Introducción a análisis de datos con ROOT y python en Física de Altas Energías

Xuan González

Verano 2018

Contenidos

1. Primeros pasos

En estos apuntes se realiza una introducción al uso de python como lenguaje de programación orientada a objetos (muy brevemente), las librerías ROOT del CERN para análisis en física de partículas y su implementación en python a partir de pyROOT.

Como requisitos previos, es necesario tener conocimientos básicos de programación con python. Es recomendable tener conocimientos básicos de física de partículas (haber cursado la asignatura de cuarto curso Física Nuclear y de Partículas Elementales) aunque no es imprescindible para empezar a familiarizarse con las herramientas de programación utilizadas en Física de Altas Energías.

1.1. El software necesario

Todas las herramientas necesarias están ya instaladas en los ordenadores del grupo, por lo que sólo hace falta tener un cliente de ssh (es recomendable una buena conexión a Internet). Si lo prefieres, puedes instalar el software en tu ordenador y trabajar en local, pero en este caso es muy recomendable usar GNU/Linux.

1.1.1. Cómo conectarse a través de ssh

En GNU/Linux (o Mac) (si usas bash como shell) puedes acceder con tu nombre de usuario y contraseña escribiendo en la terminal:

```
$ ssh usuario@fanaeui.geol.uniovi.es -X
```

Para no tener que memorizar todo el comando puedes crear un alias y guardarlo en tu fichero `.bashrc`. Por ejemplo, el siguiene comando crea el alias pero solo es válido en la shell en la que se ejecuta:

```
$ alias fanaeui='ssh usuario@fanaeui.geol.uniovi.es -X'
```

Mientras que el siguiente comando guarda permanentemente el alias en el `.bashrc` lo que permite que esté disponible en futuras shells.

```
$ echo "alias fanaeui='ssh usuario@fanaeui.geol.uniovi.es -X'" >> ~/.bashrc
```

También se puede añadir manualmente usando un editor de textos (`nano, pico, vim, gedit, kate, geany`). Una vez abierto el editor añade el alias (`alias fanaeui='ssh usuario@fanaeui.geol.uniovi.es -X'`) al final y guarda el fichero

```
$ gedit ~/.bashrc &
```

A partir de este momento podrás usar el comando `fanaeui` para acceder a nuestras máquinas.

En Windows existen varias alternativas. Una de las más usadas es PuTTY.

1.1.2. Software para trabajar en local

Si lo prefieres, puedes trabajar localmente, pero es necesario tener el software instalado:

- Es MUY recomendable utilizar GNU/Linux si trabajas en local.
- python con las librerías básicas (al menos `numpy`).
- El programa ROOT del CERN.

1.2. Comándos básicos de bash

A continuación se muestra una lista de comandos básicos para moverse por la terminal o hacer scripts sencillos ejecutables en bash:

```
echo "hola" # imprimir por pantalla
cd [dir] # moverse a un directorio
ls -lrt # lista archivos ordenados por fecha inversa
mkdir [dir] # Crea un directorio
rm (-rf) [path] # Borra archivos
pushd [dir] # se mueve a un directorio, pero guarda la ruta incial
```

```
wget [website] # Descargar algo de la web
grep [palabra] [fichero] # Busca ocurrencias de [palabra] en [fichero]
```

En linux, los archivos de bash tienen la terminación .sh y pueden contener un conjunto de instrucciones que podrían ser ejecutadas en la terminal una a una. Un ejemplo de código ejecutable que puede ser guardado en un programa.sh:

```
#!/bin/bash
for i in {1..10}
do
        echo $i
done
```

Se puede ejecutar con el comando:

```
$ source programa.sh
```

O bien, dándole privilegios de ejecución:

```
$ chmod a+x programa.sh # Cambiar los permisos
$ ./programa.sh # Ejecutar
```

2. Python y la programación orientada a objetos

La programación orientada objetos se basa en la definición y utilización de clases que, a partir de unos datos de entrada, proporcionan una serie de outputs y dispone de una variedad de métodos que, de forma despreocupada, el usuario puede utilizar (algoritmos complejos, cálculos de todo tipo, interacción con otras clases, etcétera). De manera general, se pueden definir clases que contengran la información necesaria para producir una serie de outputs y llamar a diversos algoritmos y después utilizar objetos de esa clase de manera que toda la maquinaria computacional que no es relevante (si sólo nos interesa el resultado) quede 'encapsulada'.

En python, los objetos de una clase se crean con un constructor (que puede requerir cero o varios argumentos). Los atributos (variables o funciones de la clase) se llaman con la estructura 'objeto.atributo' o 'objeto.funcion(inputs)'.

Como ejemplo, podemos ver la clase de ROOT para hacer operaciones con cuadrimomentos o vectores de Lorentz: TLorentzVector.

```
import ROOT
a = 2       # Esto es un numero entero (int)
b = 0.3     # Esto es un decimal (float)
c = 'hola' # Esto es una cadena de caracteres (string)
d = ['a', 2, 9.2] # Esto es una lista
e = ROOT.TLorentzVector() # Esto es un objeto TLorentzVector
```

Esta clase contiene todos los algoritmos necesarios para obtener cualquier variable cinemática a partir de la información mínima que necesitamos. Por ejemplo, toda la información cinemática de una partícula está dada con sus tres componentes del momento y su energía. Hay un método de la clase TLorentzVector para darle esta información a un objeto de la clase, y luego podemos usar métodos para obtener cualquier información cinemática de la partícula, sin importarnos cómo el programa lo calcula o almacena los valores. Ejemplo:

```
import ROOT
p = ROOT.TLorentzVector() # Esto es un objeto TLorentzVector
p.SetPxPyPzE(px, py, pz, energy) # Introducimos los valores
mass = p.M() # La masa de la particula
transverse_momentum = p.Pt() # Momento transverso
phi_angle = p.Phi() # El angulo polar
pseudorapidity = p.Eta() # La pseudorapidez
```

Otro ejemplo de clase podría ser la clase matriz, la cual almacena una serie de NxM valores y contiene diferentes métodos para obtener la matriz inversa, el determinante, suma de matrices, producto de matrices etcétera.

