Using PyROOT for analysis and simulations

JAROSLAV ADAM

202010.07

- PyROOT stands for using ROOT from Python; opening a `TBrowser`:

```
In [1]: from ROOT import TBrowser
In [2]: x = TBrowser()
```

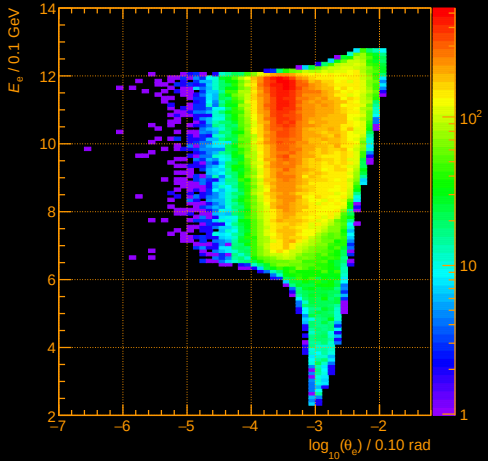
- Nice overview was given by Eduardo Rodrigues on August 26
- My experience with PyROOT will be shown on physics driven examples
- Outline:
  1. Histograms from a TTree
  2. Accessing TTree branches and creating a TTree
  3. Importing and compiling existing ROOT macros
  4. Reading a more complex data structures from a TTree
  5. TF1 with function binding
  6. Sophisticated RooFit model with use of classes
  7. Navigating in inheritance hierarchy

- Links to resources
  - Section in ROOT User's Guide
  - Online ROOT manual
  - Slides by Pere Mato and Danilo Piparo
- Prerequisites to using PyROOT
  - Variable PYTHONPATH should point to `lib` in ROOT build
  - Example from EIC nodes:

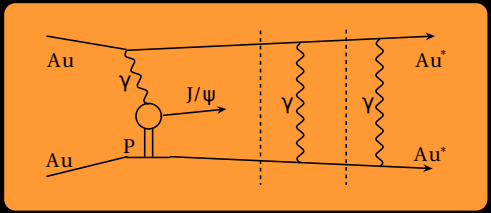
```
$ echo $PYTHONPATH
/afs/rhic.bnl.gov/eic/restructured/env/EIC2020a/root6/lib
```

- Energy and polar angle for electrons hitting the low-Q^2 tagger in Geant4 output
- The tree has energy el_en, log of angle el_theta and hit information Q2_hit

```
#open file and get the tree
inp = TFile.Open("lmon_py_18x275.root")
tree = inp.Get("DetectorTree")

#2D histogram
hE = TH2D("hE", "hE", 58, -7, -1.2, 120, 2, 14)

#fill the histogram with Draw methot
tree.Draw("el_en:el_theta", "Q2_hit==1")

#axis labels and plot margins here

#make the plot
hE.Draw("colz")
```



Need to access an individual TTree branch

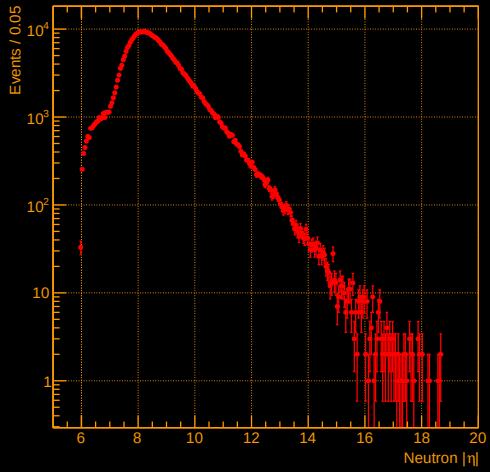


- The nOOn event generator provides neutrons emitted by excited Au* nuclei in the process of J/ψ photoproduction
- The photoproduction itself is simulated by Starlight generator, the nOOn adds the neutrons to e^+e^- pairs from J/ψ decay
- Neutron multiplicity could reach up to all neutrons in gold
- nOOn output is a `TClonesArray` of `TParticle` with the e^+e^- followed by the neutrons

