Tutorial para iniciar un analisis con CMSSW

En este tutorial vamos a dar los pasos iniciales para hacer un analisis de datos (reales / simulacion) con CMSSW. Esto complementa las instrucciones dadas en el Twiki de generación de datos de MC.

Preliminares

- Cuenta en el servidor de Altas Energias
- Configuracion para correr CMSSW
- Area en scratch lista para trabajar con la version **CMSSW_3_11_3**
- Datos de entrada para el analisis: **/scratch/aosorio/data/HSCPs/tutorial/HSCPsEvents.root** (10000 eventos de HSCPs - staus(308)).

Creacion de un EDAnalyzer

Entrar en el area de trabajo y alistar el entorno para la version de CMSSW. Adicionalmente crear en el directorio src/ un directorio en el que trabajeremos el analisis de datos.

```
cd /scratch/aosorio
cd Analysis/CMSSW_3_11_3/src
cmsenv
mkdir MyAnalyzers
```

Dentro de CMSSW hay una serie de scripts que crean esqueletos de los tres tipos basicos de programas que uno puede hacer con este software: **Producers**, **Filters**, **Analyzers**. Los scripts son:

- mkedanlzr: creador de analyzers
- mkedfltr: creador de filters
- mkedprod: creador de producers

Creemos entonces dentro del directorio "MyAnlyzers" un analyzer:

```
cd MyAnalyzers
mkedanlzr PATOne -histo
```

La opcion "-histo" instruye al script que adicione "lo necesario para crear histogramas" en nuestro analyzer.

Creacion de PATtuplas

Para usar el Physics Analysis Toolkit o PAT sobre nuestros datos, necesitamos hacer un paso adicional que consiste en crear unos nuevos archivos que contengan las colecciones propias de PAT basados en datos RECO. Por suerte ya existen una serie de scripts de configuracion estandar que nos sirven para crear dichos archivos o PATtuplas (ver).

- El paso que sigue es descargar el siguiente script:

```
[hep] wget --no-check-certificate https://twiki.cern.ch/twiki/pub/Main/CMSUniandesGroupAnalysis/p
```