2.1. Ejemplo de una clase en python

En python, una clase se define con `class` y todos sus atributos vienen referenciados, dentro de la clase, con el prefijo `self.`. Veamos un ejemplo de clase en python: la clase `Aeroplane`. Esta clase tendrá funciones para calcular el peso de un avión, teniendo en cuenta su peso vacío, el del combustible y el de los pasajeros y nos dirá si despega o no. Dispondrá de funciones (métodos) que nos permitirán llenar el avión de combustible y de pasajeros, o avisarnos si no caben tantos pasajeros en un avión dado. Podremos crear múltiples aviones y definir sus propiedades (capacidad, peso máximo al que despega, etcétera).

Consejo: escribe esta clase en un script llamado `aeroplane.py`, para después usarla como un módulo.

```
class Aeroplane:
  def SetPassengers(self, nPassengers, meanWeight = 65):
    if nPassengers > self.capacity:
      print('\n\tNo caben tantos pasajeros!!!\n')
      return
    else:
      self.nPassengers = nPassengers
      self.meanWeight  = meanWeight

  def FillWithFuel(self,nLitres):
    self.fuel = nLitres

  def AddPassengers(self, nPassengers, meanWeight = 65):
```

```
    totalPas = self.nPassengers + nPassengers
    if totalPas > self.capacity:
      print('\n\tNo caben tantos pasajeros!!!\n')
      return
    else:
      self.nPassengers = totalPas
      self.meanWeight  = meanWeight

  def GetTotalWeight(self):
    return self.weight + self.nPassengers*self.meanWeight + self.fuel*self.
        fuelDensity

  def DoesItTakeOff(self):
    weight = self.GetTotalWeight()
    if weight < self.maxWeight:
      print('\n\tDespega perfectamente!\n')
    else:
      print('\n\tNo se levanta del suelo...\n')

  def __init__(self, name = '', weight = 50e3, capacity = 200, maxWeight = 1e5):
    self.name = name
    self.weight = weight # OEW
    self.capacity = capacity
    self.maxWeight = maxWeight

    # Initial parameters:
    self.fuel = 0 # In liters, for example: 50e3 liters
    self.nPassengers = 0
    self.meanWeight = 65 # kg
    self.fuelDensity = 0.8 # kg/L for kerosene
```

Prueba a utilizar esta clase. Por ejemplo, en una sesión interactiva de python:

```
>>> from aeroplane import Aeroplane

>>> aero1 = Aeroplane('Mi modelo 1', 50e3, 250, 1e5)
>>> aero2 = Aeroplane('Mi modelo 2', 40e3, 180, 80000)
```

Ya hemos creado dos aeroplanos diferentes, el segundo más pequeño (menor capacidad, peso máximo para que despegue y menos peso). Tenemos 190 pasajeros, vamos a realizar el embarque:

```
>>> aero1.SetPassengers(190)
>>> aero2.SetPassengers(190)

        No caben tantos pasajeros!!!
```

Vemos que no caben en el segundo. Seguiremos con el primero y seremos generosos con el combustible: 50 mil litros de queroseno. Y comprobamos si despega:

```
>>> aero1.FillWithFuel(50000)
>>> aero1.DoesItTakeOff()

        No se levanta del suelo...
```

Es demasiado peso, vamos a probar con 40 mil litros de combustible:

```
>>> aero1.FillWithFuel(40000)
>>> aero1.DoesItTakeOff()

        Despega perfectamente!
```

Puedes probar a extender la clase añadiendo la opción de fijar un inicio y un destino, una cantidad de kilómetros a recorrer y una tasa de consumo de combustible (que dependerá del peso del avión! Por tanto, del propio consumo de combustible y del número de pasajeros). El problema se iría complicando pero, una vez creada la clase, podríamos usarla de manera fácil para saber los requisitos del avión para realizar un viaje completo.

3. Introducción a ROOT

ROOT es un conjunto de librerías, escritas en C++, elaboradas por el CERN. Incluyen diferentes herramientas y está enfocado al análisis de datos en física de partículas, pero tiene un rango de aplicación más amplio, puesto que también se pueden usar para realizar otro tipo de análisis de datos, simulaciones, ajustes, estudios estadísticos, etcétera.

En su sitio web (https://root.cern.ch/) se puede encontrar toda la información acerca de ROOT, varios tutoriales, cómo instalarlo en diferentes sistemas operativos, etcétera.

