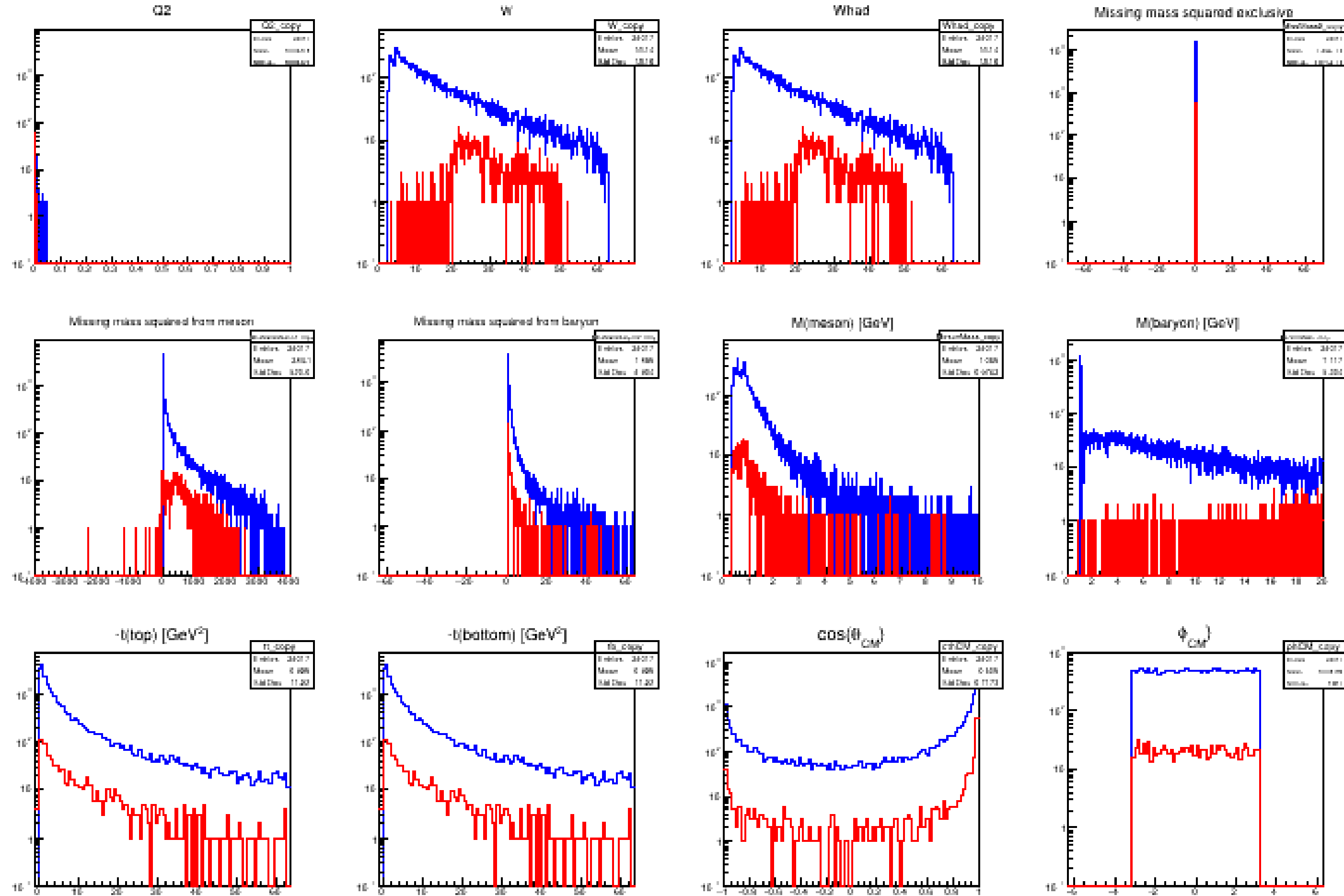
Can use particle momentum

Let ReactionBenchmarks Create all histograms

