Table of Contents

ATLAS Tutorial @ Naples

4th may 2018

Indice

Introduction

L'idea di questo tutorial è di fornire gli strumenti per eseguire semplici operazioni sugli oggetti fisici di interesse conservati in una nTupla di root. Utilizzando un `MakeClass` di root è possibile accedere rapidamente alle variabili presenti nel tree e scrivere semplici algoritmi in C++ nella struttura di codice che si ottiene.

Informazioni preliminari.

Per accedere alla UI bisogna usare il seguente comando

```
$ ssh -XY -p5022 ILMIOUSER@atlasui03.na.infn.it
```

setup di root:

```
$ setupATLAS
$ lsetup "root 6.14.04-x86_64-slc6-gcc62-opt"
```

Informazioni preliminari per il tutorial

Per i dati raccolti nel 2017 si può usare la semplice ntupla:

```
/data3/agiannini/TutorialATLAS_4mag18/Tree/data17.root
```

per copiare in locale questo file:

```
$ mkdir myDirectory
$ cd myDirectory
$ cp /data3/agiannini/TutorialATLAS_4mag18/Tree/data17.root .
```

Per aprire il file root:

```
$ root -l data17.root
```

per vedere cosa c'è all'interno del file root:

```
$ .ls
```

per vedere quali variabili sono presenti nel tree:

```
$ Nominal->Print()
```

per usare il MakeClass:

```
$ Nominal->MakeClass ("myClass")
```

Il MakeClass crea una struttura di codice in cui poter implementare una semplice analisi; inoltre, si occupa in maniera semplice della aperture del tree e della lettura delle variabili. E' possibile vedere quali variabili è possibile usare guardano nel .h che è stato creato. In questo tutorial possono essere interessanti variabili del tipo:

```
vector<double> electron_pt;
vector<double> electron_eta;
```

```
vector<double> electron_phi;
vector<double> electron_E;

...

vector<double> *muon_pt;
vector<double> *muon_eta;
vector<double> *muon_phi;
vector<double> *muon_E;
```

How to run

```
root -l
.L myClass.C
myClass NOMEAPIACERE
NOMEAPIACERE.Loop()
```

Alcune info sulla classe vector di C++

La classe `vector` risulta estremamente più efficiente rispetto agli array. Il tipo del vettore va specificato tra `<>`. Di seguito alcuni esempi

```
#include <vector>
int main()
{
 // Un vettore dinamico
 std::vector <double> v_dyna;

 // Un vettore con 10 elementi (inizializzati tutti a 0)
 std::vector <int> vint_10_elements(10);

 // Un vettore con 10 elementi, inizializzati a 90
 std::vector <int> vint_10(10, 90);

 // Un vettore inizializzato con il contenuto di un altro
 std::vector<int> vcopy(vint_10_elements);

 // Un vettore con 5 valori presi da un altro
 std::vector<int> vcopy_5(vint_10.begin(),vint_10.begin()+5);

}
```

Se voglio inizializzarlo con dei valori iniziali, posso scrivere così:

```
std::vector<int> v = {1, 2, 3, 4, 5};
//(attenzione: questo tipo di inizializzazione è disponibile solo nel nuovo standard C++11).
```

Per accedere all i-esimo elemento, si può fare in diversi modi:

```
std::cout << v[2] << std::endl; //come negli array
std::cout << v.at(2) << std::endl;
```

Anche con il vettore, se cercate di accedere un elemento al di fuori della dimensione dell array, otterrete un errore.

Per aggiungere un elemento ad un vettore basta fare

```
v.push_back(10);

// per avere la dimensione
std::cout << v.size() << std::endl;
```

Un loop con i vettori si può fare in diversi modi:

```
std::vector<int> v = {0, 1, 2, 3, 4, 5};

// ciclo for con iteratori
for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)
{
std::cout << *it << "\n"; //notate che il contenuto del vettore è *it
}


 // iterazione basata su indici
for (int i = 0; i<v.size();i++)
{
std::cout << v.at(i) << "\n"; //si accede all'i-esimo evento
}
```

Altre info qui

ROOT tips and tricks

- ROOT intro slide
- Breve esempi su come utilizzare la classe degli Istogrammi
- Capitolo sul fit
- Un pò di estetica: legenda, opzioni grafiche, colori, stile
- Classe dei TLorentzVector