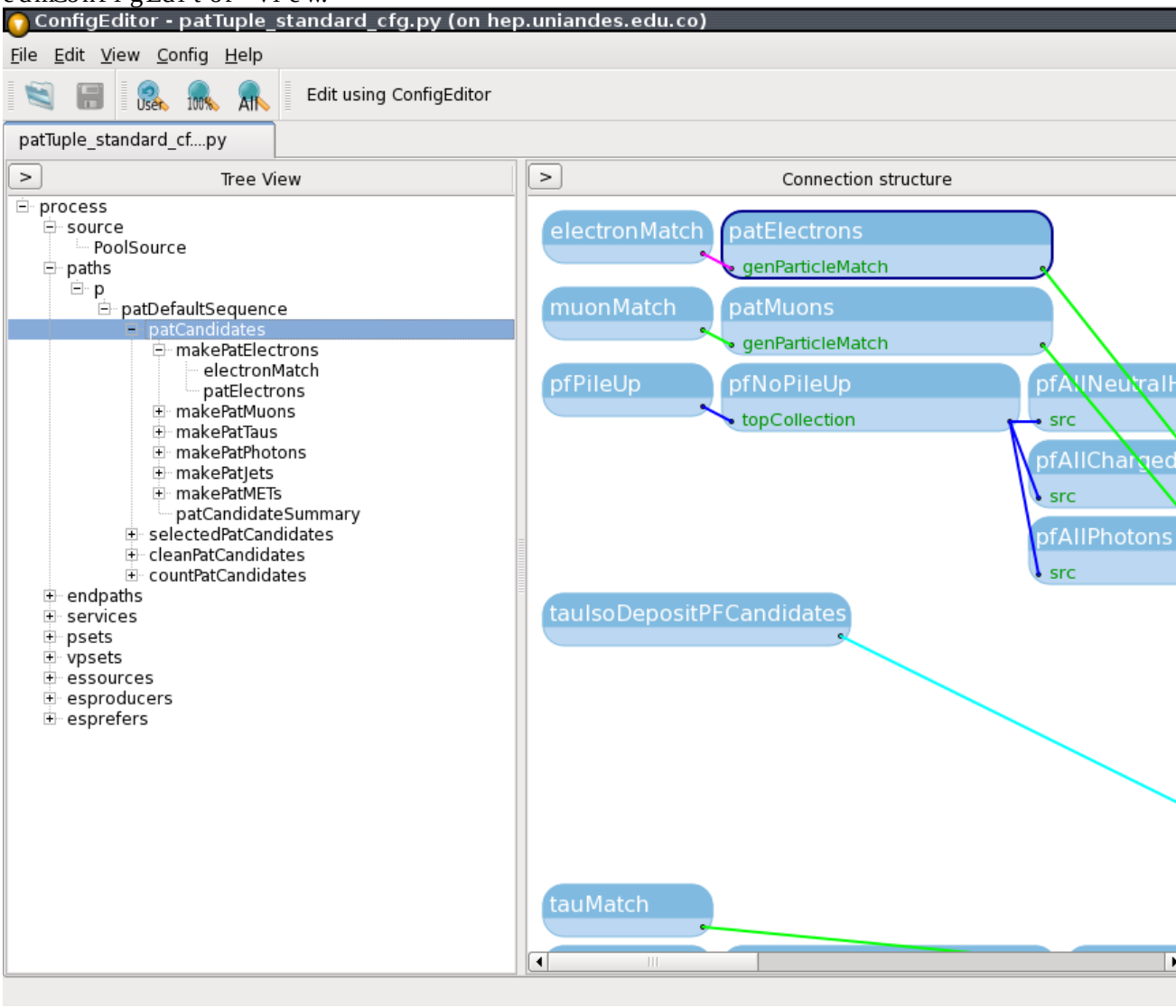
- Debido a la forma como el twiki guarda el archivo, debemos cambiarle el nombre:

```
[hep] mv patTuple_standard_cfg.py.txt patTuple_standard_cfg.py
```

- Demos un vistazo a este configuration file usando el comando **edmConfigEditor**. En realidad es mas que un vistazo, pues nos permite visualizar como se configura **cmsRun** con el archivo para creacion de PATtuplas estandar:

```
[hep] edmConfigEditor patTuple_standard_cfg.py
```

- edmConfigEditor view:



- Corramos **cmsRun** sobre el archivo de configuracion

```
[hep] cmsRun patTuple_standard_cfg.py
```

- salida-cmsRun-patTuple.txt: Salida de cmsRun - patTuple

Por comodidad, he corrido sobre los 10000 eventos y colocado el resultado en el siguiente directorio:

- **/scratch/aosorio/data/HSCPs/tutorial/patTuple_HSCPs.root**

PAT analysis

Ahora que tenemos la base para hacer nuestro analisis, podemos empezar a editar el Analyzer que hemos creado. Lo primero que yo sugiero siempre, es separar declaración de definición:

```
cd PATOne/src
cp PATOne.cc PATOne.h
```

Editar PATOne.h y PATOne.cc para tal efecto. En un lado debe quedar solo la declaracion de la clase dentro de **PATOne.h** y en el otro las definiciones de sus metodos (constructor, destructor y otros) dentro de **PATOne.cc**. Estudiar el esqueleto.

Dado que vamos a utilizar las colecciones que contienen objetos de tipo **PAT**, necesitamos adicionar algunos archivos de declaración de estas clases para que nuestro Analyzer sepa manejarlos. Editar "PATOne.h" y adicionar las siguientes lineas:

```
#include "DataFormats/PatCandidates/interface/Muon.h"
#include "DataFormats/PatCandidates/interface/Jet.h"
#include "DataFormats/PatCandidates/interface/MET.h"
```

Como vemos, estamos incluyendo las declaraciones de tres clases de objetos que son de tipo **PatCandidates** que son muones, jets y MET. Estos includes nos permitiran realizar nuestro analisis basados pat::Jet, pat::Muon y pat::MET.