En los ordenadores del grupo la configuración de la *shell* para usar ROOT se establece a través del alias `root6`. Una vez configurado el entorno, se puede acceder al modo interactivo de ROOT ejecutando en la terminal el comando `root`.

```
[user@fanaeui ~]$ root6 # Configura ROOT
[user@fanaeui ~]$ root # Ejecuta ROOT
   ------------------------------------------------------------------
  | Welcome to ROOT 6.08/04                        http://root.cern.ch |
  |                                (c) 1995-2016, The ROOT Team       |
  | Built for linuxx8664gcc                                           |
  | From tag v6-08-04, 13 January 2017                                |
  | Try '.help', '.demo', '.license', '.credits', '.quit'/'.q'        |
   ------------------------------------------------------------------

root [0]
```

En el modo interactivo, ROOT puede funcionar como una calculadora o puede recibir e interpretar órdenes recibidas línea a línea. Además, puede interpretar código o instrucciones en C++. Por ejemplo: inicia una sesión interactiva de ROOT y crea un elemento de la clase `TLorentzVector` de ROOT (un cuadrivector) para un muon con momento transverso 25 GeV/c, pseudorapidez 2.1, con un ángulo azimutal de 65º (ángulo en el plano XY en el sistema de coordenadas de CMS) y otro con momento transverso de 35 GeV/c, pseudorapidez de -0.2 y ángulo azimutal de -10º. Calcula la masa invariante del sistema dimuonico:

```
TLorentzVector mu1;
TLorentzVector mu2;
float muonMass = 0.105; // 105 MeV
mu1.SetPtEtaPhiM(25, 2.1, 65./180*TMath::Pi(), muonMass);
mu2.SetPtEtaPhiM(35, -0.2, -10./180*TMath::Pi(), muonMass);
float masaInvariante = (mu1 + mu2).M(); // Usamos el metodo M() de la clase
    TLorentzVector
```

A pesar de estar escritas en C++, todas las librerías de ROOT se pueden acceder desde python (pyROOT), y el uso de todas las clases de ROOT se realiza de manera similar en python. Aunque la ayuda de ROOT se muestre en C++, es fácil de comprender y, con pocos conocimientos de la sintaxis de C++, es fácil de transcribir a la sintaxis de python.

Para utilizar ROOT desde python, primero debes cargar ROOT en la terminal desde la que estés trabajando (recuerta, ejecuta *root6*) y a continuación puedes importar ROOT en python (*import ROOT*). Repitamos lo mismo del ejemplo anterior, ahora desde un script de python:

```
from ROOT import TLorentzVector
from math import pi
mu1 = TLorentzVector()
mu2 = TLorentzVector()
muonMass = 0.105
mu1.SetPtEtaPhiM(25, 2.1, 65./180*pi, muonMass)
mu2.SetPtEtaPhiM(35, -0.2, -10./180*pi, muonMass)
masaInvariante = (mu1 + mu2).M()
print('The invariant mass of the dimuon system is: %1.2f GeV') % masaInvariante
```

Los datos que se utilizan en un análisis en Física de Partículas tienen estructura de n-tupla plana y se almacenan en ROOT en objetos llamados *trees* (objetos de la clase `TTree`). Estos *trees* contienen diferentes *branches* (`TBranch`), que se corresponden con diferentes variables de los sucesos. En un `TTree` se almacena información de un conjunto de sucesos. Si estos sucesos se corresponden a medidas realizadas por CMS hablaremos de un *tree de datos*, mientras que si se tratan de sucesos simulados, hablaremos de un *tree de MC* (Monte Carlo) y normalmente diremos que se corresponde a un proceso del Modelo Estándar (por ejemplo: $t\bar{t}$, Drell-Yan, WW, H, etcétera).

Los *trees* u otros objetos de ROOT se guardan en archivos con extensión `'.root'`. Estos archivos pueden contener diferentes fuentes de información, aunque principalmente se usan para almacenar *trees* e histogramas.

En Oviedo, diferentes producciones de *trees* se guardan en el directorio común `/pool/ciencias/`, y aquí utilizaremos algunos *trees* en el directorio `/pool/ciencias/HeppyTreesSummer16/v2/`.

3.1. Explorar un tree

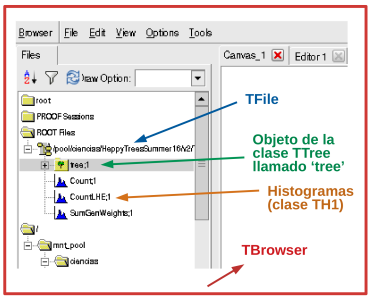
Empecemos por explorar uno de los archivos ROOT que contiene un *tree* de MC. Abramos por ejemplo un *tree* del proceso Drell-Yan:

```
root -l /pool/ciencias/HeppyTreesSummer16/v2/Tree_DYJetsToLL_M50_aMCatNLO_0.root
```

Por defecto, ROOT habrá abierto el archivo creando un objeto de la clase `TFile`, asociado a una variable que habrá llamado `_file0`. ROOT cuenta con un sencillo explorador de archivos: la clase `TBrowser`. Para iniciarlo, basta con crear un objeto de esa clase, que asociaremos a una variable (`browser`, por ejemplo):

```
TBrowser browser
```

En la barra de la izquierda encontraremos fácilmente el archivo que hemos abierto previamente. Podemos observar (doble click) que contiene un `TTree` llamado `tree` y algunos histogramas (de la clase `TH1F`).



Podemos abrir el tree haciendo doble click sobre él. Se nos muestra el conjunto de ramas del *tree*. Con un doble click sobre una de ellas se creará un histograma que se llenará con los valores de la rama dada de todos los sucesos.

