Tracking Hands-On Session

Tracking Training Day – Oct. 18, 2013

G. Sguazzoni, G.Cerati

Goals

- Locate and browse tracking code
- Find samples for developments & tests
- Run track reconstruction sequence (default or modified)
  - check timing and memory use
- Refit tracks
- Analyze tracks in EDM root files
- Analyze tracks with EDAnalyzer
  - select tracks
  - track parameters and HitPattern
  - hits and track hits
  - seeds, trajectory
- Run Track Validation
  - MultiTrackValidator
  - SeedValidator
- Advanced debugging

But, first of all let's setup a working area

This should be pretty easy for everybody:

```
ssh -XY lxplus.cern.ch
cd scratch0
mkdir TrackingTutorial
cd TrackingTutorial
cmsrel CMSSW_7_0_0_pre5
cd CMSSW_7_0_0_pre5/src
cmsenv
```

Locate and browse tracking code

- Main packages for tracking are:
  - DataFormats (/TrackReco, /TrajectorySeed, /TrackingRegHit, ...)
  - RecoTracker: reconstruction code
  - TrackingTools: common utilities with muon tracking
- Recommended starting point for browsing code is CMSSW Software Cross-Reference (lxr):
  - https://cmssdt.cern.ch/SDT/lxr/
  - you can browse or look for files, identifiers (class/method/variable names) and free text
- Let's take a look at a few files:
  - https://cmssdt.cern.ch/SDT/lxr/source/DataFormats/TrackReco/interface/Track.h
  - https://cmssdt.cern.ch/SDT/lxr/source/RecoTracker/IterativeTracking/python/iterativeTk_cff.py
  - https://cmssdt.cern.ch/SDT/lxr/source/DataFormats/TrajectorySeed/interface/TrajectorySeed.h
  - https://cmssdt.cern.ch/SDT/lxr/source/TrackingTools/TransientTrackingRecHit/interface/TransientTrackingRecHit.h
  - https://cmssdt.cern.ch/SDT/lxr/source/TrackingTools/PatternTools/interface/Trajectory.h

Find samples for developments & tests

The best samples for developing and testing Tracking are RelVal. Typical developments do not need huge datasets. RelVals are made for every (pre-) release and thus are always up to date. They contain all the information needed (clusters, tracks, mc truth) that won't be available on other samples.

Let's start from a SingleMu sample. We want it to be at CERN for a quick access:

```
dbsql "find dataset, site where dataset like /RelValSingleMu*CMSSW_7_0_0_pre5*GEN-SIM-RECO"
```

Now, let's choose Pt10 and find the files:

```
dbsql "find file where dataset=/RelValSingleMuPt10/CMSSW_7_0_0_pre5-PRE_ST62_V8-v1/GEN-SIM-RECO"
```

Another useful sample is TTbar:

```
dbsql "find dataset, site where dataset like /RelValTTbar*CMSSW_7_0_0_pre5*GEN-SIM-RECO"
```

Let's pick the inclusive sample with PU:

```
dbsql "find file where dataset=/RelValTTbar/CMSSW_7_0_0_pre5-PU_PRE_ST62_V8-v1/GEN-SIM-RECO"
```

For validation we will need also the GEN-SIM-DIGI-RAW-HLTDEBUG files:

```
dbsql "find file where dataset=/RelValSingleMuPt10/CMSSW_7_0_0_pre5-PRE_ST62_V8-v1/GEN-SIM-DIGI-RAW-HLTDEBUG"
dbsql "find file where dataset=/RelValTTbar/CMSSW_7_0_0_pre5-PU_PRE_ST62_V8-v1/GEN-SIM-DIGI-RAW-HLTDEBUG"
```

Run default track reconstruction sequence

We need a configuration file with standard setting to run the default tracking sequence.