Tutorial

Electrons

Il .C ottenuto con il MakeClass avrà una struttura del tipo:

```
#define Tutorial_emu_cxx
#include "Tutorial_emu.h"
#include <TH2.h>
#include <TStyle.h>
#include <TCanvas.h>
void Tutorial_emu::Loop(){

//  In a ROOT session, you can do:
//     root> .L Tutorial_emu.C
//     root> Tutorial_emu t
//     root> t.GetEntry(12);
// Fill t data members with entry number 12
//     root> t.Show();// Show values of entry 12
//     root> t.Show(16);// Read and show values of entry 16
//     root> t.Loop();// Loop on all entries
//
//    This is the loop skeleton where:
//   jentry is the global entry number in the chain
//   ientry is the entry number in the current Tree
// Note that the argument to GetEntry must be:
//   jentry for TChain::GetEntry
//   ientry for TTree::GetEntry and TBranch::GetEntry
//
//      To read only selected branches, Insert statements like:

// METHOD1:
//   fChain->SetBranchStatus ("*",0); // disable all branches
//   fChain->SetBranchStatus ("branchname",1); // activate branchname

// METHOD2: replace line
//   fChain->GetEntry (jentry);      //read all branches
//by b_branchname->GetEntry (ientry); //read only this branch

if (fChain = 0) return;
Long64_t nentries = fChain->GetEntriesFast ();

for (Long64_t jentry=0; jentry&lt;nentries;jentry++) {
Long64_t ientry = LoadTree (jentry);

if (ientry &lt; 0) break;

nb = fChain->GetEntry (jentry); nbytes + nb;

// if (Cut(ientry) &lt; 0) continue;

}//end loop over events

}
```

Supponiamo di volere ottenere un istogramma di una certa variabile presente nel tree, ad esempio l'impulso trasverso degli elettroni, e riempire un istogramma evento per evento con la variabile di interesse. Creiamo un istogramma all'esterno del loop su tutti gli eventi:

```
TH1F *h_pT_electrons = new TH1F ("h_pT_electrons", "h_pT_electrons", 100, 0, 500);
```

Nel caso del pT degli elettroni abbiamo delle variabili vettoriali; per ogni evento abbiamo una collezione di elettroni, nel tree, quindi, avremo una collezione di valori di pT. Per accedere al pT dello i-esimo elettrone basta utilizzare:

```
int nelectrons = electron_pt -> size();

...

for(int e=0; e<nelectrons; e++){

cout << electron_pT -> at(e) << endl;

}
```

e riempire l'histo per ogni elettrone:

```
h_pT_electrons -> Fill( electron_pT -> at(e) );
```

al termine del loop avremo l'istogramma a disposizione e possiamo farlo disegnare in interattivo da root:

```
h_pT_electrons -> Draw();
```

oppure salvarlo in un file .root per conservarlo ed utilizzarlo in un secondo momento:

```
TString name_file = "file_out.root";

TFile *file_out = new TFile(name_file, "recreate");

//Scrivo l'istogramma su file
h_pT_electrons -> Write();

//Scrivo e chiudo il file
file_out->Write();
file_out->Close();
```

per salvare l'histo in formato pdf:

```
TCanvas *c_pT_electrons = new TCanvas("c_pT_electrons", "c_pT_electrons", 1);
h_pT_electons -> Draw();
c_pT_electrons -> Print("pT_electorns.pdf");
```

Se vogliamo, ad esempio, ottenere un istogramma con il numero di elettroni che abbiamo per ogni evento possiamo usare la size del vector dei pT; all'esterno del loop aggiungiamo:

```
TH1F *h_nelectrons = new TH1F ("h_nelectrons", "h_nelectrons", 15, 0, 15);
```

all'interno del loop:

```
h_nelectrons -> Fill( electron_pt -> size() );
```

e vediamo l'istogramma all'esterno del loop:

```
h_nelectrons -> Write();
```

Proviamo ora a fare una cosa analoga con le altre variabili cinematiche degli elettroni. Aggiungiamo le seguenti linee prima del loop su tutti gli eventi:

```
TH1F *h_eta_electrons = new TH1F ("h_eta_electrons", "h_eta_electrons", 40, -5, 5);
TH1F *h_phi_electrons = new TH1F ("h_phi_electrons", "h_phi_electrons", 40, -4, 4);
TH1F *h_E_electrons = new TH1F ("h_E_electrons", "h_E_electrons", 100, 0, 500);
```

all'interno del loop sugli eventi:

```
for(int e=0; e<nelectrons; e++){
  h_eta_electrons -> Fill( electron_eta -> at(e) );
  h_phi_electrons -> Fill( electron_phi -> at(e) );
  h_E_electrons -> Fill( electron_E -> at(e) );

}// end loop over electrons.
```

dopo il loop sugli eventi:

```
h_eta_electrons -> Write();
h_phi_electrons -> Write();
h_E_electrons -> Write();
```