Volviendo a la sesión interactiva de ROOT, podemos explorar el *tree* utilizando algunos métodos de la clase `TTree`. Por ejemplo, podemos explorar sucesos individuales del *tree* utilizando `TTree::Scan()`. Ejemplo:

```
tree->Scan("Jet_pt : Jet_eta : LepGood_pt : LepGood_eta", "nJet > 4")
```

El primer argumento indica las variables que queremos explorar (en este caso, momento transverso y pseudorapidez de jets y leptones), mientras que el segundo argumento es una selección sobre los sucesos. En este caso estaremos seleccionando sucesos con al menos 5 jets.

También sin necesidad de abrir un `TBrowser` podemos obtener una lista de las ramas del *tree*, donde también obtenemos el tipo de las variables y una breve descripción. Para ello, hay que utilizar el método `TTree::Print()`, y como primer argumento podemos ponerle una cadena de caracteres. Por ejemplo, si queremos sólo obtener información sobre las ramas relacionado con leptones buenos (los llamados `LepGood` en estos *trees*):

```
tree->Print("LepGood_*")
```

También se puede hacer un histograma de manera muy sencilla a partir de las variables de *tree* utilizando `TTree::Draw()`. Incluso se puede realizar una selección de sucesos. Por ejemplo, miremos la distribución del momento transverso de los leptones en sucesos sin jets:

```
tree->Draw("LepGood_pt", "nJet == 0")
```

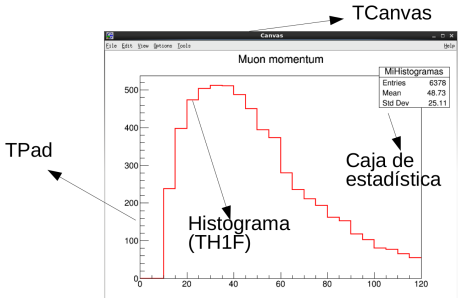
Puedes consultar otros métodos de la clase `TTree` en la ayuda de ROOT.

3.2. Dibujar y guardar un plot

A continuación vamos a:

- Crear un histograma (`TH1F`).
- Llenarlo a partir de un `TTree`, utilizando `TTree::Draw()`.
- Dibujar el histograma en un canvas (`TCanvas`).
- Personalizar el histograma y el canvas.
- Guardar el histograma (objeto `TH1F`) en un archivo ROOT y el canvas en un archivo `.png`.

El lenguaje puede ser confuso. Aclaremos: cuando hablamos de un histograma de ROOT nos referimos a un objeto de la clase `TH1F` (o `TH2F`, `TH1D`, etcétera... en general, las que empiezan por TH). Esto son objetos matemáticos, que podemos modificar las veces que queramos, hacerles ajustes, integrales, media, etcétera. Llamamos *canvas* al marco donde se pueden representar gráficamente los histogramas, texto, funciones, etcétera (clase `TCanvas`). Llamamos *pad* a cada uno de los recuadros en los que dividimos el canvas y donde podemos representar un histograma (clase `TPad`). Llamamos *plot* a cada figura que contiene uno o varios histogramas, texto, etcétera y normalmente es guardado en un archivo `.png` o `.pdf`, no modificable desde ROOT. Algunos de estos conceptos están representados en la Figura 2.



Sigue este ejemplo comentado en una sesión interactiva de python. Comprueba los métodos y constructores en la ayuda de python. Primero, abrimos el archivo ROOT de Drell-Yan con un `TFile` y asignamos una variable al `TTree` llamado `tree`:

```
from ROOT import *
f = TFile.Open('/pool/ciencias/HeppyTreesSummer16/v2/Tree_DYJetsToLL_M50_aMCatNLO_0
    .root')
t = f.Get('tree')
print 'Number of entries: ', t.GetEntries() # Para comprobar que se ha abierto bien
```

Creamos un histograma (TH1F) de una dimensión con valores decimales con 20 bines (celdas) con valores entre 0 y 100. Creamos también un canvas (`TCanvas`). Atención: NO lo cierres hasta haber terminado el programa.

```
h = TH1F("mi_histograma", "Titulo del histograma", 20, 0, 100)
c = TCanvas('c', 'c', 10, 10, 800, 600)
```

Utilizamos `TTree::Draw()` para llenar el histograma con la energía faltante transversa para sucesos con 2 o menos jets.

```
t.Draw("met_pt>>mi_histograma", "nJet<=2")
```

Cambiamos el estilo, los colores y guardamos el canvas como `.png` y el histograma en un nuevo archivo ROOT.

```
h.SetLineColor(kRed)
h.SetLineWidth(2)
h.SetFillColor(0)
h.SetTilte('Bonito histograma')
h.GetXaxis().SetTitle('MET (GeV)')
h.GetYaxis().SetTitle('Events / 5 GeV')
h.Draw('hist')
c.Print('mi_histograma.png', 'png')
out = TFile('./fileConMiHistograma.root','recreate')
h.Write()
```

Sal de la sesión de python. Puedes comprobar que tienes una imagen `mi_histograma.png` y un `mi_histograma.root` que puedes abrir con un `TBrowser` en una sesión interactiva de ROOT para comprobar que el histograma se ha guardado.