A file was already prepared for you, let's copy it and take a look:

```
cp /afs/cern.ch/cms/tracking/TrainingDay/recoTrk_cfg.py .
emacs recoTrk_cfg.py &
```

Main ingredients are input file, GlobalTag, default services and sequences. RECO files store clusters, not hits. Before running tracking we need to produce hits.

Now let's run it:

```
cmsRun recoTrk_cfg.py
```

Open the output file and look for the tracks we have made.

```
root -l reco_trk.root
TBrowser b
Events->Draw("recoTracks_generalTracks__RERECO.@obj.size()")
Events->Draw("recoTracks_generalTracks__RERECO.obj.chi2_")
Events->Draw("recoTracks_generalTracks__RERECO.obj.algorithm_")
Events->Draw("sqrt(recoTracks_generalTracks__RERECO.obj.momentum_.perp2())")
```

Enable services for timing summary and memory info and re-run.

Run custom track reconstruction sequence

What are the exact parameters we have used in the default reconstruction? For example, what are the layers used for seeding the initial step of iterative tracking?

```
edmConfigDump recoTrk_cfg.py >& dump.recoTrk_cfg.txt
less dump.recoTrk_cfg.txt
```

Look for “process.initialStepSeeds” and then for “PixelLayerTriplets”. PixelLayerTriplets is the “ComponentName” of “process.pixellayertriplets”.

Let’s modify the “layerList” by removing the first combination, i.e:

```
'BPix1+BPix2+BPix3'
```

What differences would you expect?

Modify the output file name to 'reco_trk_mod.root' and add the following lines to recoTrk_cfg.py,

```
process.pixellayertriplets.layerList = cms.vstring('BPix1+BPix2+FPix1_pos','BPix1+BPix2+FPix1_neg',
                                                   'BPix1+FPix1_pos+FPix2_pos','BPix1+FPix1_neg+FPix2_neg')
```

Rerun and look at the output file:

```
cmsRun recoTrk_cfg.py
root -l reco_trk_mod.root
Events->Draw("recoTracks_generalTracks__RERECO.obj.algorithm_")
```

Refit tracks

A very useful tool is the TrackRefit. It takes as input the track collection and refits the tracks with the same or with modified conditions (alignment, magnetic field) or algorithms (propagator, cluster parameter estimator). Refitting tracks gives access to useful objects, like hits positions and the trajectory (see later), and it is much faster than full re-tracking.

Inspired from RecoTracker/TrackProducer/test/TrackRefit.py, a configuration file for refitting track was prepared.

Copy it locally and open it:

```
cp /afs/cern.ch/cms/tracking/TrainingDay/refitTrk_cfg.py .
emacs refitTrk_cfg.py &
```

Where is the TrackRefitter module defined? Can you browse the code and find it?

Now we can run it and compare original and refitted tracks:

```
root -l refit_trk_analytical.root
TCanvas c1
Events->Draw("sqrt(recoTracks_TrackRefitter__Refitting.obj.momentum_.perp2())")
TCanvas c2
Events->Draw("sqrt(recoTracks_generalTracks__RERECO.obj.momentum_.perp2())")
```

What if we want to refit tracks under difference conditions? For example we may want to neglect energy loss (and detailed magnetic field description from Runge Kutta propagation). We can add the following lines to refitTrk_cfg.py:

```
process.TrackRefitter.Propagator = cms.string('AnalyticalPropagator')
process.RKTrajectoryFitter.Propagator = cms.string('AnalyticalPropagator')
process.RKTrajectorySmoother.Propagator = cms.string('AnalyticalPropagator')
```

What would you expect? Now, let's open again the file and check how big was the effect.