- Tambien necesitaremos adicionar una serie de **InputTag** que nos ayuden a acceder los datos que estan contenidos en el archivo **.root** de entrada. Al igual que cualquier otro miembro, necesitamos declarar las variables dentro **PATOne.h** y luego en el constructor darles una definicion. Los **InputTag** serán pasados mediante el archivo de configuracion de nuestro proceso:

- PATOne.h

```
  // input tags
  edm::InputTag m_muonSrc;
  edm::InputTag m_jetSrc;
  edm::InputTag m_metSrc;
```

- PATOne.cc

```
//Adicionar esto dentro del construcctor del analyzer:
m_muonSrc = iConfig.getUntrackedParameter<edm::InputTag>("muonSrc");
m_jetSrc  = iConfig.getUntrackedParameter<edm::InputTag>("jetSrc" );
m_metSrc  = iConfig.getUntrackedParameter<edm::InputTag>("metSrc" );
```

Aqui estamos definiendo las variables **m_muonSrc**, **m_jetSrc** y **m_metSrc** como unos **edm::InputTag** provenientes de la Parametros definidos es en un archivo de configuracion. Estas tres lineas significan que en debemos dar una definicion de estos tres parametros. Editemos entonces el archivo de

configuracion que viene en preparado cuando creamos el esqueleto:

```
import FWCore.ParameterSet.Config as cms

process = cms.Process("Demo")

process.load("FWCore.MessageService.MessageLogger_cfi")

process.maxEvents = cms.untracked.PSet( input = cms.untracked.int32(-1) )

process.source = cms.Source("PoolSource",
    # replace 'myfile.root' with the source file you want to use
    fileNames = cms.untracked.vstring(
        'file:myfile.root'
    )
)

process.demo = cms.EDAnalyzer('PatOne'
                              )

process.TFileService = cms.Service("TFileService",
    fileName = cms.string('histo.root')
)

process.p = cms.Path(process.demo)
```

Como vemos **process.demo** es el "camino que ejecuta nuestro **analyzer PATOne**. Debemos insertar los tres parametros que necesitamos para correrlo, su definicion debe entonces lucir de la siguiente forma:

```
process.demo = cms.EDAnalyzer('PatOne',
                                                muonSrc     = cms.untracked.InputTag("cleanP
                                                jetSrc          = cms.untracked.InputTag("cl
                                                metSrc        = cms.untracked.InputTag("patM
```

- Lo siguiente consiste en modificar el metodo **analyze**, que es donde ocurre el analsis. Primero necesitamos acceder a las colecciones que guardan la información de jets, muones y MET por evento. La herramienta usada en CMSSW para ello se llama un **Handle**. En nuestro caso necesitamos tres (una por cada objeto fisico):

```
 ////////////////////////////////////////////////////////////////////////////
  // 1. Jet collection
  edm::Handle<edm::View<pat::Jet> > jets;
  iEvent.getByLabel(m_jetSrc,jets);

  // 2. Muon collection
  edm::Handle<edm::View<pat::Muon> > muons;
  iEvent.getByLabel(m_muonSrc,muons);

  // 3. MET collection
  edm::Handle<edm::View<pat::MET> > mets;
  iEvent.getByLabel(m_metSrc,mets);
  ////////////////////////////////////////////////////////////////////////////
```