3.3. Realizar un ajuste

ROOT también se puede utilizar para realizar ajustes. De hecho, existe una extensión llamada RooFit con potentes herramientas para hacer ajustes y tratamientos estadísticos. A continuación veremos un ejemplo muy sencillo de cómo realizar un ajuste gaussiano a un histograma. En particular, crearemos un histograma que llegaremos con números aleatorios con distribución gaussiana y realizaremos un ajuste gaussiano con el método Fit de TH1. Puedes ejecutar el ejemplo en secuencial o (recomendable a partir de ahora) crear un script de python y ejecutarlo.

```
from ROOT import *
gROOT.SetBatch(1) # Para que no se abran los canvas
h = TH1F('h', 'Gaussian fit', 50, -5, 5)
h.FillRandom('gaus', 1000)
fit = h.Fit('gaus', 'S')
```

Cambiamos el estilo del histograma y el fit (sólo para que quede estéticamente mejor):

```
h.SetFillColor(kAzure+5)
h.GetXaxis().SetTitle("Valor")
h.GetYaxis().SetTitle("Frecuencia")
h.SetStats(0)
fitline = h.GetFunction('gaus')
fitline.SetLineWidth(3)
fitline.SetLineColor(kRed+1)
```

Obtenemos los valores de los parámetros ajustados. En este caso, un ajuste gaussiano, nos interesa la media y la desviación típica:

```
media = fit.Value(1)
sigma = fit.Value(2)
```

Por último, creamos un canvas y guardamos el plot con el histograma y el ajuste:

```
c = TCanvas('c', 'c', 10, 10, 800, 600)
h.Draw("hist")
fitline.Draw("same")
t = TLatex(); t.DrawLatexNDC(0.15, 0.7, "#splitline{#mu = %1.2f}{#sigma = %1.2f}"%(
    media, sigma))
c.Print('GaussianFit.png', 'png')
```

3.4. Bucle sobre los sucesos de un tree

Es muy común realizar el análisis de los datos en física de partículas a través de un bucle que recorra todos los sucesos de una muestra (ya sea datos o simulación) y, en cada suceso, se definan objetos y cantidades físicas, se realiza una selección, se apliquen correcciones, etcétera.

Utilizando pyROOT es muy sencillo hacer un bucle sobre todos los sucesos de un *tree* dado. En este ejemplo vamos a construir la masa invariante del bosón Z en una muestra de Drell-Yan, a través de un bucle donde construiremos esa variable y realizaremos una selección sobre los leptones y jets.

```
from ROOT import *
gROOT.SetBatch(1) # Para que no se abran los canvas

pathToTree = '/pool/ciencias/HeppyTreesSummer16/v2/'
treeName = 'DYJetsToLL_M50_aMCatNLO_0'

# Abrimos el rootfile
f = TFile.Open(pathToTree + 'Tree_' + treeName + '.root')

# Creamos el histograma de masa invariante
h = TH1F('hInvMass', 'Invariant mass of two muons', 50, 65, 105)

# Sucesos en el tree
totalEvents = f.tree.GetEntries()
```

```
iEntry = -1
maxEvents = 100000
```

Empezamos en bucle. Acabará cuando hayamos analizado 100000 sucesos (`maxEvents`). Haremos una selección de sucesos con dos muones y al menos un jet con momento transverso $p_T \geq 100$ GeV. Posteriormente calcularemos la masa invariante del par de muones haciendo uso de la clase `TLorentzVector` y llenaremos el histograma con este valor.

```
for event in f.tree:
  iEntry += 1
  if   iEntry == maxEvents: break # Analiza un numero de sucesos

  elif iEntry % 5000 == 0: print '>> progress: %/%i'%(iEntry, maxEvents) # Para
      saber cuanto queda

  # Seleccionamos dos muones
  if event.nLepGood < 2: continue # Al menos dos leptons
  lep1id = event.LepGood_pdgId[0]
  lep2id = event.LepGood_pdgId[1]
  if not (abs(lep1id) == 13 and abs(lep2id) == 13): continue # Munoes: pdgId = 13,
      electrones: pdgId = 11

  ## Al menos un jet de pT > 100 GeV
  nJetPt100 = 0
  for i in range(event.nJet):
    if event.Jet_pt[i] >= 100: nJetPt100 += 1
  if nJetPt100 == 0: continue

  # Creamos los TLorentzVector de cada muon y obtenemos la masa invariante
  muo1 = TLorentzVector()
  muo2 = TLorentzVector()
  muo1.SetPtEtaPhiE(event.LepGood_pt[0], event.LepGood_eta[0], event.LepGood_phi
      [0], event.LepGood_energy[0])
  muo2.SetPtEtaPhiE(event.LepGood_pt[1], event.LepGood_eta[1], event.LepGood_phi
      [1], event.LepGood_energy[1])
  invariantMass = (muo1 + muo2).M()

  # Anyadimos al histograma el valor de masa invariante de este suceso
  h.Fill(invariantMass)
```