Preparing an EDAnalyzer

Luckily, there is a simple command (mkedanlzr) to generate the skeleton of an EDAnalyzer accessing tracks and making plots.

```
mkdir TrackingTests
cd TrackingTests/
mkedanlzr DemoTrackAnalyzer example_track example_histo
cd DemoTrackAnalyzer/
mv python/ConfFile_cfg.py demoTrackAnalyzer_cfg.py
```

we will have to edit the following files:

```
emacs plugins/DemoTrackAnalyzer.cc plugins/BuildFile.xml demoTrackAnalyzer_cfg.py &
```

Then we can compile, run and open the output file with the usual commands:

```
scramv1 b -j 8
cmsRun demoTrackAnalyzer_cfg.py
root -l trackAnalysis.root
```

But before, let's edit it step by step.

Step1: plot track variables and hit pattern

As a first step we want to add the plot for the track pT, the number of crossed layers and the number of outer layers with no hits.

The skeleton already contains the declaration of the output file, of a histogram and a loop over tracks where the histogram is filled.

You can surely add the plot of the track pT. Any idea how to get the information about layers with/without hits?

The config file needs to be slightly modified. First replace the old collection name 'ctfWithMaterialTracks' with the up-to-date 'generalTracks'. Also, 'trackAnalysis.root' looks more appropriate than 'histo.root'.

An example of the code can be found at:

```
/afs/cern.ch/cms/tracking/TrainingDay/step1/
```

Another option for the input collection is a custom selection. A very useful tool is the string-based track selector, for example:

```
process.selectedTracks = cms.EDFilter('TrackSelector',
     src = cms.InputTag('generalTracks'),
     cut = cms.string("quality('highPurity') & (algo=4) & abs(eta)<0.9")
)
```

Ok, this should have been pretty easy. Let's do something more advanced in the next step.

Step2: access to hit information (1)

Tracking analyses are often related to hit properties. In this step we'll learn how to access hits on a track and hits from the inclusive hit collection. Two main ingredients that are needed are the TransientTrackingRecBuilder and the TrackRefitter.

In fact, hits stored on disk (TrackingRecHit) miss some relevant information:

1. local position is not stored and depends on the track direction hypothesis; therefore hits on track need to be refitted
2. they do not contain information about the tracker geometry and it is needed to translate local into global position; therefore, to access the global position of a hit, a transient version (TransientTrackingRecHit) linked to the tracker geometry needs to be created using the TransientTrackingRecHitBuilder.

We already know how to run the Refitter, so we can add it to cfg file and use tracks from refitter as input (keep in mind that to do this you need also the globalTag and the standard includes).

Step2: access to hit information (2)

Now we need to get a TransientTrackingRecHitBuilder performing the following steps:

1. Include needed files:

```
#include "FWCore/Framework/interface/ESHandle.h"
#include "TrackingTools/Records/interface/TransientRecHitRecord.h"
#include "TrackingTools/TransientTrackingRecHit/interface/TransientTrackingRecHitBuilder.h"
```

2. Declare the variables for the builder name and the builder itself:

```
std::string builderName;
const TransientTrackingRecHitBuilder* builder;
```

3. In the constructor, get the builder name from the config file:

```
builderName(iConfig.getParameter<std::string>("TTRHBuilder"))
```

4. Uncomment 'beginRun' and get the builder from the EventSetup:

```
virtual void beginRun(edm::Run const&, edm::EventSetup const&) override;
[...]
void DemoTrackAnalyzer::beginRun(edm::Run const& run, edm::EventSetup const& setup) {
  edm::ESHandle<TransientTrackingRecHitBuilder> theBuilder;
  setup.get<TransientRecHitRecord>().get(builderName,theBuilder);
  builder=theBuilder.product();
}
```

5. Set the builder name in the config file:

```
TTRHBuilder = cms.string('WithAngleAndTemplate')
```

6. Add the proper library in the BuildFile.xml:

```
<use name="TrackingTools/Records"/>
<use name="TrackingTools/TransientTrackingRecHit"/>
```

Step2: access to hit information (3)

We want to plot 2D maps of the hits, on the ZR plane for the hits on track and on the XY plane for all pixel barrel hits. Also, for pixel hits, we want to know how the various layers are populated (number of hits per layer).