Comput.Phys.Commun. 253 (2020) 107181, Comput.Phys.Commun. 212 (2017) 258-268

- Want to get pseudorapidity for all neutrons generated by nOOn
- The nOOn writes particles clones array to its output

```
#get the nOOn particles
particles = TClonesArray("TParticle", 200)
tree.SetBranchAddress("particles", particles)

#loop over the tree
for iev in xrange(tree.GetEntriesFast()):
    tree.GetEntry(iev)

    #particles loop
    for imc in xrange(particles.GetEntriesFast()):
        part = particles.At(imc)
        #select the neutrons
        if part.GetPdgCode() != 2112: continue
        #fill absolute pseudorapidity
        hEta.Fill( abs(part.Eta()) )

#make the plot
hEta.Draw()
```



- Takes ∼1 sec for 100k events

- A generator provides text file with p_T and mass as its first two columns
- The task is to create a TTree with pt and mass branches and fill it from the text file
- Done by making a C structure and addressing variables in it

- Make the tree with its branches:

```
#output file and tree
out = TFile("out.root", "recreate")
tree = TTree("tree", "tree")

#C structure to hold tree variables
cmd = "struct tree_str {Float_t pt, mass;};"
gROOT.ProcessLine( cmd )

#instance of the C structure
tree_out = ROOT.tree_str()

#make tree branches
#pointing to variables in the C structure
tree.Branch("pt", AddressOf(tree_out, "pt"), "pt/F")
tree.Branch("mass", AddressOf(tree_out, "mass"), "mass/F")
```

- Read the text file and write the tree:

```
#input loop
for line in open("gen.txt"):
    #columns for the line
    col = line.split()

    #assign branch values
    tree_out.pt = float(col[0])
    tree_out.mass = float(col[1])

    #fill the tree
    tree.Fill()

#write and close
tree.Write()
out.Close()
```

Importing a compiled .C macro

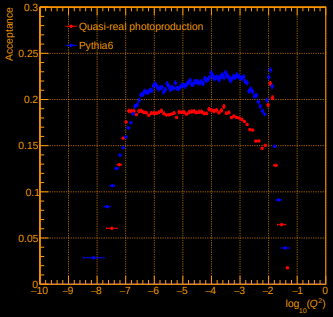
- Combined acceptance of both low-Q^2 taggers in bins of variable size
- Iterative procedure for bins of a given maximal binomial error in each bin
- A class is compiled to make it fast, plot is done in Python

- Class to compile:

```
class acc_Q2_calc {
  TTree *tree; //input tree
public:
  acc_Q2_calc(TTree *t): tree(t) {
  }

  TGraphAsymmErrors get_acc() {
    //the procedure comes here

    //resulting acceptance
    TGraphAsymmErrors acc(&s, &a);
    return acc;
  }
};
```

- Run and make the plot:

```
#build the class
gROOT.ProcessLine(".L acc_Q2_calc.C+")

#instance of it with input tree
calc = ROOT.acc_Q2_calc(tree)

#run and get the acceptance
acc = calc.get_acc()

#draw it
acc.Draw()
```



github.com/adamjaro/lmon/blob/master/macro/plots_Q2_acc.py
github.com/adamjaro/lmon/blob/master/macro/acc_Q2_calc.C

- A set of C++ `vectors` could be used to store the array of hits from a detector
- Minimal example is deposited energy and position of the current hit
- There is a branch for each `vector` in output `TTree`

- Hits in calorimeter description:

```
// hit energy, GeV
std::vector<Float_t> fHitEn;

// hit position in x, mm
std::vector<Float_t> fHitX;
// hit position in y, mm
std::vector<Float_t> fHitY;
// hit position in z, mm
std::vector<Float_t> fHitZ;
```

- Writing the hits in stepping:

```
//hit energy and position
G4double en_step = track->GetTotalEnergy();
const G4ThreeVector hp = step->GetPostStepPoint()->GetPosition();

//add the hit
fHitEn.push_back( en_step/GeV );
fHitX.push_back( hp.x() );
fHitY.push_back( hp.y() );
fHitZ.push_back( hp.z() );
```

github.com/adamjaro/lmon/blob/master/src/BoxCalV2.cxx
github.com/adamjaro/lmon/blob/master/include/BoxCalV2.h

- A Python class can represent the calorimeter hits; position in x and y is omitted
- Hits collection:

```
class BoxCalV2Hits:
    def __init__(self, name, tree):
        #hit representation in the tree
        self.en = std.vector(float)()
        self.hx = std.vector(float)()

        #energy and position based on detector name
        tree.SetBranchAddress(name+"_HitEn", self.en)
        tree.SetBranchAddress(name+"_HitX", self.hx)

        #interface to the hit by a nested class
        self.hit = self.Hit()

    def GetHit(self, ihit):
        #get the hit at a given 'ihit'
        self.hit.en = self.en.at(ihit)
        self.hit.x = self.hx.at(ihit)

        return self.hit
```