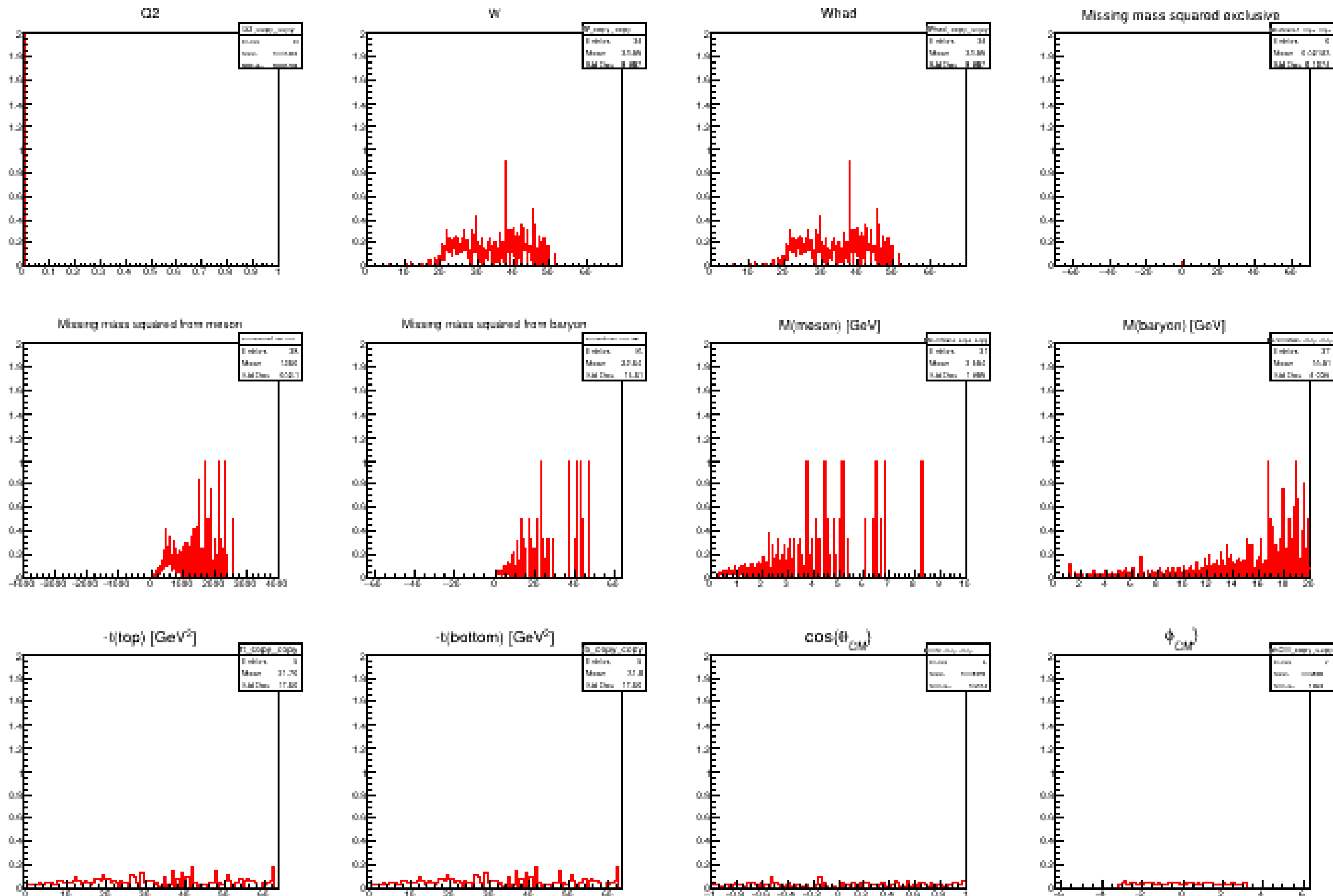
Store in vector for now
Must create all reactions and benchmarks first to use parallel lazy execution