First we should include TH2.h, then declare and book the histograms: what are sensible ranges and binning?

Can you find out how to loop over track hits?

Once you found it you can build the TransientTrackingRecHit with:

```
TransientTrackingRecHit::RecHitPointer thit=builder->build(&**hit);
```

and then fill the ZR map.

To access pixel hits, as you recall, first we need to add process.siPixelRecHits to the Path because only clusters are saved in RECO data format.

Step2: access to hit information (4)

Then you can add to the C++ code the following:

```
// access to hit collections
//typedef edmNew::DetSetVector<SiPixelRecHit> SiPixelRecHitCollection;
edm::Handle<SiPixelRecHitCollection> pixelHits;
iEvent.getByLabel("siPixelRecHits", pixelHits);
for (SiPixelRecHitCollection::const_iterator it = pixelHits->begin(); it!=pixelHits->end(); it++ ) {
  //outer loop over modules (DetId)
  DetId hitId = it->detId();
  for (SiPixelRecHitCollection::DetSet::const_iterator hit = it->begin(); hit!=it->end(); hit++ ) {
    //inner loop over hits in module
    if (hitId.subdetId() == (int) PixelSubdetector::PixelBarrel ) {
      TransientTrackingRecHit::RecHitPointer ttrh = builder->build(&*hit);
      if (ttrh->isValid()) h_pixhits_XYmap->Fill(ttrh->globalPosition().x(),ttrh->globalPosition().y());
      if (PXBDetId(hitId).layer()==1) h_pixhits_layer->Fill(1);
      else if (PXBDetId(hitId).layer()==2) h_pixhits_layer->Fill(2);
      else if (PXBDetId(hitId).layer()==3) h_pixhits_layer->Fill(3);
    } else if (hitId.subdetId() == (int) PixelSubdetector::PixelEndcap ) {
      if (PXFDetId(hitId).disk()==1) h_pixhits_layer->Fill(4);
      else if (PXFDetId(hitId).disk()==2) h_pixhits_layer->Fill(5);
    }
  }
}
```

Do you understand the structure of this code?

Now you can run the analyzer!

Note that example for this step can be found at:

`/afs/cern.ch/cms/tracking/TrainingDay/step2/`

Could you do something similar for strips?

Step3: access to seeds and trajectory (1)

Increasing complexity, two key components of tracking are seeds and the trajectory.

The seed is is the starting point for track building and it is made of a vector of hits and a trajectory state (i.e. track parameters) positioned on the last hit along the direction. The trajectory state stored in the seed is a persistent version (PTrajectoryStateOnDet) that needs to be converted to the usual TrajectoryStateOnSurface via a helper function that needs the MagneticField and the hit surface. Again, one can access to the seed collection or to the seed that originated a specific track.

The trajectory is the object used during the track fit and is made (mainly) of a vector of TrajectoryMeasurements, where each measurement contains a hit plus the trajectory states (predicted and updated). Trajectories are not saved in EDM files but are put into the event when tracks are made (full tracking or refitting). A trajectory-track map can be used to get one from the other.

Step3: access to seeds and trajectory (2)

Let's start from seeds. We want to plot the p_T of the seeds that originated a track and the pseudorapidity of all the seeds used in the InitialStep. Please declare and book the corresponding histograms. Then we need the following:

1. We need to include the headers:

```
#include "DataFormats/TrajectorySeed/interface/TrajectorySeed.h"
#include "DataFormats/TrajectorySeed/interface/TrajectorySeedCollection.h"
#include "TrackingTools/TrajectoryState/interface/TrajectoryStateTransform.h"
#include "MagneticField/Engine/interface/MagneticField.h"
#include "MagneticField/Records/interface/IdealMagneticFieldRecord.h"
```

