CMSSW Tutorial

Gordon Ball
Imperial College London

gordon.ball@cern.ch
GPG 0x324543E5

Summary

- We cover
  - Checking out a CMSSW release
  - Creating a package
  - Writing a configuration file
  - Writing a
    - Filter
    - Producer
    - Analyser
  - Submitting to the grid
  - Comparing data and MC

General

- This can be done either on the lx0X shared machines at Imperial, lxplus5 at CERN or your own (SLC5/centos5) linux desktop
- You will need a valid grid certificate with CMS VOMS role in order to access data stored on our SE, and to submit grid jobs
- The object of the exercise is to make a muon-muon mass spectrum
- We will be using real CMS data from Run2010B
- The later, more time-consuming parts are optional (but probably helpful)
- I'm assuming you're using a bash-type shell, where relevant

Checking out CMSSW

- Source the general CMS environment
  - `source /vols/cms/grid/setup.sh`
- List the available versions
  - `scram list CMSSW`
- Check out version 3.8.7
  - `scram project CMSSW CMSSW_3_8_7`
  - A CMSSW area can grow quite large (~100sMB), do this somewhere with sufficient space. Your AFS home on LXPLUS isn't a good choice.
- Get the environment for this version
  - `cd CMSSW_3_8_7`
  - `cmsenv`
- You should now have a fresh CMSSW area, and commands like `cmsRun` should now be in your `$PATH`
- You'll only ever use the `src/` directory in the release area

- The src directory will contain
  - Any packages you write
  - Packages you check out from CVS to use or modify
- All packages in a standard CMSSW release are available to use without being checked out – you only need to have code here if it isn't in CMSSW or you need to use a modified version
- Make a new package
  - mkdir -p Tutorial/Exercise

gordon.ball@cern.ch
GPG 0x324543E5

- Our subpackage is an empty directory, which we need to populate (there is a script for this, but we'll do it by hand here)
- Create directories
  - src
  - interface
  - python
  - test
- Create a file (in the subpackage directory)
  - BuildFile
- Copy the contents from `/home/hep/gfball/tutorial/BuildFile`
  - Writing a Buildfile by hand is usually only necessary if you're doing something exotic, we'll just use a stock one
  - This defines which libraries need to be linked to this one at runtime

Configuration File

- Copy a skeleton config file from `/home/hep/gfball/tutorial/config_cfg.py` to `test/`
  - The _cfg suffix is not required but indicates this is a complete, runnable config not one intended to be included in another file
- This file
  - imports the CMSSW configuration system (mandatory)
  - creates a Process object called process (mandatory)
  - creates a Source to load some files from the SE (necessary unless running pythia et al)
  - sets a limit on the number of events to be processed
  - defines a Path which is currently empty
- Since we're going to be accessing files on the SE, you need to create a grid proxy
  - `voms-proxy-init -voms cms`