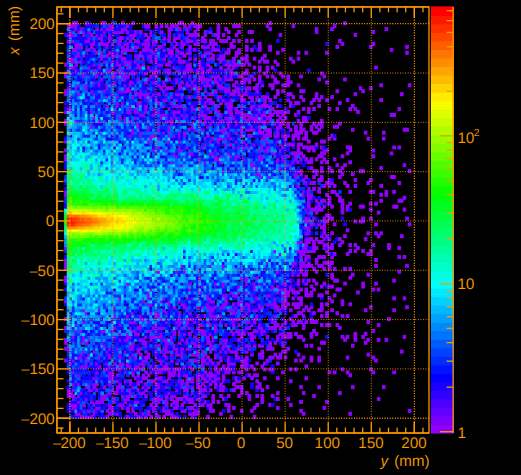
- Nested class for a hit:

```
class Hit:
    #implementation for the hit interface
    def __init__(self):
        self.en = 0. # GeV
        self.x = 0. # mm

    def GlobalToLocal(self, x0):
        #transform to local detector coordinates
        self.x -= x0
```

github.com/adamjaro/lmon/blob/master/macro/BoxCalV2Hits.py

- Interface to the hits allows for manipulation and selections

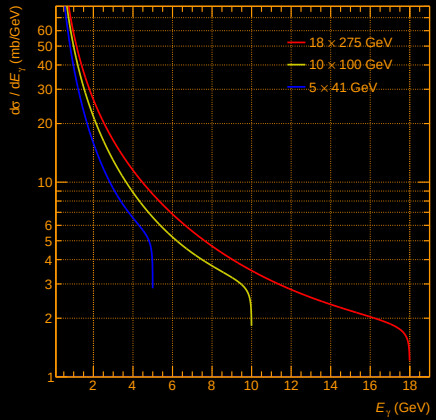
```
#Tagger 1 hits
hits = BoxCalV2Hits("lowQ2s1", tree)

#loop over input tree
for ievt in xrange(tree.GetEntriesFast()):
    tree.GetEntry(ievt)

    #loop over the hits
    for ihit in xrange(hits.GetN()):
        hit = hits.GetHit(ihit)

        #move to local tagger coordinates
        hit.GlobalToLocal(xpos, 0, zpos)

        #apply selection for z and energy
        if hit.z < zmin: continue
        if hit.en < emin: continue

        #hits distribution in x and y
        hXY.Fill(hit.x, hit.y)

#plot the hits in x and y
hXY.Draw()
```



- Parametrization is given in terms of electron and proton beam energy E_e and E_p

$$\frac{d\sigma}{dE_\gamma} = 4\alpha r_e^2 \frac{E'_e}{E_\gamma E_e}\left(\frac{E_e}{E'_e} + \frac{E'_e}{E_e} - \frac{2}{3}\right)\left(\ln\frac{4E_pE_eE'_e}{m_pm_eE_\gamma} - \frac{1}{2}\right)$$

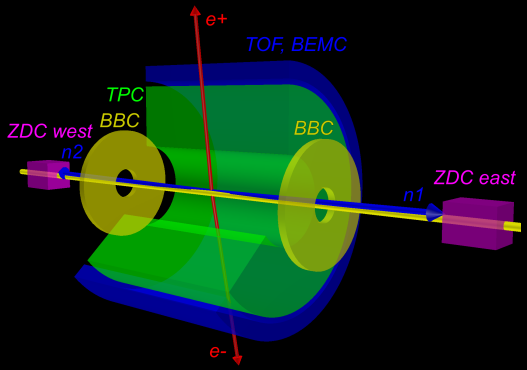
```python
#implementation as TF1 with binding to member function:
class gen_zeus:
    def __init__(self):
        #parametrization for dSigma/dE_gamma from eq1
        self.dSigDe = TF1("dSigDe", self.eq1, self.emin, self.Ee)

    def eq1(self, x):
        #E_gamma as input argument
        Eg = x[0]
        #electron and proton energy
        Ee = self.Ee
        Ep = self.Ep
        #scattered electron Ee'
        Escat = Ee - Eg

        t1 = Escat/(Eg*Ee)
        t2 = (Ee/Escat) + (Escat/Ee) - 2./3
        t3 = TMath.Log(4*Ep*Ee*Escat/(self.mep*Eg)) - 1./2

        return self.ar2*t1*t2*t3 # sigma at a given E_gamma
```