2. As done for the TTRHB we need to declare a variable for the MagneticField and get it from the EventSetup (no name is needed in this case). Can you do it?
3. Inside the track loop, get the track seed, convert the persistent state to a TSOS and fill the histogram with the seed pT:

```
edm::RefToBase<TrajectorySeed> tkseed = itTrack->seedRef();
if (tkseed->nHits()==3) {
  TransientTrackingRecHit::RecHitPointer recHit2 = builder->build(&*(tkseed->recHits().first+2));
  TrajectoryStateOnSurface state = trajectoryStateTransform::transientState(
                                   tkseed->startingState(), recHit2->surface(), theMF);
  h_track_seed_pt->Fill(state.globalMomentum().perp());
}
```

4. Can you get the InitialStep seed collection, loop over it and fill the eta plot?
5. Keep in mind that you also need to modify the BuildFile:

```
<use name="MagneticField/Engine"/>
<use name="TrackingTools/TrajectoryState"/>
```

6. And that the seed collection is not stored in RECO, so you need to re-run the tracking to get access... please modify the cfg file accordingly.

Step3: access to seeds and trajectory (3)

Now let's focus on trajectories. Our goal is to compute the pull (residual divided by error) of TOB hit local x positions with respect to the forward predicted state. After declaring and booking the histogram, the steps to be made are:

1. Include headers:

```
#include "TrackingTools/PatternTools/interface/Trajectory.h"
#include "TrackingTools/PatternTools/interface/TrajTrackAssociation.h"
```

2. Before the track loop, get the association map and initialize an iterator to the beginning of the map:

```
Handle<TrajTrackAssociationCollection> trajTrackAssociationHandle;
iEvent.getByLabel(tracksTag_,trajTrackAssociationHandle);
TrajTrackAssociationCollection::const_iterator itTrajMap = trajTrackAssociationHandle-
>begin();
```

3. Don't forget to ++ the iterator in track loop "for" statement
4. Inside the track loop, get the trajectory, loop over the TrajectoryMeasurements and, if we are in TOB, fill the pull histogram:

```
Ref<std::vector<Trajectory> > traj  = itTrajMap->key;
vector<TrajectoryMeasurement> trajMeas = traj-> measurements();
for (vector<TrajectoryMeasurement>::iterator meas = trajMeas.begin();
     meas != trajMeas.end(); meas++) {
  TransientTrackingRecHit::ConstRecHitPointer hit = meas->recHit();
  DetId hitId = hit->geographicalId();
  if (hit->isValid() && hitId.subdetId() == (int) StripSubdetector::TOB) {
    TrajectoryStateOnSurface fwdState = meas->forwardPredictedState();
    float delta = hit->localPosition().x()-fwdState.localPosition().x();
    float err2 = hit->localPositionError().xx()+fwdState.localError().positionError().xx();
    h_tobhit_locx_pull->Fill(delta/sqrt(err2));
  }
}
```

Step3: access to seeds and trajectory (4)

Run it on 5k single muon events and 50 ttbar events: do you understand the results?

Find a copy of the code for this step at:

`/afs/cern.ch/cms/tracking/TrainingDay/step3/`

Run the MultiTrackValidator (1)

The tracking validation suite is in Validation/RecoTrack package.
An configuration file example is in test/MultiTrackValidator_cfg.py.
Copy it locally and open it:

```
cp ${CMSSW_RELEASE_BASE}/src/Validation/RecoTrack/test/MultiTrackValidator_cfg.py .
emacs MultiTrackValidator_cfg.py &
```

We need to fix a few things before running (they will be fixed asap in release):

1. Update files with the list we found with dbs
2. Update the globalTag
3. GeometryPilot2_cff is outdated, use Geometry_cff
4. Add the following line to compute efficiency and fake rate:

```
process.multiTrackValidator.runStandalone = cms.bool(True)
```

Run it!

```
cmsRun MultiTrackValidator_cfg.py
```

This will produce validation plots for all highPurity tracks (see how cutsRecoTracks selector is defined). You can add other collection names to "process.multiTrackValidator.label" (e.g. generalTracks or every iterative step as done in Validation/RecoTrack/python/TrackValidation_cff.py).