En este punto, una vez acabe el bucle, ya tendremos el histograma con la masa invariante de dos muones en sucesos de Drell-Yan con al menos un jet de al menos 100 GeV de momento transverso. Podemos ahora hacer un ajuste gaussiano al pico de resonancia y ver si se ajusta adecuadamente y obtener el valor de la masa del bosón Z a partir del ajuste.

```
# Estilo
h.GetXaxis().SetTitle("m_{#mu#mu} (GeV)")
h.GetYaxis().SetTitle("Events")
h.SetLineColor(kViolet+3)
h.SetFillColor(kGreen+2)
h.SetLineWidth(3)
h.SetStats(0)

# Hacemos el ajuste
fit = h.Fit('gaus', 'S')
Zmass = fit.Value(1)
sigma = fit.Value(2)
fitline = h.GetFunction('gaus')
fitline.SetLineWidth(3)
fitline.SetLineColor(kRed+1)

# Creamos el canvas y dibujamos
c = TCanvas('c', 'c', 10, 10, 800, 600)
h.Draw('hist')
fitline.Draw("same")
t = TLatex(); t.DrawLatexNDC(0.15, 0.7, "#splitline{M_{Z} = %1.2f GeV}{#sigma =
    %1.2f GeV}"%(Zmass, sigma))
c.Print('ZMass.png', 'png')
```

4. Para profundizar y ejercitarse

Crea histogramas normalizados a 1 pb^{-1}

El número de sucesos para un proceso dado (por ejemplo, procesos de producción de pares top-antitop, $t\bar{t}$) que esperamos producir en colisiones de protones será:

$$N_{ext} = \sigma \cdot \int \mathcal{L} dt \tag{1}$$

donde $\int \mathcal{L}$dt es la luminosidad integrada, que depende del tiempo de toma de datos y de parámetros del acelerador (se mide en inversos de sección eficaz), y σ es la sección eficaz del proceso. La luminosidad instantánea depende de la forma del haz de protones. El perfil geométrico de los paquetes de protones del LHC en los puntos de colisión es aproximadamente gaussiano y la luminosidad es de aproximadamente $2 \cdot 10 \cdot 10^{34}$ cm$^{-2} \cdot$ s^{-1} y se puede calcular como:

$$\mathcal{L} = f \cdot \frac{N_1 N_2}{4\pi\sigma_x\sigma_y} \tag{2}$$

donde N_1 y N_2 es el número de protones por paquete, f es la frecuencia de colisiones (que depende del número de vueltas por segundo, o longitud del acelerador suponiendo una velocidad de c, y del número de paquetes que caben en cada vuelta, de unos 2800, o la distancia temporal entre ellos, unos 25 ns), y σ_x, σ_y son las correspondientes a los perfiles gaussianos de la distrubución de protones en las direcciones X e Y dentro de los paquetes.

Sin embargo, para evitar fluctuaciones estadísticas y conseguir un mejor modelado de un proceso del Modelo Estándar dado, normalmente se generan más sucesos (si se puede, un orden de magnitud más) de los sucesos esperados. A veces se generan para un estado final dado (por ejemplo: dos leptonces), por lo que hay que multiplicar la cantidad en (1) por la fracción de desintegración (BR, branching ratio).

Por ejemplo, con una cantidad de 2 fb^{-1} (= 2000 pb^{-1}) de datos recogidos a 13 TeV en CMS, se espera que se produzca una cantidad de sucesos $t\bar{t}$ con desintegración dileptónica de (la sección efiaz es 831.8 pb y la fracción de desintegración a dos leptones es aproximadamente el cuadrado de 0.108 (BR W$\rightarrow$lepton) $\cdot$ 3 (e,μ,τ)):

$$N = BR \cdot \sigma_{t\bar{t}}^{13TeV} \cdot \int \mathcal{L} dt = (0{,}108 \cdot 3)^2 \cdot 831{,}8 \cdot 2000 = 174638{,}07$$

Sin embargo, la estimación será mucho más precisa si se producen 1 millón de sucesos. Para poder comparar las distribuciones con lo obtenido en los datos y con lo esperado para otros sucesos, se debe normalizar el número de sucesos a lo esperado. Esto se hace normalmente aplicando un peso a cada suceso, que será de:

$$\omega = \frac{\sigma}{\text{Número de sucesos generados}} \cdot L \tag{3}$$

donde $L = \int \mathcal{L}$dt.

Guarda histogramas aplicando un peso normalizando a 1 pb^{-1}. El número de sucesos generados se lee de un histograma llamado `hcount` que se guarda junto con los *trees*. La sección eficaz la puedes leer, para cada proceso del Modelo Estándar, en la siguiente tabla, en la pestagna 'DR80XSummer16asymptoticMiniAODv2_2':

https://docs.google.com/spreadsheets/d/1b4qnWfZrimEGYc1z4dHl21-A9qyJgpqNUbhOlvCzjbE

El peso en un histograma se introduce junto en la función `Fill()`. Para comparar con datos, se tiene que normalizar a la luminosidad utilizada, lo cual se puede hacer con la función `TH1F::Scale()`. Ejemplo:

```
peso = xsec/NEvents
for i in range(events):
  h.Fill(variable, peso)
h.Scale(lumi)
```

Crea una función para cargar todos los ficheros ROOT de un mismo proceso

Cuando los archivos ROOT que contienen los *trees* son muy pesados, éstos se dividen en varios. Normalmente, la estructura del nombre de estos archivos es: `Tree_Proceso_i.root`, donde `i` es un número. Para poder realizar una normalización adecuada, hay que utilizar todos los *trees* de un mismo proceso.

