Table of Contents

Refitting Tracks With Constraints

Complete:

Goals of this page:

This page is intended to familiarize you with the possibility to refit tracks with additional constraints. In particular, you will learn:

- the basic ideas behind refitting with contraints
- how to set the constraints on a track by track basis
- how to run the refitter with the constraints
- features introduced to implement constraints

Introduction

The possibility to refit a track using additional constraints is very important in many situations: for example it can be used to set the momentum of the track in such cases when the refitted hits are too few to properly determine it. Other application could be a beam-spot or generic vertex constraint.
An important feature of this implementation of the refitter with contraints is that it does not simply add the constraint at the end of the fit, but it takes it into account as an additional *ordinary* hit: in this way all the measurement points are affected by the presence of the *constraining hit*.
A typical use case for refitting with constraints is represented by the alignment of the tracker.

Set up your Environment

Simply check out the release (CMSSW_1_6_0 or later). For full instruction look at WorkBookSetComputerNode.

How it works

The main idea is that the constraint is a new kind of hit with user defined values of parameters and associated errors. At the moment there are 2 different kind of constraints that could be implemented:

1. the momentum constraint (magnitude **NOT** direction)
2. the vertex constraint

The end user has to produce an association(-map) between the track to be refitted and the constraint to be applied. This approach is optimal because it provides the necessary flexibility.

The TrackRefitter has new methods that treat the newly added hit in the proper way. (For detailed explanation of the refitting process please take a look here.

How to Set Constraints

Two easy examples on how to implement refitting with constraints are located in the *test* directory of the *RecoTracker/TrackProducer* package. They are:

1. MomentumConstraintProducer.cc
2. VertexConstraintProducer.cc

The track-constraint association is performed in the `produce` method. For example it can be set as:

```
for (reco::TrackCollection::const_iterator i=theTCollection->begin(); i!=theTCollection->end();
  MomentumConstraint tmp(10.,0.01) ;
  pairs->push_back(tmp);
  output->insert(reco::TrackRef(theTCollection,index),edm::Ref<std::vector<MomentumConstraint>
  index++;
}

iEvent.put(pairs);
iEvent.put(output);
```

The code above sets a momentum constraint (the same for all tracks in the collection) of 10 GeV/c magnitude and a corresponding 0.01 GeV/c error. This is an extremely simple example of track-momentum constraint association: the code can be fully customized to your specific needs.

Since CMSSW_3_1_0_pre6, the momentum constraint and the error can be set in the configuration file (check `RecoTracker/TrackProducer/python/MomentumConstraintProducer_cff.py`). If they are not set, the MomentumConstraintProducer.cc sets the constraint to the track momentum value. For other use cases, the user can edit the MomentumConstraintProducer.cc according to his needs.

The main difference with respect to the momentum constraint are illustrated in the following code:

```
for (reco::TrackCollection::const_iterator i=theTCollection->begin(); i!=theTCollection->end();
  VertexConstraint tmp(GlobalPoint(0,0,0),GlobalError(0.01,0,0.01,0,0,0.001));
  pairs->push_back(tmp);
  output->insert(reco::TrackRef(theTCollection,index), edm::Ref<std::vector<VertexConstraint> >
  index++;
}
```

1. The user has to build a VertexConstraint using a GlobalPoint and the associated GlobalError
2. The GlobalPoint represents the vertex position in the global CMS reference frame
3. The GlobalError is the covariance matrix (the values used in GlobalError are squared)

Please remember to recompile the code before submitting the job.

Running the Refitter with Constraints

A new configuration parameter (*string constraint*) has been added to the TrackRefitter module (in file RecoTracker/TrackProducer/python/TrackRefitter_cfi.py) to set the appropriate constraint to use. The TrackRefitter class has an instance of the TrackProducerAlgorithm: the different values of this parameter control which method of the algorithm must be called. The meaning of the new parameter are illustrated in the following table:

Value	Meaning	TrackProducerAlgorithm Method Called
""	no constraint	`runWithTrack`
"momentum"	momentum constraint	`runWithMomentum`
"vertex"	vertex constraint	`runWithVertex`

To use the TrackRefitter with constraint the end user has to:

1. set the appropriate value for the *string constraint* parameter:

```
process.TrackRefitter.constraint = "momentum"
```

2. add the appropriate constraint producer module to the configuration file:

```
process.doConstraint  = cms.EDProducer("MomentumConstraintProducer")
```

or

```
process.doConstraint  = cms.EDProducer("VertexConstraintProducer")
```

3. set the appropriate label of the input source for the TrackRefitter:

```
process.TrackRefitter.src = "doConstraint"
```

4. add the constraint producer module to the path:

```
process.p1 = cms.Path(process.doConstraint * process.TrackRefitter}
```

Advanced Features

The implementation of the constraints required some architectural changes to some CMSSW packages. The main changes are listed below.

Seed Direction Information in TrackExtra

In order to perform the refitting of a track in the same condition used during its original final fit a new private member has been added to the TrackExtra class, namely `seedDir_`. The public method to get it is `PropagationDirection seedDirection() const`.
The simple addition of a private member does not spoil the backward compatibility with previous releases. Before performing any kind of refitting (constrained or not), the hits are assumed to be ordered according to the `seedDir` direction. Moreover, the initial state for refitting is chosen between `innerStateFromTrack` or `outerStateFromTrack` according to their distance from the first hit: the closets is selected.
For more details look at `TrackProducerAlgorithm::getInitialState`.

Changes to TTRHit

The TransientTrackingRecHit is a non persistent version of TrackingRecHit that provides additional features like the globalPosition and the globalPositionError of the hit. Up until now these information were built starting from the GeomDet associated to the hit: the Surface of the Det is used for the conversion from local to global coordinates. All the valid TTRHits had a valid instance of a GeomDet; invalid TTRHits could be of different types:

- with a valid GeomDet when the invalid hit was created on the Surface of a detector;
- without a valid GeomDet when created on the Surface of a non-sensitive object.

The need to build a constraining hit not belonging to any particular detector forced a change in the TTRHit implementation. The new implementation is such that a hit could be created **without** a GeomDet but with a Surface. In this way the conversion from local to global is always possible. To accomplish this a new method (`Surface * surface`) has been added to all TTRHits: if GeomDet is available it returns its surface, otherwise it returns the Surface used to create the hit.
Moreover invalid hits without a valid GeomDet can be now created using a DetLayer that provides the Surface where the hit is created.

The current implementation is such that the invalid hit created using a DetLayer is correctly build during the pattern recognition (via `LayerMeasuremens` class) but when it is made persistent the DetLayer information is lost and is not restored when the transient hit is rebuild before the final fit. Additional information has to be stored in order to recover the lost info. This could be useful also for taking into account the passive material during the final fit.

New kind of hits

The previously explained changes to the TTRhits have produced two new classes:

- TRecHit1DMomConstraint
- TRecHit2DPosConstraint

These classes are implemented in `RecoTracker/TransientTrackingRecHit`. These are the hits used to apply the constraints. They are transient with no persistent counterpart.
These hits have the following properties:

- are always valid,
- have a fake DetId,
- the method `canImproveWithTrack` returns always `false`
- contains the `projectionMatrix` to properly propagate the contraints to the trajectory parameters.

Refitting tracks using the track error matrix

Another method for track refitting is to use just the track error matrix. This is described in https://twiki.cern.ch/twiki/bin/view/CMS/SWGuideVertexFitTrackRefit#Single_Track_vertex_constraint. This latter method can be used on Analysis Object Data (AOD), and is presumably faster, but it is not quite so accurate

Applying a constraint with the algorithm and refitting

For Z to mu mu events, some recipes allow to apply constraints (momentum, vertex or trajectory state) with the help of a devoted algorithm called TwoBodyDecay. These recipes are described in https://twiki.cern.ch/twiki/bin/view/CMSPublic/ReffiterWithTwobodydecayConstraint.

Review status

Reviewer/Editor and Date	Comments
GiuseppeCerati - 29 Apr 2009	Documentation Review
MarcoRovere, GiuseppeCerati - 20 Jul 2007	created page

Responsible: MarcoRovere, GiuseppeCerati
Last reviewed by: *Never reviewed*

This topic: CMSPublic > SWGuideRefitterWithConstraints
Topic revision: r17 - 2012-02-14 - EricConte