github.com/adamjaro/eic-lgen/blob/master/gen_zeus.py

2D fit model for forward neutrons

- Very forward neutrons from process on page 5 reach Zero-Degree Calorimeters ZDC
- There could be one or more neutrons n1 and n2 in each ZDC
- Fit $f_{\mathrm{ZDC}}(e, w)$ has to separate one neutron and more neutrons from both ZDC signals
- Gaussians $G_{1\mathrm{n,E}}$ and $G_{1\mathrm{n,W}}$ for one neutron in east and west ZDC
- Reversed Crystal Balls $\mathrm{CB}_{2+\mathrm{n,E}}$ and $\mathrm{CB}_{2+\mathrm{n,W}}$ for two and more neutrons
- Constants $c_{1,2,3,4}$ are normalization for individual cases



$$f_{\mathrm{ZDC}}(e, w) = \underbrace{c_1 G_{1\mathrm{n,E}} \cdot G_{1\mathrm{n,W}}}_{\text{one neutron each}} + \underbrace{c_2 G_{1\mathrm{n,E}} \cdot \mathrm{CB}_{2+\mathrm{n,W}} + c_3 \mathrm{CB}_{2+\mathrm{n,E}} \cdot G_{1\mathrm{n,W}}}_{\text{more neutrons in one ZDC and one in the other}} + \underbrace{c_4 \mathrm{CB}_{2+\mathrm{n,E}} \cdot \mathrm{CB}_{2+\mathrm{n,W}}}_{\text{more neutrons in both}}$$

Result of 2D fit to ZDC signal

- Implemented by RooFit classes RooGaussian, RooCBShape, RooProdPdf and RooAddPdf

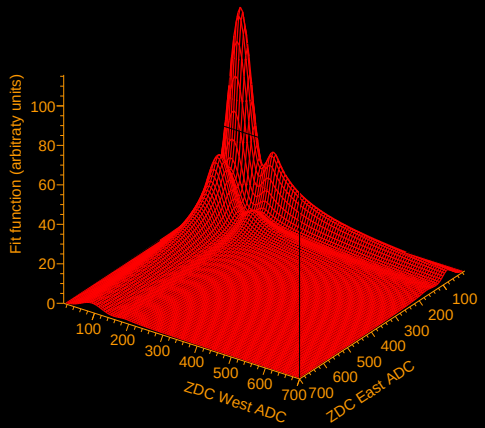


Figure: Fit function $f_{\mathrm{ZDC}}(e, w)$

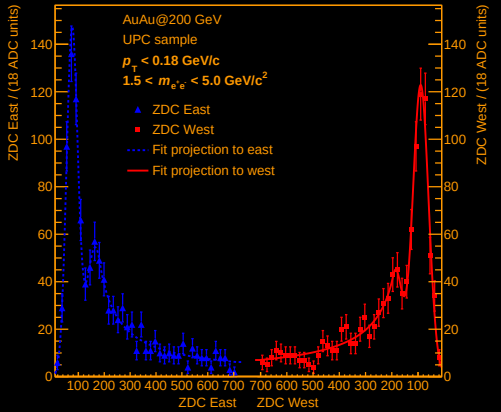


Figure: Projection to east and west ZDC

github.com/adamjaro/star-upc/blob/master/macro/zdc_plots/fit_func.py

- Begins with auxiliary classes for Guassian and reversed Crystal Ball

```python
# One neutron Gaussian
class Gauss:
  def __init__(self, adc, name):
    #input ADC values
    self.adc = adc

    #mean
    mean_nam = "mean_1n_" + name
    self.mean_1n = RooRealVar(mean_nam,\
                   mean_nam, 40., 120.)
    #sigma
    sigma_nam = "sigma_1n_" + name
    self.sigma_1n = RooRealVar(sigma_nam,\
                    sigma_nam, 0., 100.)
    #Gaussian pdf
    gauss_nam = "gauss_1n_" + name
    self.gauss_1n = RooGaussian(gauss_nam,\
                    gauss_nam, self.adc,\
                    self.mean_1n,\
                    self.sigma_1n)
```

```python
# Two and more neutrons reversed Crystal Ball
class CrystalBall:
  def __init__(self, adc, name):
    #input ADC values
    self.adc = adc
    #mean
    mean_nam = "mean_2n_" + name
    self.mean_2n = RooRealVar(mean_nam, mean_nam, 150., 200.)
    #sigma
    sigma_nam = "sigma_2n_" + name
    self.sigma_2n = RooRealVar(sigma_nam, sigma_nam, 0., 100.)
    #alpha
    alpha_nam = "alpha_2xn_" + name
    self.alpha_2xn = RooRealVar(alpha_nam, alpha_nam, -10., 0.)
    #n
    n_nam = "n_2xn_" + name
    self.n_2xn = RooRealVar(n_nam, n_nam, 0., 20.)
    #CrystalBall
    cb_name = "cb_2xn_" + name
    self.cb_2xn = RooCBShape(cb_name, cb_name, self.adc,\
                  self.mean_2n, self.sigma_2n,\
                  self.alpha_2xn, self.n_2xn)
```