Los Handle nos permiten acceder a las colecciones que contienen los datos. De ahora en adelante, estamos listos para dar inicio a nuestro analisis. Por ejemplo podemos crear los siguientes **loops** sobre los muones y jets:

```
 //...Loop sobre Jets
```

```
  int nJets = 0;

  for(edm::View<pat::Jet>::const_iterator jet=jets->begin(); jet!=jets->end(); ++jet){
    if( jet->pt()>50.0 ) //seleccionamos solo jets con Pt > 50
      ++nJets;
  }

  ////Este es un histograma el cual llenaremos en un momento
  ///h_1DHistContainer["Njets"]->Fill(nJets);

  //...Loop sobre Muones

  int nMuons = 0;

  for(edm::View<pat::Muon>::const_iterator muon=muons->begin(); muon!=muons->end(); ++muon){
    ++nMuons;
  }

 ////Este es un histograma el cual llenaremos en un momento
  ////h_1DHistContainer["Nmuons"]->Fill(nMuons);

  //...MET

  if( ! mets->empty() ) {
    ////Este es un histograma el cual llenaremos en un momento
    ///h_1DHistContainer["met"]->Fill( (*mets)[0].et() );
  }
```

Estamos listos para compilar el codigo! Pero antes ... debemos editar un ultimo archivo que viene con nuestro esqueleto "Buildfile". Este debe quedar de la siguiente forma:

```
<use name=FWCore/Framework>
<use name=FWCore/PluginManager>
<use name=FWCore/ParameterSet>
<use name=FWCore/Utilities>
<use name=FWCore/ServiceRegistry>
<use name=DataFormats/Common>
<use name=DataFormats/JetReco>
<use name=DataFormats/MuonReco>
<use name=DataFormats/PatCandidates>
<use name=PhysicsTools/PatUtils>
<use name=PhysicsTools/UtilAlgos>
<use name=PhysicsTools/Utilities>
<use name=CommonTools/UtilAlgos>
<flags EDM_PLUGIN=1>
<export>
   <lib name=MyAnalyzersPatOne>
</export>
```

- Para la compilacion tendremos que hacer:

```
scram b -c
scram b -j 8

[hep] /home/aosorio/scratch/Tutorials/CMSSW_3_11_3/src/MyAnalyzers/PatOne > scram b
Reading cached build data
>> Local Products Rules ..... started
>> Local Products Rules ..... done
>> Entering Package MyAnalyzers/PatOne
>> Updating python symlinks.
>> Compiling cfi/cff python modules: src/MyAnalyzers/PatOne/python
```

```
Entering library rule at MyAnalyzers/PatOne
>> Compiling /scratch/aosorio/Tutorials/CMSSW_3_11_3/src/MyAnalyzers/PatOne/src/PatOne.cc
>> Building shared library tmp/slc5_amd64_gcc434/src/MyAnalyzers/PatOne/src/MyAnalyzersPatOne/lib
@@@@ Checking shared library for missing symbols: libMyAnalyzersPatOne.so
@@@@ ----> OK, shared library FULLY-BOUND (no missing symbols): libMyAnalyzersPatOne.so
Leaving library rule at MyAnalyzers/PatOne
@@@@ Running edmWriteConfigs for MyAnalyzersPatOne
--- Registered EDM Plugin: MyAnalyzersPatOne
>> Leaving Package MyAnalyzers/PatOne
>> Package MyAnalyzers/PatOne built
```

Creacion de Histogramas

Para la creacion de histogramas desde un Analyzer se utiliza el servicio **TFileService**, el cual crea un archivo de salida **.root**. El servicio y los histogramas se inizializan preferiblemente en el metodo "beginJob()".

- Una forma recomendada de administrar los histogramas es mediante un contenedor. En C++ podemos usar un **std::map**. Esto nos permite almacenar y llamar un histograma por medio de una llave. Necesitamos primero declarar nuestro contenedor como parte de nuestra clase:

```
//Adicionar esta linea en PATOne.h
std::map< std::string, TH1F * > h_1DHistContainer;
```

Ahora podemos crear los histogramas de 1D que deseemos:

```
  // Estas lineas se colocan en la definicion del metodo beginJob() en PATOne.cc
  edm::Service<TFileService> fs;
  // book histograms:
  h_1DHistContainer["Nmuons"] = fs->make<TH1F>("Nmuons",   "Muon multiplicity",     10, 0,  10);
  h_1DHistContainer["Njets" ] = fs->make<TH1F>("Njets",    "Jet multiplicity",      10, 0,  10);
  h_1DHistContainer["met"   ] = fs->make<TH1F>("MET",      "missing E_{T}",         20, 0, 100);
```

Ahora si podemos retirar los comentarios en nuestro **analyzer PATOne** y llenar los histogramas. Notar que es muy facil llamar al histograma y que no manejamos como un apuntador directamente, puesto que al utilizar la llave sobre el contenedor, este nos devuelve uno.