Supponiamo ora di volere identificare per ciascun evento l'elettrone con il pT più elevato fra tutti e poi l'elettrone con il secondo pT più elevato:

```
for(int e=0; e<nelectrons; e++){
 if( electron_pt -> at(e) > pT_leading ){
 e_leading = e;
 pT_leading = electron_pt -> at(e);
 }
}

for(int e=0; e<nelectrons; e++){

if(e!=e_leading){
if( electron_pt -> at(e) > pT_sub_leading ){
  e_sub_leading = e;
  pT_sub_leading = electron_pt -> at(e);
  }
 }
}

h_pT_e_leading -> Fill( electron_pt -> at(e_leading) );
h_pT_e_sub_leading -> Fill( electron_pt -> at(e_sub_leading) );
```

Supponiamo ora di volere rispondere alla domanda: qual è la probabilità di trovare un elettrone con un pT > 150 GeV? oppure con eta compreso tra -1.1 e 1.1? Per fare questo possiamo utilizzare dei contatori. Definiamo all'esterno del loop:

```
int number_of_electrons = 0;
```

e per ciascun evento contiamo quanti elettroni abbiamo:

```
for(int e=0; e<nelectrons; e++){
number_of_electrons += 1;
}
```

terminato il loop sugli eventi possiamo ad esempio far stampare a terminale il numero di elettroni che abbiamo contato:

```
cout << "Number of electrons:  " << number_of_electrons << endl;
```

con la stessa idea contiamo ora quanti elettroni troviamo che soddisfino un certo taglio cinematico, ad esempio, pT>150 GeV. All'esterno del loop:

```
int number_of_electrons_cut = 0;
```

e per ciascun evento contiamo quanti elettroni abbiamo:

```
for(int e=0; e<nelectrons; e++){
      if( electron_pt -> at(e) > 150 ) number_of_electrons_cut += 1;
}
```

terminato il loop sugli eventi possiamo ad esempio far stampare a terminale il numero di elettroni che abbiamo contato:

```
cout << "Number of electrons with pT&gt;150 GeV:  " << number_of_electrons_cut << endl;
```

a questo punto è immediato calcolare la probabilità di trovare un elettrone con pT > 150 GeV:

```
cout << "Fraction of electrons with pT&gt;150 GeV:  " << number_of_electrons / number_of_electron
```

In generale è possibile ottenere la stessa informazione utilizzando un istogramma e calcolare l'integrale totale e l'integrale nello specifico range cinematico cui si è interessati; il metodo dei TH1 da utilizzare è:

```
h_pT_electrons -> Integral();
h_pT_electrons -> Integral( bin_a, bin_b );
```

Muons

Supponiamo di volere ottenere un istogramma di una certa variabile presente nel tree, ad esempio l'impulso trasverso dei muoi, e riempire un istogramma evento per evento con la variabile di interesse. Creiamo un istogramma all'esterno del loop su tutti gli eventi:

```
TH1F *h_pT_muons = new TH1F ("h_pT_muons", "h_pT_muons", 100, 0, 500);
```

Nel caso del pT dei muoni abbiamo delle variabili vettoriali; per ogni evento abbiamo una collezione di muoni, nel tree, quindi, avremo una collezione di valori di pT. Per accedere al pT dello i-esimo muone basta utilizzare:

```
int nelectrons = muon_pt -> size();

...

for(int m=0; m<muons; m++){
 cout << muon_pt -> at(m) << endl;
}
```

e riempire l'histo per ogni elettrone:

```
h_pT_muons -> Fill( muon_pt -> at(m) );
```

al termine del loop avremo l'istogramma a disposizione e possiamo farlo disegnare in interattivo da root:

```
h_pT_muons -> Draw();
```

oppure salvarlo in un file .root per conservarlo ed utilizzarlo in un secondo momento (va definito fuori dal loop sugli eventi):

```
TString name_file = "file_out.root";
TFile *file_out = new TFile(name_file, "recreate");
h_pT_muons -> Write();
```

per salvare l'histo in formato pdf:

```
TCanvas *c_pT_muons = new TCanvas("c_pT_muons", "c_pT_muons", 1);
h_pT_muons -> Draw();
c_pT_muons -> Print("pT_muons.pdf");
```