gordon.ball@cern.ch
GPG 0x324543E5

Using existing code

- It is always better not to write your own if you don't have to
- We will use an existing filter to select events which have a well-reconstructed vertex (a good start for looking at data)
- Add to the config file

  -
    ```
    process.vertex_filter =
      cms.EDFilter("GoodVertexFilter", vertexCollection =
      cms.InputTag("offlinePrimaryVertices"), minimumNDOF
      = cms.uint32(4), maxAbsZ = cms.double(25), maxd0 =
      cms.double(2)))
    ```
- This loads the filter, but it also needs to be in the Path to be run
- Edit the path to
  - `process.path = cms.Path(process.vertex_filter)`
- You can now run this with
  - `cmsRun config_cfg.py`
- But it will still do nothing particularly useful

- There we defined the parameters for a pre-existing module ourselves, but authors are supposed to provide a python fragment containing recommended parameters so we don't need to know them ourselves
- Remove the definition and add instead
  - process.load(“DPGAnalysis.Skims.goodvertexSkim_cff”)
- This loads the file
  - DPGAnalysis/Skims/python/goodvertexSkim_cff.py (look in CVS/LXR)
- Then you need to add the object it defines to your Path instead of the one you defined
  - process.path = cms.Path(process.goodvertexSkim)
    - The name in this case is the same as the filename, but this isn't always the case – you usually need to look in the file you're loading and see what is defined
- Unless there is a good reason to change parameters, you should load defaults where possible

Our own filter

- Copy the skeletons
  - `/home/hep/gfball/tutorial/Filter.cc` to src/TwoMuonFilter.cc
  - `/home/hep/gfball/tutorial/Filter.h` to interface/TwoMuonFilter.h
- Make a class called “TwoMuonFilter”
- We want to work with muons (class reco::Muon), for which we need the appropriate header file.
  - Use the identifier search at http://cmslxr.fnal.gov/lxr to look it up
  - You will need both the file Muon.h (find the path) defining the object and MuonFwd.h (defining collections of, references to, etc)

Collection names

- Use edmFileUtil to list the branches in one of the data files
  - `edmFileUtil -P -f <filename> | grep -i muon`
- This produces lots of lines like
  - Branch XXX of Events tree. NAME. total size = YYYYY
- NAME is of the form
  - CLASS_COLLECTION_[INSTANCE]_PROCESS
- For the main muon collection
  - CLASS is “recoMuons” (meaning std::vector<reco::Muon>)
  - COLLECTION is “muons” (meaning the producer was named “muons”)
  - INSTANCE is empty
  - PROCESS is “RECO”
- Usually you'll only need to know the collection and class, the others only matter if one producer produces multiple collections, or the file has been reprocessed and new versions of the same objects added

- In the `bool filter(Event, EventSetup)` method, called with each event
- Create a Handle for the muon collection
  - `edm::Handle<reco::MuonCollection>` ...
- Load the collection from the file
  - `Event.getByLabel(label, handle)`
  - (Using the label you found with edmFileTool, “muons”)
- Print to screen the size of the muon collection
  - Use `std::cout` as normal
  - `reco::MuonCollection` is a `typedef` for `std::vector<reco::Muon>`
- Return `true` if there are at least two muons

- Add your two-muon filter to the config file
  - `process.twoMuonFilter = cms.EDFilter(“TwoMuonFilter”)`
- Add the filter to the end of the config file path
  - using “+”, not “,”, ie
  - cms.Path(module1+module2+module3....)
- Run CMSSW and you should see muon counts on the screen

Producer

- Having identified events with at least two muons, we want to reconstruct their possible parents
- Copy the producer src and interface skeletons from /home/hep/gfball/tutorial
- Make a new class named “ParentProducer”
- We will produce instances of reco::LeafCandidate (the most basic particle that can be created)
  - Look up the include file for this
- We have to add a call to the constructor declaring the new collection we will create
  - produces<std::vector<reco::LeafCandidate> >();

gordon.ball@cern.ch
GPG 0x324543E5

- In the `void produce(Event, EventSetup)` method
- Declare a collection to hold the pair-candidates we create
  - `std::auto_ptr<std::vector<reco::LeafCandidate> > parentCollection(new std::vector<reco::LeafCandidate>);`
- Load all muons as before
- Perform a nested loop to find all possible pairs
- Require that pairs have opposite charges
- Create a new LeafCandidate from the sum of the child lorentz vectors (and with charge 0)
  - `reco::LeafCandidate(int charge, LorentzVector& p4)`
- Add this to the new collection
  - `parentCollection->push_back`
- Finally, add the new collection to the Event
  - `event.put(parentCollection);`

Analyser

- Add your producer to the process and path in the Config
  - process.parentProducer = cms.EDProducer(“ParentProducer”)
- This doesn't immediately do anything if run – if we saved the resulting output it would now contain a new collection, but we want to make a histogram of it
- Copy analyser skeletons from /home/hep/gfball/tutorial and create an analyser “ParentAnalyser” (beware Analyser vs Analyzer)

TFileService

- TFileService allows you to use the same TFile from multiple modules (you could create a TFile separately in each module, but this is easier)
- Declare a TFileService in the config file
  - `process.TFileService = cms.Service(“TFileService”, fileName=cms.string(“exercise.root”))`
- Get a reference to it in the analyser constructor
  - `edm::Service<TFileService> tfs;`
- Declare some histograms in the class definition
  - `private:  TH1F* hMass;`
- Use the fileservice to make the histogram
  - `tfs->make<TH1F>(“hMass”, “mu-mu mass”, 200, 0, 200);`

gordon.ball@cern.ch
GPG 0x324543E5

- Create a handle for vector<LeafCandidate> and load the collection like you loaded the muons
  - The label is the name attached to the producing class in the python config, eg “parentProducer” for process.parentProducer
- Loop over the parent candidates, filling the histogram with their mass
  - hMass->Fill(candidate.p4().M());
- Run CMSSW, and you should get a root file output in the current directory containing the mass plot
- There won't be sufficient events in the one file your config currently examines
  - Look up the dataset “/Mu/Run2010B-Nov4ReReco_v1/RECO” using https://cmsweb.cern.ch/dbs_discovery/
  - Expand the list of sites to find Imperial, and get the list of filenames (LFNs) available, then add them to the config
  - There are about 500k events available, about 50k should be enough to see a Z peak

Parameters

- At the moment, all our parameters (such as they are) are hardcoded
- It is better to have general-purpose modules that read variables from the python config
- Retool the ParentProducer to place a Pt and Eta cut on the muons considered as pairs
- Add to the class definitions doubles to hold the cut values
- Read the cut values from the configuration in the constructor
  - `muonPt = cfg.getParameter<double>(“muonPt”);`
- Add these values to your config file