Distributions

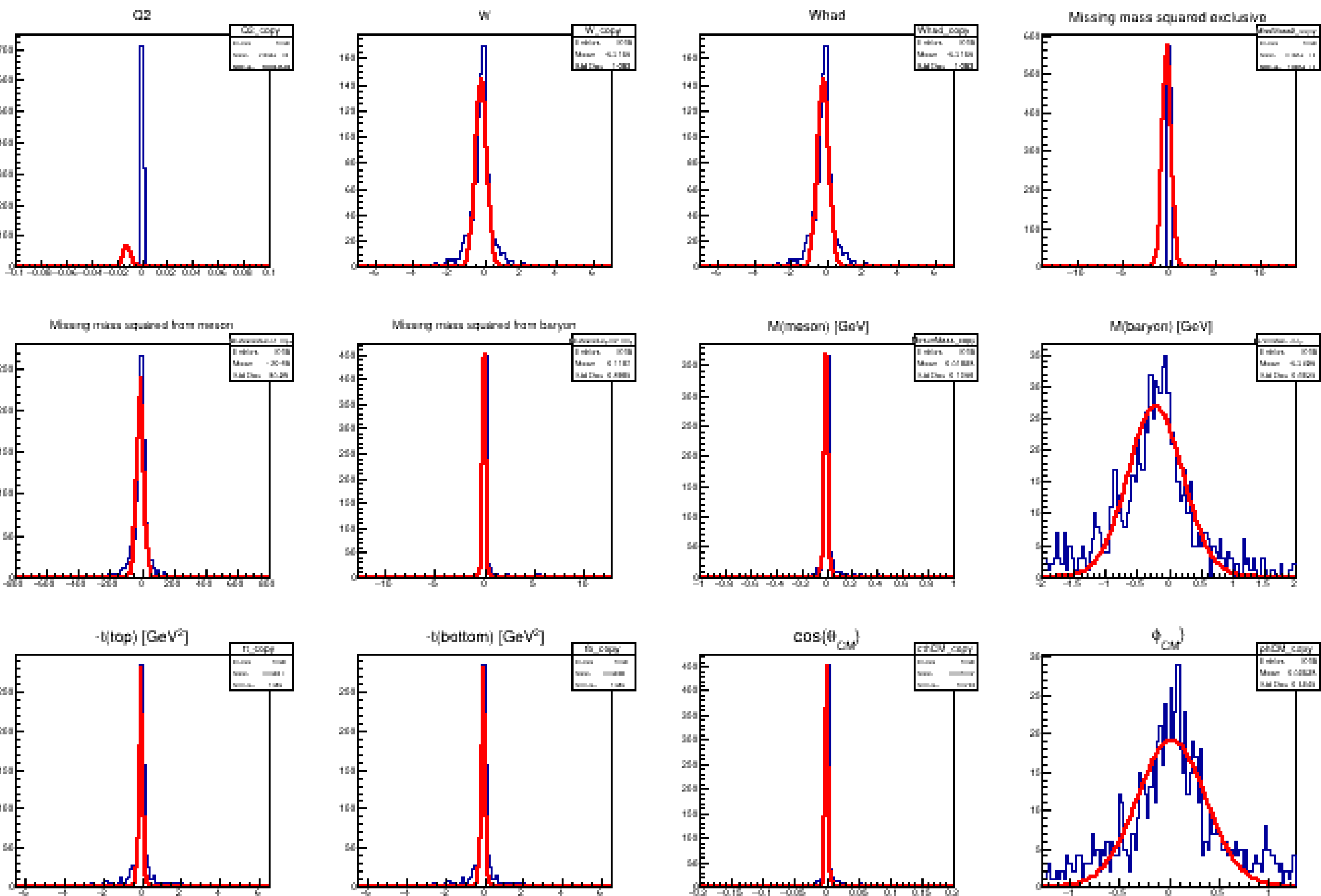
Pythia6 ; ePIC simulation/reconstruction; 10x100; lowQ2 e- ; π+π-



Efficiencies



Resolutions



common_benchmarks


```
"tests": [
    {
        "description": "Resolution benchmark for Q2",
        "name": "Q2",
        "quantity": "Resolution",
        "result": "pass",
        "target": "0.004645738386922428",
        "title": "Q2",
        "value": 0.003097158924614952З,
        "weight": 1.0
    },
    {
        "description": "Resolution benchmark for W",
        "name": "W",
        "quantity": "Resolution",
        "result": "pass",
        "target": "0.45497710832131266",
        "title": "W",
        "value": 0.3033180722142084,
        "weight": 1.0
    },
    {
        "description": "Resolution benchmark for Whad",
        "name": "Whad",
        "quantity": "Resolution",
        "result": "pass",
        "target": "0.45497710832131266",
        "title": "Whad",
        "value": 0.3033180722142084,
        "weight": 1.0
    },
    {
        "description": "Resolution benchmark for Missing mass squared exclusive",
        "name": "MissMass2",
        "quantity": "Resolution",
        "result": "pass",
        "target": "0.7384756350898487",
        "title": "Missing mass squared exclusive",
        "value": 0.49231709005989915,
        "weight": 1.0
    },
    {
        "description": "Resolution benchmark for Missing mass squared from meson",
        "name": "MissMassMeson2",
        "quantity": "Resolution",
        "result": "pass",
        "target": "33.416131530304796",
        "title": "Missing mass squared from meson",
        "value": 22.2774210202032,
        "weight": 1.0
    },
    {
        "description": "Resolution benchmark for Missing mass squared from baryon",
        "name": "MissMassBaryon2",
        "quantity": "Resolution",
        "result": "pass",
        "target": "0.21956725505406646",
        "title": "Missing mass squared from baryon",
        "value": 0.1463781700360443,
        "weight": 1.0
    },
    {
        "description": "Resolution benchmark for M(meson) [GeV]",
        "name": "MesonMass",
        "quantity": "Resolution",
        "result": "pass",
        "target": "0.020473300732896833",
        "title": "M(meson) [GeV]",
        "value": 0.013648867155264557,
        "weight": 1.0
    },
    {
        "description": "Resolution benchmark for M(baryon) [GeV]",
        "name": "BaryonMass",
        "quantity": "Resolution",
        "result": "pass",
        "target": "0.6511929613181693",
        "title": "M(baryon) [GeV]",
        "value": 0.4341286408787795,
        "weight": 1.0
    },
```

Conclusions

RDataFrame offers a appealing framework for data analysis

Collaborations can build a common scheme on top of this
- for example using edm4hep, edm4eic

Here we only use ROOT dependencies (simple to install)

We make the dataframe “Reaction Aware” by defining indices
User coding is minimised
Kinematic functions are reusable
Truth Matching is automated
Many reactions can be run in parallel on multi-cores

But the code behind the scenes is complicated and difficult to develop...

Title