Para hacer un loop sobre todos los sucesos de un mismo proceso se puede utilizar la clase TChain, que no es más que una clase que sirve para encadenar objetos TTree. Por ejemplo, para cargar un número determinado de trees con nombre 'tree':

```
path = '/pool/ciencias/HeppyTreesSummer16/v2/'
name = 'ZZTo4L'
t = TChain('miTChain', 'tree')
t.Add(path + 'Tree_' + name + '_*.root' # El * se substituye por cualquier cadena
    de caracteres
```

Crea un selector que guarde histogramas normalizados en un .root

Programa un selector en forma de clase, con métodos para indicar el path donde están los rootfiles, el nombre de los trees, etcétera. Este selector debe realizar un loop sobre todos los sucesos, realizar una selección, llenar histogramas y guardarlos en un archivo de salida.

Para guardar histogramas en un fichero utiliza:

```
processName = 'ZZTo4L' # Esto es un input del programa

# Despues del el loop y haber llenado los histogramas
outpath = './histogramas/'
fout = TFile(outpath + processName + '.root')
h1.Write()
h2.Write()
# etc
fout.Close()
```

Crea un plotter

Primero: crea una función o una clase que lea y cargue histogramas a partir de un rootfile. Ejemplo:

```
def loadHisto(path, fileName, histoName):
  f = TFile.Open(path + '/' + fileName + '.root')
  h = f.Get(histoName)
  f.Close()
  return h
```

Después: crea funciones (o una clase) que utilice histogramas, les ponga un estilo determinado y los represente a tu gusto. Ejemplo:

```
def drawHisto(h, name, xtit = '', ytit = ''):
  c = TCanvas('c', 'c', 10, 10, 800, 600)
  h.SetLineColor(kRed)
  h.SetLineWidth(2)
  h.SetLineStyle(1)
  h.SetFillColor(0)
  h.SetStats(0)
  l = TLatex() # para poner titulos
  leg = TLegend() # hacer una leyenda
  h.GetXaxis().SetTitle(xtit)
  h.GetYaxis().SetTitle(ytit)
  h.Draw('hist')
  c.Print(name + '.png', '.png')
```

Produce diferentes tipos de plots

Haz un plotter (funciones o clases, etc) donde aproveches las funciones para cargar histogramas o funciones para darle estilo, etc, y dibujes:

- Stack plot: usando THStack, con diferentes colores, etc.
- Plots de comparación: puedes dibujar varios histogramas en el mismo TPad (usa Draw("same") para ello). Puedes dibujarlos normalizados a 1 (usando h.Scale(h.Integral())).

- Ratio plots: es decir, comparar la división de dos distribuciones (usa h1.Divide(h2)).
- Dibuja errores sistemáticos: usa h.SetBinError(bin, value) para asignar un error a un determinado bin. Puedes tomar el valor por defecto (estadístico) que se asigna automáticamente cuando llenas el histograma con Fill. Para pintar el histograma como una banda de error, haz: h.Draw("same, e2"), tras haber dibujado el histograma. Cambia el estilo con h.SetFillColor, h.SetFillStyle.

Nota: si quieres clonar histogramas (por ejemplo, para usarlos con diferentes propósitos) puedes usar: hclone = h.Clone("nuevo_nombre").

Compara con datos

Añade a tu análisis las muestras de datos (SingleMuon, SingleElec, DoubleMuon, DoubleElec, MuonEG). Una vez hayas ejecutado el selector para estos datos, puedes buscar la luminosidad integrada a la que corresponden y compararlo con los diferentes procesos del Modelo Estándar. Recuerda: el peso para los sucesos de datos tiene que ser siempre 1.

Empieza siempre seleccionando sucesos con dos leptones, construyendo la masa invariantes y comparando con la teoría en el pico del Z. Cuando compares con los datos, debería ser bueno cerca de 91 GeV con sólo usar la muestra de DY, pero irá empeorando a medida que te alejas, a no ser que incluyas todos los procesos del Modelo Estándar adecuados ($t\bar{t}$, WW, ZZ, tW...).

Para obtener un acuerdo bueno, puedes elegir una de estas opciones:

- Selecciona sucesos con dos leptones, ee o $\mu\mu$ (excluye sucesos eμ), con una masa invariante mInv cercana al Z: $|mInv - m_Z| < 10$ GeV. Compara datos con la muestra de DY.
- Selecciona sucesos con dos leptones en cualquier rango de masas invariantes y compara datos con la suma de: DY, WJets, $t\bar{t}$, tW, WW, WZ, ZZ. Utiliza THStack y asigna diferentes colores a los diferentes procesos. Prueba a seleccionar sucesos con además al menos dos jets (de p_T >30 GeV) y que contienen una pareja eμ (selección pura $t\bar{t}$).

Ejemplo de identificación de sabor de los leptones (a implementar en el selector, dentro del el bucle que recorre los sucesos del tree):

```
if(nLepGood >= 2):
  id1 = abs(LepGood_pdgId[0])
  id2 = abs(LepGood_pdgId[1])
  chanId = id1 + id2
  if    id1 + id2 == 22: channel = 'ElEl'
  elif id1 + id2 == 24: channel = 'ElMu'
  elif id1 + is2 == 26: channel = 'MuMu'

  # Por ejemplo, si solo queremos sucesos e+mu:
  if chanId != 24: continue # go no next event
```

Ejemplo de selección de número de jets (a implementar en el selector, dentro del el bucle que recorre los sucesos del tree):

```
nJet = 0
for i in range(nJet):
  jetPt = Jet_pt[i]
  if jetPt > 30: nJet += 1 # Jet with pT > 30 GeV !!

if nJet < 2: continue # go to next event...
```

Crea un modulo con funciones básicas

Crea un 'functions.py' con varias funciones que consideres útiles, para calcular diferentes variables cinemáticas. Por ejemplo, a partir de px, py, pz, E de dos leptones, construyes los TLorentzVector y escribes diferentes funciones para calcular: la masa invariante, el ángulo entre dos leptones, el momento dileptónico, la masa transversa, la diferencia de pseudoradpidez... Puedes incluir funciones que calculen el p_T de la suma de los jets en un vector (o una lista), el mayor de los ángulos entre un leptón y una lista de jets, etcétera. Estas funciones son muy útiles a la hora de entender varios procesos en CMS.

