Table of Contents

7.3.2 Vertex Fitting

Complete: 🟩

Contents

Introduction

The task of Vertex Fitting, given a set of tracks, is to compute the best estimate of the vertex parameters (position, covariance matrix, constrained track parameters and their covariances) as well as indicators of the success of the fit (total chi^2, number of degrees of freedom, track weights).

All the fitting algorithms are implemented in VertexFitter classes, which control all the steps of the vertex fit from the input of the initial information to the output of the estimated quantities. These algorithms are used in other algorithms (vertex finders, b-taggers, etc.), and can be used in an analysis by themselves. They have to be invoked by the user, and take as a minumum a selected set of tracks as input. As such, a VertexFitter does not produce data which it puts back into the event, since they are not stand-alone EDAnalyzer or EDProducer All fitters work in the same way, with the exception of the constructor, since each concrete algorithm can take different parameters as input.

As an example, we will use the KalmanVertexFitter (KVF). It is a fitter that uses the Kalman filter formalism.

Setup for tutorial

In the `src` directory of your area do:

```
addpkg RecoVertex/KalmanVertexFit
cd RecoVertex/KalmanVertexFit
scramv1 b
```

An example EDAnalyzer is given in the `test` directory: KVFTest.cc (and KVFTest.h), with the configuration file analyzeKVF_cfg.py. To see how to use a fitter, look into **`KVFTest.cc`**. A few explanations are given below.

Usage of a VertexFitter

To use a fitter, simply instantiate one and request the vertex with a set of `TransientTracks`:

```
KalmanVertexFitter fitter;
TransientVertex myVertex = fitter.vertex(vectorOfTransientTracks);
```

If you want to change the parameters of the fitter, set them with the appropriate set method (different fitters will obviously have different parameters):

- Convergence criterion (maximum transverse distance between vertex computed in the previous and the current iterations): **`setMaximumDistance (float maxShift)`**
- Maximum number of iterations: **`setMaximumNumberOfIterations(int maxIterations)`**

The vertex returned may not be valid in some cases. The condition might be different for each fitter. For the KalmanVertexFitter, as for most other fitters, an invalid vertex is returned when the maximum number of iterations is exceeded or the fitted position is out of the tracker bounds. An error message is printed in the log, which for the two above-mentionned cases is:

```
The maximum number of steps has been exceeded. Returned vertex is invalid.
Fitted position is out of tracker bounds. Returned vertex is invalid.
```

and the user has to check the validity of the vertex with the method **`isValid()`**.

Usage of a TransientTracks object

(More detailed information about TransientTracks be found here).

The tracks which are used for vertex reconstruction and for b/tau tagging are the `TransientTracks`. Being constructed from a `reco::Track`, they have access to all the data from it and provide methods which can not be provided by the `reco::Track`.

While the `reco::Track` provides only the perigee parameters at the point of closest approach to the nominal vertex (0.,0.,0.), the TransientTracks can provide states at any point along its trajectory. Having access to the magnetic field, it allows you to propagate the track through the various propagators provided in `TrackingTools/GeomPropagators`. Several other tools, such IPextrapolators are meant to be used with the TransientTracks.

If you have a collection of `reco::Track`, you must convert them first to a collection of `TransientTracks`:

```
#include "TrackingTools/TransientTrack/interface/TransientTrackBuilder.h"
#include "TrackingTools/Records/interface/TransientTrackRecord.h"

    // get RECO tracks from the event
    edm::Handle<reco::TrackCollection> tks;
    iEvent.getByLabel(trackLabel(), tks);

    //get the builder:
    edm::ESHandle<TransientTrackBuilder> theB;
    iSetup.get<TransientTrackRecord>().get("TransientTrackBuilder",theB);
    //do the conversion:
    vector<TransientTrack> t_tks = (*theB).build(tks);
```

The SimpleVertexTree

The classes SimpleVertexTree is provided to produce a simple TTree to analyse the results of a fitter. It compares the reconstructed vertex with a simulated vertex.