Se vogliamo, ad esempio, ottenere un istogramma con il numero di muoni che abbiamo per ogni evento possiamo usare la size del vector dei pT; all'esterno del loop aggiungiamo:

```
TH1F *h_muons = new TH1F ("h_nmuons", "h_nmuons", 15, 0, 15);
```

all'interno del loop:

```
h_nmuons -> Fill( muon_pt -> size() );
```

e vediamo l'istogramma all'esterno del loop:

```
h_nmuons -> Write();
```

Proviamo ora a fare una cosa analoga con le altre variabili cinematiche dei muoni. Aggiungiamo le seguienti linee prima del loop su tutti gli eventi:

```
TH1F *h_eta_muons = new TH1F ("h_eta_muons", "h_eta_muons", 40, -5, 5);
TH1F *h_phi_muons = new TH1F ("h_phi_muons", "h_phi_muons", 40, -4, 4);
TH1F *h_E_muons = new TH1F ("h_E_muons", "h_E_muons", 100, 0, 500);
```

all'interno del loop sugli eventi:

```
for(int m=0; m<nmuons; m++){

h_eta_muons -> Fill( muon_eta -> at(m) );
h_phi_muons -> Fill( muon_phi -> at(m) );
h_E_muons -> Fill( muon_E -> at(m) );

}// end loop over muons.
```

dopo il loop sugli eventi:

```
h_eta_muons -> Write();
h_phi_muons -> Write();
h_E_muons -> Write();
```

Supponiamo ora di volere identificare per ciascun evento il muone con il pT più elevato fra tutti e poi il muone con il secondo pT più elevato:

```
for(int m=0; m<nmuons; m++){
if( muon_pt -> at(m) > pT_leading ){
mu_leading = m;
pT_leading = muon_pt -> at(m);
}

}

for(int m=0; m<nmuons; m++){

if(m!=mu_leading){
if( muon_pt -> at(m) > pT_sub_leading ){
mu_sub_leading = m;
pT_sub_leading = muon_pt -> at(m);
}
}
}

h_pT_mu_leading -> Fill( muon_pt -> at(mu_leading) );
h_pT_mu_sub_leading -> Fill( muon_pt -> at(mu_sub_leading) );
```

Supponiamo ora di volere rispondere alla domanda: qual è la probabilità di trovare un muone con un pT > 150 GeV? oppure con eta compreso tra -1.1 e 1.1?

Per fare questo possiamo utilizzare dei contatori. Definiamo all'esterno del loop:

```
int number_of_muons = 0;
```

e per ciascun evento contiamo quanti muoni abbiamo:

```
for(int m=0; m<nmuons; m++){
number_of_muons += 1;
```

```
}
```

terminato il loop sugli eventi possiamo ad esempio far stampare a terminale il numero di muoni che abbiamo contato:

```
cout << "Number of muons: " << number_of_muons << endl;
```

con la stessa idea contiamo ora quanti muoni troviamo che soddisfino un certo taglio cinematico, ad esempio, pT>150 GeV. All'esterno del loop:

```
int number_of_muons_cut = 0;
```

e per ciascun evento contiamo quanti muoni abbiamo:

```
for(int m=0; m<nmuons; m++){
if( muon_pt -> at(m) > 150 ) number_of_muons_cut += 1;
}
```

terminato il loop sugli eventi possiamo ad esempio far stampare a terminale il numero di muoni che abbiamo contato:

```
cout << "Number of muons with pT&gt;150 GeV: " << number_of_muons_cut << endl;
```

a questo punto è immediato calcolare la probabilità di trovare un muone con pT > 150 GeV:

```
cout << "Fraction of muons with pT&gt;150 GeV: " << number_of_muons / number_of_muons_cut << endl
```

In generale è possibile ottenere la stessa informazione utilizzando un istogramma e calcolare l'integrale totale e l'integrale nello specifico range cinematico cui si è interessati; il metodo dei TH1 da utilizzare è:

```
h_pT_muons -> Integral();
h_pT_muons -> Integral( bin_a, bin_b );
```

Invariant mass of a system

Proviamo ora a ricostruire la massa invariante di coppie di 2 elettroni oppure di 2 muoni. Per fare utilizziamo la classe `TLorentzVector` di root. Con questa classe è possibile gestire e fare operazioni con i qudri-vettori di Lorentz. In generale, un quadri-vettore è completamente definito dalle 4 componenti (E, px, py, pz) oppure, equivalentemente, dalle 4 componenti (E, pT, eta, phi). Se abbiamo un muone possiamo definere un TLorentzVector nel seguente modo:

```
TLorentzVector muon;
muon.SetPtEtaPhiE(100, 1.5, 0.5, 120);
```