github.com/adamjaro/star-upc/blob/master/macro/zdc_plots/fit_func.py
github.com/adamjaro/star-upc/blob/master/macro/zdc_plots/zdc_fit_2d.py

Model for 2D fit to ZDC signal

- The model contains two instances of `Gauss` and `CrystalBall` each for east and west ZDCs
- Implements the $f_{\mathrm{ZDC}}(e, w)$ fit function with `RooProdPdf` and `RooAddPdf`
- Only the fit function is visible, all complexity is hidden

```
#input ACD distributions
adc_east = RooRealVar("zdc_adc_east",\
           "ZDC ADC east", adc_min, adc_max)
adc_west = RooRealVar("zdc_adc_west",\
           "ZDC ADC west", adc_min, adc_max)

#data from input tree
data = RooDataSet("data", "data",\
       tree, RooArgSet(adc_east, adc_west))

#create the model
model = Model2D(adc_east, adc_west)

#make the fit with f_ZDC(e,w)
res = model.fZDCew.fitTo(data)
```

github.com/adamjaro/star-upc/blob/master/macro/zdc_plots/fit_func.py
github.com/adamjaro/star-upc/blob/master/macro/zdc_plots/zdc_fit_2d.py

- Need to log result of a fit to a text file
- Should include covariance and correlation matrix
- Result of a graph fit with TF1 function is object of `TFitResult`
- Suitable `Print()` exists only in `Fit::FitResult` base class
- Here reference is taken to a stream to further manipulate with the result

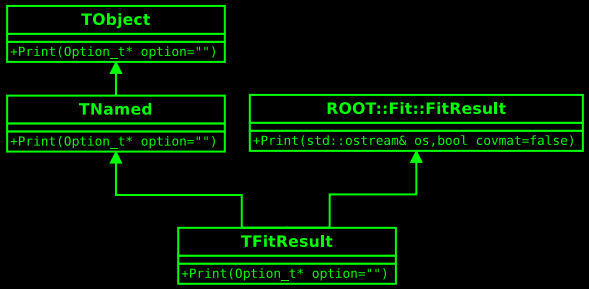


Figure: Inheritance diagram for `TFitResult`

Function call to a specific base

- Call to `Print()` of `Fit::FitResult` for an object of `TFitResult` derived class

C++ approach

01 Function call for base subobject

02 Upcast to specific base

```
void LogFit(TFitResult *res) {

  //1: subobject of base class
  stringstream s1;
  res->Fit::FitResult::Print(s1, kTRUE);

  //2: upcast
  Fit::FitResult *fit_res =
    dynamic_cast<Fit::FitResult*>( res );

  stringstream s2;
  fit_res->Print(s2, kTRUE);

  //do the work with stream
}
```

Python approach

01 Derived object as argument to base method

02 Proxy base object for method call

```
def LogFit(res):

    #1: call to base FitResult::Print method
    s1 = std.stringstream()
    Fit.FitResult.Print(res, s1, True)

    #2: proxy to Fit.FitResult (beyond TObject)
    fit_res = super(TObject, res)

    s2 = std.stringstream()
    fit_res.Print(s2, True)

    #do the work with stream
```

- I found PyROOT a good substitute to ROOT macros:
  1. Much less to type when creating new code
  2. Much less to read when getting back to previous work
- Easy to compile and load a piece of C++ code if performance is an issue
- No direct function overloading but the code is cleaner in the end
- PyROOT is a way for nice Python syntax while having access to more complex ROOT features like Lorentz kinematics, TF1/2 functions, RooFit and data structures in trees