  -
    ```
    process.parent_producer =
      cms.EDProducer(“ParentProducer”, muonPt =
      cms.double(15)...
    ```
- Accept Pt>15 and |Eta| < 2.5

gordon.ball@cern.ch
GPG 0x324543E5

Config Fragment

- As before, it would be useful to provide a config fragment with defaults for this module
  - Create python/parentProducer_cff.py
- In the file
  - import cms as per your config file
  - do not create a process – this isn't a runnable file
  - create a default ParentProducer
  - adjust your config file to load this instead of defining ParentProducer directly
- Note you'll need to build the area with `scram b` so the appropriate links are made to this python file

Grid/CRAB

- You should now have a working configuration that analyses Muon events and plots the mass distribution of their pairs into a root file, which should show a declining distribution with small peaks for resonances
- This is a very crude analysis (we've not done quality cuts on the muons, isolation, checked the trigger, etc) but it will hopefully show something
- We can't run on more than a few tens of thousands of events interactively before it takes too long
- We should use CRAB to split it into lots of jobs and submit to the grid
- Set up CRAB
  - `source /vols/cms/grid/CRAB/current/crab.sh`
  - (in a shell with cmssw environment already configured)

- You need a crab.cfg file, copy a skeleton from /home/hep/gfball/tutorial
  - The data to run over
    - CMSSW.datasetpath = /Mu/Run2010B_Nov4ReReco-v1/RECO
  - The configuration file to use
    - CMSSW.pset = config_cfg.py
  - How to split the data (require 2 of 3)
    - CMSSW.total_number_of_lumis = -1
    - CMSSW.lumis_per_job
    - CMSSW.number_of_jobs = 50
  - What output is to be returned
    - CMSSW.output_file = exercise.root
  - Where the output is to be returned to
    - USER.copy_data = 1
    - USER.storage_element = T2_UK_London_IC
    - USER.user_remote_dir = tutorial2011

- When this the config file is written, submit the jobs with
  - Double check the config file still works first
  - `crab -create -submit`
- A site hosting the data will be located and your jobs sent there
- To find out how it is going
  - `crab -status`
  - (this will provide a monitoring URL as well as output)
- Once the jobs are finished, you should have a set of root files in
  - `$DCACHE_SRM_ROOT/store/user/<username>/tutorial2011`
- You need to download these locally with lcg-cp/srmcp/sesync
- You need to write a ROOT or PyROOT script to load all the files, extract the ~20 histograms and add them all together
  - Or use `hadd output inputs...`

Comparing it to MC

- A bit more advanced and requiring quite a bit of time, optional...
- After the jobs are finished, get a lumi report with
  - `crab -status; crab -get; crab -report`
- You can then find out how much luminosity was analysed with
  - `lumiCalc.py -i <crab_dir>/res/lumiSummary.json overview`
- Then you need to repeat the analysis on an appropriate MC sample
  - Search DBS discovery for a Fall10 DYtoMuMu sample
  - You need to replace references to “lumis” in crab.cfg with “events”, eg “events_per_job” for MC instead of data
  - The cross section can be found on the generation twiki page, eg GeneratorProduction2010
  - The NNLO cross section for DYtoMuMu is ~1666pb
- Appropriately scale and superimpose the mass plots