Creacion de una ROOT-tupla

Existe varias formas de crear ROOT-tuplas dentro de CMSSW. Intentare describir primero la forma mas basica, utilzando ROOT.

Una ROOT-tupla pequeña

En ROOT la clase que administra la creacion, lectura y escritura de tuplas se llama TTree. En este ejemplo crearemos un TTree sencillo al que llenaremos con algunas variables.

- Primero, debemos incluir en nuestro archivo PATOne.h un "inlucde "TTree.h" y declarar un apuntador a un TTree como miembro de la clase:

```
  TTree * m_tree;

  //Declarar las variables que iran dentro del arbol
  int m_njets;
```

```
  float m_njetE[20];
```

- Segundo, y al igual que con los histogramas, debemos ir al metodo **beginJob()** para definir nuestro **TTree**:

```
  m_tree = new TTree("t1","PatOne ROOT-tuple");

  //adicionar una rama por variable
  m_tree->Branch("nJets"    ,&m_njets     ,"nJets/I");
  m_tree->Branch("nJetE"   , m_njetE      ,"nJetE[nJets] /F");
```

- Tercero, procedemos a llenar nuestras variables:

```
m_njets = nJets;

///.........

/// cuando todas las variables del evento esten listas, se llena el arbol:

m_tree->Fill();
```

Creacion de una EDM-tuple

Como su nombre lo indica, una EDM-tuple mantiene la estructura que CMSSW da a cada evento en el llamdo Event Data Model. Es decir se mantienen las colecciones, con todas las ventajas que esto trae. Existen varios mecanismos para hacerlo. Uno de ellos se describe a continuación.

- El concepto clave en este ejercicio es el de **Filter** (o filtro) en CMSSW
- Por ello, lo primero que haremos es crear un filtro usando el script que nos arma un esqueleto
- Vamos entonces a nuestra area de trabajo en CMSSW_3_11_3/src/, preparemos el entorno para CMSSW, y creemos un esqueleto para hacer un filtro:

```
[hep] cd CMSSW_3_11_3/src
[hep] cmsenv
[hep] cd MyAnalyzers/
[hep] mkedfltr EDMtupler
```

- Inspeccionemos el modulo que se ha creado:

```
[hep] cd EDMtupler
[hep] ls
BuildFile  doc  interface  python  src  test
[hep] cd src
[hep] emacs EDMtupler.cc
```

- Este nuevo modulo es un **Filter**, el cual tiene la capacidad de producir (y filtrar) archivos EDM. Esta propiedad de **producir** nueva informacion es la que vamos a explotar para crear nuestra EDM-tuple.

- En el constructor de nuestro modulo insertemos las siguientes lineas:

```
   //un input TAG (no olvidar declararlo como miembro de esta clase (es decir adicionar edm::Inpu

   m_muonSrc = iConfig.getUntrackedParameter<edm::InputTag>("muonSrc");

   //estas son las variables que guardaremos en nuestra EDMtuple:

   produces< int    > ("isGoodMuon").setBranchAlias("isGoodMuon");
   produces< double > ("muonPx").setBranchAlias("muonPx");
   produces< double > ("muonPy").setBranchAlias("muonPy");
   produces< double > ("muonPz").setBranchAlias("muonPz");
   produces< double > ("muonE").setBranchAlias("muonE");
```