questo TLorentzVector ci descreverà un muone con un pT di 100 GeV, eta pari a 1.5, ecc... utilizzando i vari metodi di tale classe è possibile calcolare rapidamente le altre variabili, come ad esempio il modulo del tri-impulso o la massa della particella:

```
muon.P();
muon.M();
```

e fare operazioni tra 2 quadri-vettori:

```
TLorentzVector muon1, muon2, particle;

muon1.SetPtEtaPhiE(100, 1.5, 0.5, 120);
muon2.SetPtEtaPhiE(180, -2.1, 3.1, 200);

particle = muon1 + muon2; //equivale alla somma di quadri-vettori

particle.Pt();
particle.M();

ecc...
```

proviamo ora ad usare i TLorentzVector sul nostro tree. Consideriamo tutte le coppie possibili di muoni, calcoliamone la massa invariante e riempiamo un istogramma. Aggiungiamo le seguenti linee prima del loop su tutti gli eventi:

```
TLorentzVector muon1, muon2, pair_mumu;

TH1F* h_mmumu = new TH1F ("h_mmumu", "h_mmumu", 400, 0, 200);
```

all'interno del loop sugli eventi:

```
for(int i=0; i<nmuons; i++){
  for(int j=i+1; j<nmuons; j++){

     muon1.SetPtEtaPhiE( muon_pt->at(i), muon_eta->at(i), muon_phi->at(i), muon_E->at(i) );
     muon2.SetPtEtaPhiE( muon_pt->at(j), muon_eta->at(j), muon_phi->at(j), muon_E->at(j) );

     pair_mumu = muon1 + muon2;

     if( muon_charge->at(i)!=muon_charge->at(j) ) h_mmumu -> Fill( pair_mumu.M() );

  }

}
```

e all'esterno del loop:

```
h_mmumu -> Write();
```

Cosa notiamo nell'istogramma? Ce lo aspettavamo?

Su questo spettro si può giocare. Si può provare a vedere come cambia se si applicano delle selezioni cinematiche sulle coppie selezionate.

Nei dati che stiamo osservando c'è quindi sicuramente la produzione di almeno una risonanza, consideriamo la Z. Nei processi di fisica che conosciamo la maggior parte dei processi prevede la produzione di una singola Z per processo, cerchiamo allora, fissato un evento, di selezionare una sola Z. Come possiamo fare? Quali criteri ci vengono in mente?

1. Selezioniamo eventi con esattamente 2 muoni (o 2 elettroni);

2. Selezioniamo tra tutte le coppie possibili la coppia con la massa invariante più vicino al valore della massa della Z;

3. Selezioniamo tra tutte le coppie possibili la coppia con i pT più elevati.

```
my code...
```

Fitting a histogram

Seguendo la parte precedente del tutorial, dovremmo aver prodotto un file .root nel quale abbiamo salvato vari istogrammi. Creiamo un codice (macro) rapido che ci consenta di aprire il file .root, leggere l'istogramma ed eseguire delle operazioni su di esso, come ad esempio un fit.

Creiamo un file con un emacs:

```
$ emacs FitMacro.C &
```

e scriviamo le linee:

```
using namespace std;

void FitMacro (){

TString name_file_input = "file_histos.root";
TFile *file_histo = new TFile(name_file_input);
TString name_histo = "h_mumu_highpT";

TH1F *h = (TH1F*)file_histo -> Get(name_histo);
TF1 *gaussian = new TF1("gaussian", "gaus", 80, 100);

h -> Fit("gaussian", "R");
}
```

come vediamo dopo aver eseguito questa macro questa ci produce una canvas con il nostro istogramma e il fit che abbiamo deciso di esguire.

Possiamo ora provare utilizzando funzioni diverse, gaussiana, lorentziana, Breit-Wigner, ..., e vedere cosa cambia nel fit.

1. I valori stimati della massa e della larghezza della risonanza scelta cambia se consideriamo coppie di elettroni oppure coppie di muoni?

2. I valori delle larghezze ottenute sono compatibili con i valori che trovate sul PDG?

Breit-Wigner di TMath (non elativistica) e Breit-Wigner relativistica:

```
TF1 *BW = new TF1("BW", "[2]*TMath::BreitWigner(x,[0],[1])", 60, 120);
BW -> SetParameters (1, 90, 0.1);

TF1 *myBW = new TF1("myBW", "(1*[0]/((x*x-[1]*[1])*(x*x-[1]*[1]) + [1]*[1]*[2]*[2]))", 60, 120);
myBW -> SetParameters (1, 90, 1);<br /><br />
```

This topic: Sandbox > ATLASTutorialNaples
Topic revision: r14 - 2020-10-14 - FrancescoCirotto