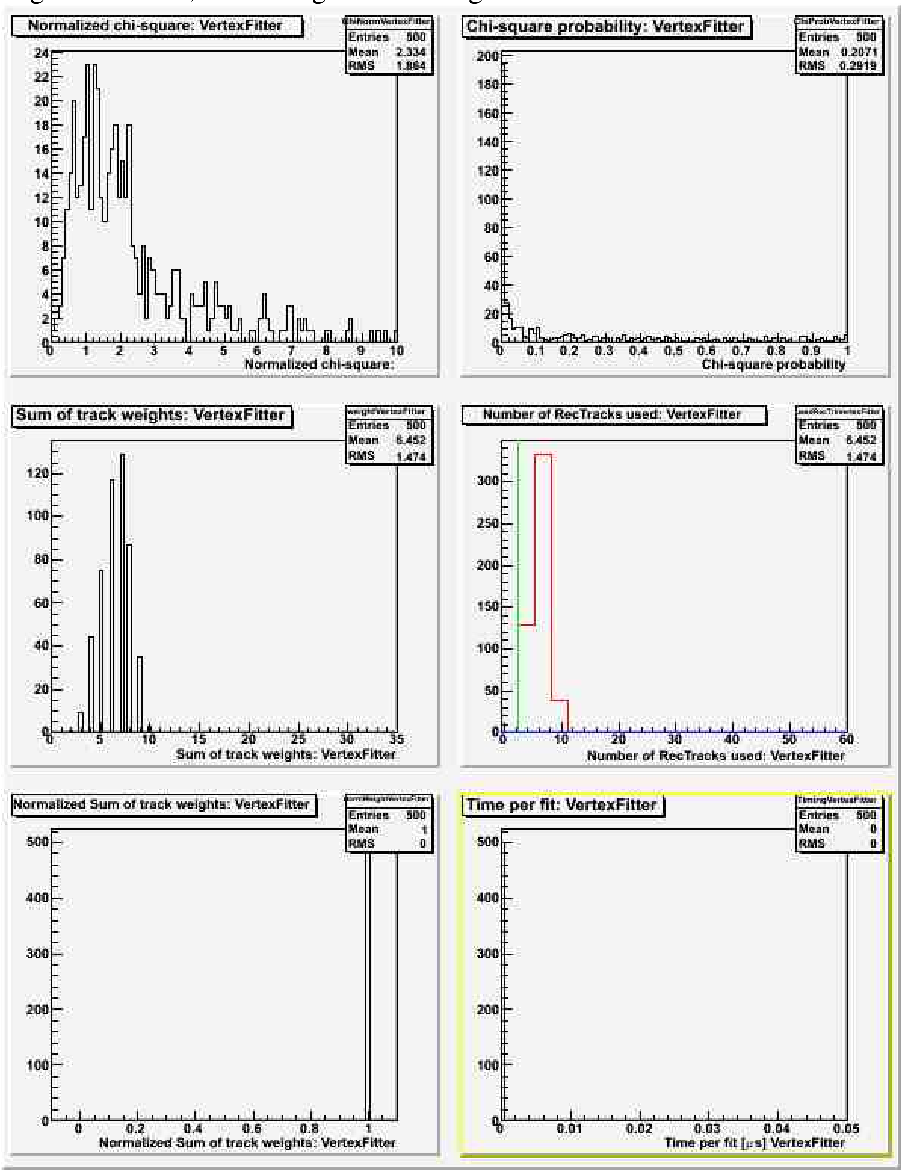
Example

Running now **`analyzeKVF.cfg`**, you will see for each event the number of tracks found and the position of the fitted vertex. At the end of the job, some simple statistics are printed. The tree produced by SimpleVertexTree is stored in the root file `simpleVertexAnalyzer.root` In a root session, look at the output with the following:

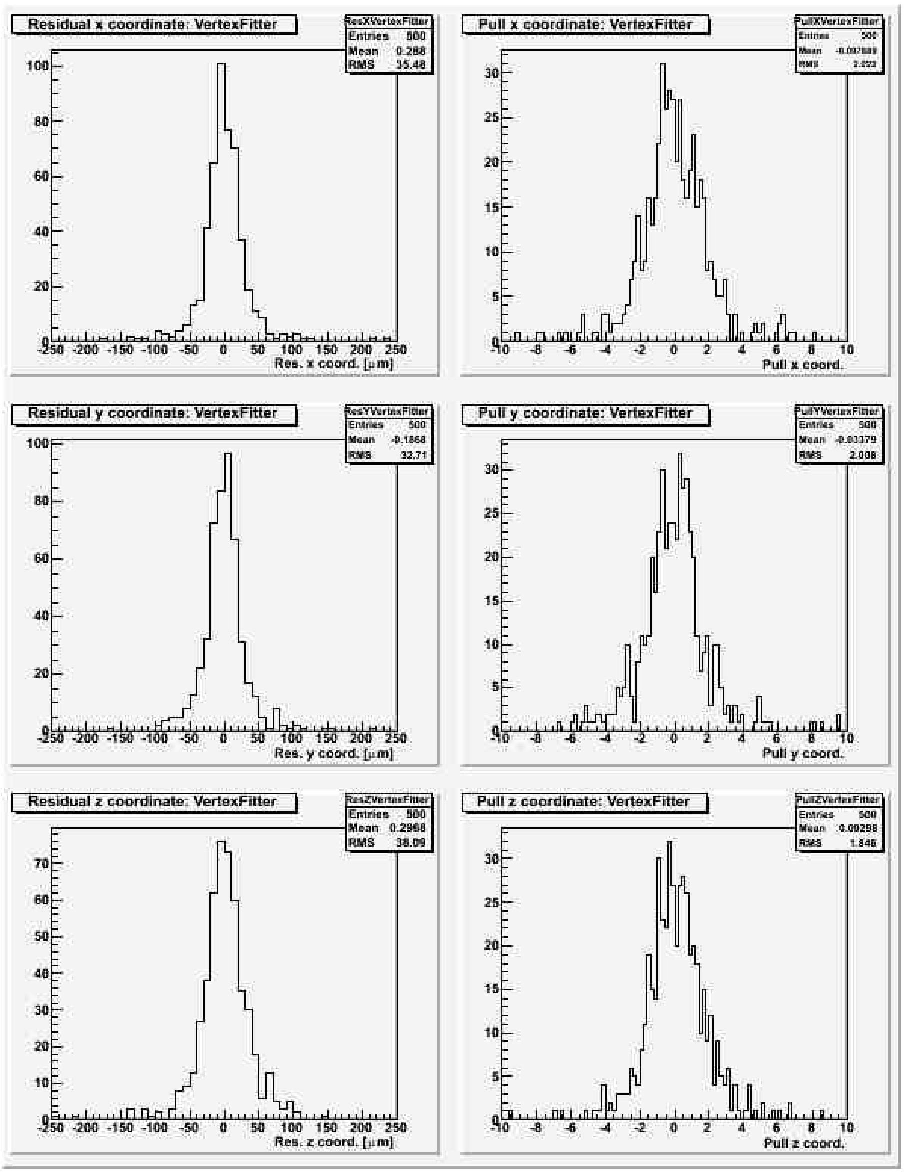
```
.L SimpleVertexAnalysis.C++
t=new SimpleVertexAnalysis("simpleVertexAnalyzer.root")
t->vertexLoop()
t->plotVertexResult()
```

You will see two canvases with the following histograms:

- Page with chi^2, track weight and timing distributions:



- Page with residual (left) and pull (right) distributions:



Review status

Reviewer/Editor and Date (copy from screen)	Comments
JennyWilliams - 05 Dec 2006	tidied, added completeness bar, responsible/review fields.
JennyWilliams - 27 Mar 2007	Edited to move vertexing links into swguide

Responsible: ThomasSpeer
Last reviewed by: ThomasSpeer - 19 Feb 2008

This topic: CMSPublic > WorkBookVertexFittingTutorial
Topic revision: r31 - 2010-08-06 - unknown