Estudia variables de identificación de leptones

Explora diferentes variables de identificación y aislamiento de leptones (LepGood_*). LepGood incluye electrones y muones. Para un leptón dado, la variable LepGood_pdgId vale ± 11 si es un electrón y ± 13 si es un muon. Haz una selección de electrones y/o muones y dibuja variables como: RelIso, dxy, dz, pixelLayers, miniRelIso, nStations (sólo muon), convVeto (sólo elec), eInvMinusPInt (sólo elec), hadronicOverEm (sólo elec), sigmaIetaIeta (sólo elec).

Puedes explorar también variables similares de los Jets (colección Jet_*).

Compara el acuerdo datos vs predicción para una selección dileptónica en el pico del Z o fuera si incluyes todos los procesos.

NOTA: puedes obtener una breve descripción de las variables abriendo un rootfile con ROOT y ejecutando la línea tree->Print("LepGood_*").

Estudia variables cinemáticas básicas para sucesos con dos leptones

Realiza una selección de dos leptones y pinta las variables cinemáticas básicas de leptones y jets: p_T de los leptones y jets, pseudorapidez ($|\eta|$) de leptones y jets, ángulos entre leptones y jets, masa invariante, MET, HT = $\sum \text{jet}_{p_T}$, número de jets, masa transversa entre cada lepton y MET, etcétera.

Compara el acuerdo datos vs predicción para una selección dileptónica en el pico del Z o fuera si incluyes todos los procesos.

NOTA: puedes añadir una selección en número de jets (prueba a seleccionar jets con $p_T > 30$ GeV y $|\eta| < 2.4$. NOTA: puedes dividir los sucesos en tres categorías (canales) dependiendo del sabor de los leptones (ee, $\mu\mu$, eμ).

Información de interés

Consejos de programación

- Guarda los programas cada poco tiempo. Haz pruebas, ejecuta cada pocas líneas.
- Empieza a diseñar y probar los programas con un caso sencillo (en el caso del análisis: prueba primero con un solo histograma, sólo con Drell-Yan, etc).
- Crea programas diferentes en .py diferentes y usa import para usar las funciones y clases que hayas definido en otros programas.
- Pon comentarios. No sólo son útiles para que otras personas puedan entender el programa, sino también para uno mismo. Recomendación: Unas pocas líneas al principio de cada programa explicando qué hace, las funciones más importantes y cómo se ejecuta; una breve descripción de cada función al comienzo de su definición; comentarios en línea explicando lo que hace cada parte del programa o cada instrucción que no sea evidente.
- Haz 'print' de vez en cuando para poder seguir el estado de la ejecución. Por ejemplo, en los loops sobre sucesos de un tree:

```
for k in range(nEvents):
  if k %10000 == 0: print '>>Progress: %i of %i'%(k, nEvents)
```

- Pon mensajes de error cuando una condición indispensable para que el programa funcione no se cumpla. Por ejemplo:

```
import os
def loadHisto(path, fileName):
  if not os.path.isdir(path):
    print 'ERROR: path ' + path + ' does not exist!!!'
  elif not os.path.isfile(path + fileName):
    print 'ERROR: file ' + path + fileName + 'does not exist!!!'
```

- Procura que tu código sea versátil.
- Simplifica el trabajo 'externo' y anticípate a los posibles errores. Ejemplo, si queremos abrir un fichero .root en una ruta dada, podrías incluir en tu función:

```
def loadHisto(path, fileName):
  if not path[-1] == '/': path += '/'
  if not fileName[-5:] == '.root': fileName += '.root'
  # etc
```

Software útil

- Edición de texto: vim, emacs (editores que se pueden abrir en terminal). LaTeX, overleaf (para editar LaTeXen línea y compartir). Conviene tener una cuenta de Google, ya que con mucha frecuencia se utilizan docs, slides y spreadsheets para editar online y en grupo.
- Lenguajes de programación: C++, python, bash. Librerías de ROOT.
- Compartir código: GitHub. Si aprendéis a usarlo, lo podréis utilizar para el TFG. También os puede servir para vuestros proyectos personales, para escribir vueltro CV en latex, etc.
- Comunicación: Skype. Es el estándar para comunicarse en ciencia con gente de todo el mundo.

Sitios web de interés

- Web del PDG. Con reviews de todos los temas de actualidad en física de partículas. Un punto de partida para empezar a aprender sobre cualquier tema y una fuente de enlaces a otras publicaciones.
- Página oficial de publicaciones de CMS.
- Web del CERN.

- Web de ROOT.
- En arXiv y InspireHEP encontraréis el 100 % de las publicaciones de experimentos del CERN y otros.
- Hoja de google con las muestras de datos y MC en Oviedo e información relativa.
- Repositorio de GitHub del grupo de Oviedo para el framework de análisis PAF, y página oficial de PAF.
- Página web del grupo.
- Información útil en la wiki del grupo.