- Aqui estamos expresando que vamos a producir ciertas "ramas" -tal como manejamos en una ROOT-tuple. Para este ejemplo, la idea es obtener alguna informacion sobre muones o pat::Muon. No olvidar adicionar el siguiente **include** al principio del archivo:

```
#include "DataFormats/PatCandidates/interface/Muon.h"
```

- Ahora podemos trabajar en el metodo **filter** y adicionar lo siguiente:

```
   // Create memory to be put into event
   std::auto_ptr< int    > isGoodMuon     (new int   );
   std::auto_ptr< double > muonPx        (new double);
   std::auto_ptr< double > muonPy        (new double);
   std::auto_ptr< double > muonPz        (new double);
   std::auto_ptr< double > muonE         (new double);
```

* Con esto, estamos creando y asignando la memoria para las "ramas" que iran en nuestra EDM-tuple. Documentacion sobre estos apuntadores especiales llamadados **auto_ptr** se puede leer aqui. Estamos listos para llenar nuestro EDM-tuple con la informacion que queremos. Por ejemplo:

```
  // 1. Muon collection
  edm::Handle<edm::View<pat::Muon> > muons;
  iEvent.getByLabel( m_muonSrc, muons);

  for(edm::View<pat::Muon>::const_iterator muon=muons->begin(); muon!=muons->end(); ++muon) {

  ///llenamos la informacion que termina en nuestra EDM-tupla
  if( muon->isGlobalMuon() ) {
      *isGoodMuon = 1;
      *muonPx = muon->px();
      *muonPy = muon->py();
      *muonPz = muon->pz();
      *muonE  = muon->energy();
    } else {
      *isGoodMuon = -1;
      *muonPx = -1.0;
      *muonPy = -1.0;
      *muonPz = -1.0;
      *muonE  = -1.0;

    }

  }
```

- Listo! Tenemos una EDMtupla a la que le hemos adicionado algunas variables con informacion sobre muones. El filtro por el momento siempre devuelve un valor "true", pero esto no es obligatorio. Podemos hacer uso de esto para crear EDMtuples que sean selectivas sobre cierta propiedad o propiedades del evento.

- Para compilar y correr necesitamos editar el archivo **Buildfile** y crear un archivo de configuracion. Lamentablemente, el script que arma el esqueleto del filtro no provee un archivo de configuracion modelo. Por simplicidad (y dado que no hay nada nuevo en ellos) ustedes pueden bajar de este Twiki los dos archivos necesarios:

- edmtupler_cfg.py.txt: Archivo de configuracion para correr el EDMtupler
- BuildFile.xml: Buildfile para compilar el EDMtupler

- O si lo prefieren, los pueden descargar directamente en su area:

```
[hep] wget --no-check-certificate https://twiki.cern.ch/twiki/pub/Main/CMSUniandesGroupAnalysis/e
[hep] wget --no-check-certificate https://twiki.cern.ch/twiki/pub/Main/CMSUniandesGroupAnalysis/B
```

- Veamos que contiene el archivo de configuracion:

```
import FWCore.ParameterSet.Config as cms

process = cms.Process("Demo")

process.load("FWCore.MessageService.MessageLogger_cfi")

process.maxEvents = cms.untracked.PSet( input = cms.untracked.int32(100) )

process.source = cms.Source("PoolSource",
                            fileNames = cms.untracked.vstring(
    'file:/scratch/aosorio/data/HSCPs/tutorial/patTuple_HSCPs.root'
    )
                            )

process.applyFilter = cms.EDFilter('EDMtupler',
                                   muonSrc     = cms.untracked.InputTag("cleanPatMuons") )

process.p = cms.Path(process.applyFilter)

process.out = cms.OutputModule("PoolOutputModule",
                               SelectEvents = cms.untracked.PSet( SelectEvents = cms.vstring('p')
                               fileName = cms.untracked.string('EDMtuple.root'),
                               outputCommands = cms.untracked.vstring('keep *','drop *_cleanPatPh

process.e = cms.EndPath(process.out)
```

- Al ejecutar el EDMtuple sobre este archivo de configuracion dado, el filtro actua -en el momento no hay ninguna regla que se aplique- y todos los eventos son guardados con sus colecciones en el archivo de salida "EDMtuple.root".