Official validation plots for each release are at:
http://cmsdoc.cern.ch/cms/Physics/tracking/validation/MC/

Run the MultiTrackValidator (2)

By default, each efficiency plot is made with a different selection (sort of N-1). They are defined in:

`Validation/RecoTrack/python/TrackingParticleSelectionsForEfficiency_cff.py`

You can modify the TrackingParticle selection to test efficiencies in a different phase space. For example, in order to use the same selection for all plots you can do:

```
process.multiTrackValidator.ptMinTP = cms.double(0.4)
process.multiTrackValidator.lipTP = cms.double(35.0)
process.multiTrackValidator.tipTP = cms.double(70.0)
process.multiTrackValidator.histoProducerAlgoBlock.generalTpSelector.ptMin = cms.double(0.4)
process.multiTrackValidator.histoProducerAlgoBlock.generalTpSelector.lip = cms.double(35.0)
process.multiTrackValidator.histoProducerAlgoBlock.generalTpSelector.tip = cms.double(70.0)
process.multiTrackValidator.histoProducerAlgoBlock.TpSelectorForEfficiencyVsEta  =
       process.multiTrackValidator.histoProducerAlgoBlock.generalTpSelector.clone()
process.multiTrackValidator.histoProducerAlgoBlock.TpSelectorForEfficiencyVsPhi  =
       process.multiTrackValidator.histoProducerAlgoBlock.generalTpSelector.clone()
process.multiTrackValidator.histoProducerAlgoBlock.TpSelectorForEfficiencyVsPt   =
       process.multiTrackValidator.histoProducerAlgoBlock.generalTpSelector.clone()
process.multiTrackValidator.histoProducerAlgoBlock.TpSelectorForEfficiencyVsVTXR =
       process.multiTrackValidator.histoProducerAlgoBlock.generalTpSelector.clone()
process.multiTrackValidator.histoProducerAlgoBlock.TpSelectorForEfficiencyVsVTXZ =
       process.multiTrackValidator.histoProducerAlgoBlock.generalTpSelector.clone()
```

Run the SeedValidator

The SeedValidator is a similar to the MultiTrackValidator but works with seeds. It is useful to test existing or new seeding algorithms.

Again, copy the cfg file locally and open it:

```
cp ${CMSSW_RELEASE_BASE}/src/Validation/RecoTrack/test/SeedValidator_cfg.py .
emacs SeedValidator_cfg.py &
```

It will need similar fixes as the MultiTrackValidator, plus moving to the quickTrackAssociatorByHits:

```
process.load("SimTracker.TrackAssociation.quickTrackAssociatorByHits_cfi")
process.quickTrackAssociatorByHits.SimToRecoDenominator = cms.string('reco')
[...]
process.trackerSeedValidator.associators = cms.vstring('quickTrackAssociatorByHits')
```

Once more, since seeds are not saved in RECO, we need to re-run the full tracking on the fly. In this example, it runs over initialStepSeeds.

Run it!

```
cmsRun SeedValidator_cfg.py
```

Very advanced: debug patter recognition

This is only for reference (and for brave people!). We won't go over it at the training day.

Checkout the RecoTracker/CkfPattern and RecoTracker/MeasurementDet packages and compile with the debug option:

scramv1 b -j 8 USER_CXXFLAGS="-DEDM_ML_DEBUG"

Enable the following categories for the message logger:

"TkStripMeasurementDet","TkPixelMeasurementDet", "CkfPattern","CkfTrackCandidateMakerBase"

Also, you may want to uncomment the following line in RecoTracker/CkfPattern/src/TrajectorySegmentBuilder.cc:

//#define DBG_TSB

In this way, from a single seed you can follow the evolution of the combinatorial pattern recognition.

Thank you for attending the 1st Tracking Training Day!

Your feedback is very valuable to help making the edition (even) better!