- Despues de correr, podemos ver que contiene el archivo de salida haciendo uso del comando de CMSSW **edmDumpEventContent**:

```
 edmDumpEventContent EDMtuple.root
Type                                Module              Label         Process
-------------------------------------------------------------------------------
```

```
edm::OwnVector<reco::BaseTagInfo,edm::ClonePolicy<reco::BaseTagInfo> >    "selectedPatJets"     "
vector<CaloTower>                       "selectedPatJets"     "caloTowers"   "PAT"
vector<pat::Electron>                   "cleanPatElectrons"   ""             "PAT"
vector<pat::Jet>                        "cleanPatJets"        ""             "PAT"
vector<pat::MET>                        "patMETs"             ""             "PAT"
vector<pat::Muon>                       "cleanPatMuons"       ""             "PAT"
vector<pat::Photon>                     "cleanPatPhotons"     ""             "PAT"
vector<pat::Tau>                        "cleanPatTaus"        ""             "PAT"
vector<reco::GenJet>                    "selectedPatJets"     "genJets"      "PAT"
double                                  "applyFilter"         "muonE"        "Demo"
double                                  "applyFilter"         "muonPx"       "Demo"
double                                  "applyFilter"         "muonPy"       "Demo"
double                                  "applyFilter"         "muonPz"       "Demo"
edm::TriggerResults                     "TriggerResults"      ""             "Demo"
int                                     "applyFilter"         "isGoodMuon"   "Demo"
[hep] /home/aosorio/scratch/Tutorials/CMSSW_3_11_3/src/MyAnalyzers/EDMtupler >
```

- Como vemos, se mantienen todas colecciones -que pueden ser de utilidad- y adicionalmente tenemos nuestras variables.

- Si queremos, podemos eliminar colecciones que no se necesitan. Para ello debemos editar el archivo de configuracion. Buscar la linea siguiente:

```
outputCommands = cms.untracked.vstring('keep *') )
```

- y adicionar la siguiente instruccion:

```
outputCommands = cms.untracked.vstring('keep *','drop *_cleanPatPhotons_*_*') )
```

- Con esta instruccion, nos libramos de la colecciones asociadad a la etiqueta "cleanPatPhotons". Para comprobarlo, podemos volver a ver que ha quedado en nuestro archivo de salida:

```
 edmDumpEventContent EDMtuple.root

Type                                    Module                Label          Process
-------------------------------------------------------------------------------------
edm::OwnVector<reco::BaseTagInfo,edm::ClonePolicy<reco::BaseTagInfo> >    "selectedPatJets"     "
vector<CaloTower>                       "selectedPatJets"     "caloTowers"   "PAT"
vector<pat::Electron>                   "cleanPatElectrons"   ""             "PAT"
vector<pat::Jet>                        "cleanPatJets"        ""             "PAT"
vector<pat::MET>                        "patMETs"             ""             "PAT"
vector<pat::Muon>                       "cleanPatMuons"       ""             "PAT"
vector<pat::Tau>                        "cleanPatTaus"        ""             "PAT"
vector<reco::GenJet>                    "selectedPatJets"     "genJets"      "PAT"
double                                  "applyFilter"         "muonE"        "Demo"
double                                  "applyFilter"         "muonPx"       "Demo"
double                                  "applyFilter"         "muonPy"       "Demo"
double                                  "applyFilter"         "muonPz"       "Demo"
edm::TriggerResults                     "TriggerResults"      ""             "Demo"
int                                     "applyFilter"         "isGoodMuon"   "Demo"
```

- Ya no aparece la coleccion que hemos eliminado. Listo!!!!

Copyright &© 2008-2019 by the contributing authors. All material on this collaboration platform is the property of the

contributing authors.
Ideas, requests, problems regarding TWiki? Send feedback