Table of Contents

The Velo Cluster and Track Monitor packages

Latest update on 19 July 2006 by Aras Papadelis

Introduction

Material from the Software training day on 27/6-2006 can be found here.

The Velo cluster and track monitor packages consists of VeloTrackDataMonitor and VeloClusterDataMonitor. They are not real time monitoring algorithms but require input from a data file.

- VeloClusterDataMonitor produces histograms from the zero-suppressed cluster bank.
- VeloTrackDataMonitor produces histograms using Tracks from the Velo pattern recognition.

Since the packages use the zero-suppressed buffer they are not suited for studying noise levels, pedestals, common mode correction, FIR corrections etc. For that kind of monitoring the package VeloDataFullMonitor is needed.
These packages are primarily intended for use during the VELO testbeam and commissioning phase.

Installation

NB, with the release of Vetra v2r1 there is no longer any need for installing the monitoring packages on your home account, unless you want to add your own hacks to the monitoring code. If you want to run the data monitoring from Brunel you still need to follow these instructions:

For now, let's assume that we are running Vetra. The installation procedure is identical for the two packages. Let's install VeloClustersDataMonitor:

1. Make sure that you have Vetra installed.
2. Go to your cmt directory (probably ~/cmtuser)
   `> cd ~/cmtuser`
3. Set the Vetra environment
   `> VetraEnv` (choose the latest version, or whatever version you prefer)
4. Get the latest version from CVS:
   `> getpack Velo/VeloClusterDataMonitor` (select the head version for the latest)
5. Compile your new package:
   ```
   > cd Velo/VeloClusterDataMonitor/v*/cmt
   > cmt config
   > source setup.csh
   > make
   ```
6. If everything is OK, the program should now compile without complaints. You have installed VeloClusterDataMonitor! Now repeat this procedure if you want to install VeloTrackDataMonitor.

Running the monitoring...

...in Brunel

OK, now we want to run the algorithm from somewhere, for example Brunel or Vetra. **If you are running Vetra you can skip this section**. Here I will show how to run the monitoring from Brunel. I assume that you have Brunel installed.

1. Make sure that the Brunel environment is set using `BrunelEnv`.
2. Edit the Brunel requirements file:
   ```
   > cd $BRUNELROOT/cmt
   > emacs requirements
   ```
3. Add the line
   `use VeloClusterDataMonitor v* Velo -no_auto_imports` (add a second line with VeloTrackDataMonitor if you want to install that too)
4. Save the file, exit the editor and type
   `> cmt config > source setup.csh` (if you don't get any error messages here that means you're doing just fine)
   `> make`
5. Brunel will now recompile with VeloClusterDataMonitor (and VeloTrackDataMonitor if you added it in step 3)
6. If everything works you can now put this https://uimon.cern.ch/twiki/pub/LHCb/VeloDataMonitor/VeloDataMonitor.opts monitoring options file in your options directory, and include it from your main options file.

You should now be able to run VeloDataMonitor from Brunel!

...in Vetra

If you are using Vetra v2r2 or higher running the high level monitoring should be a piece of cake! The default Zero Suppressed data (ZS) options files that come with Vetra take care of most of the stuff for you. I've prepared a special page for how to run Vetra over ZS data here

Trouble shooting

One of the common errors that you may get is that Vetra / Brunel reports: `ERROR ---- Track Container retrieved but EMPTY ---`

This usually means two things:

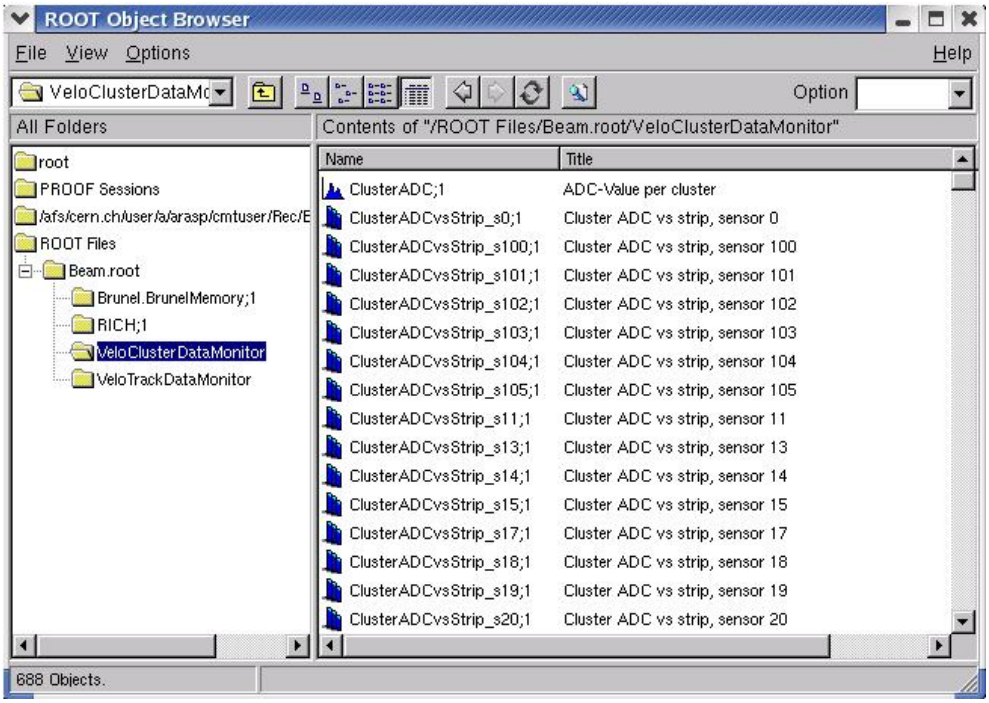
1. No tracks can be found because the sensors have not been mapped properly. Read more about this sensor remapping here
2. You are using the wrong pattern recognition. For simulated beam-beam collision data files you need to use the "LHCb Pattern Recognition" (PatVeloRTracking and PatVeloSpaceTracking) and for ACDC simulated and real data you most likely need to use the generic pattern recognition (PatVeloGeneric). From Vetra this can be set by choosing which of the two tracking options files that should be included: VetraTracking.opts or VetraGenericTracking.opts.

Using the packages

The VeloDataMonitor packages will produce a lot of histograms. If you are running with a full VELO detector, expect at least a couple of hundred of them. If you are interested in studying the data on an event by event basis the cluster information can be written to a ntuple by simply setting an option in the VeloClusterDataMonitor options file (*No ntuple output is implemented for the track information at this point but if anyone wants it I can add it easily /Aras P*).

1. Let's start by simply running Vetra with the monitoring packages and the default settings. At the end of the run, Brunel/Vetra will produce a root file. Open it.
2. Open a TBrowser (with `TBrowser a` for example) and open the file. If everything worked properly you should see the directories VeloClusterDataMonitor and VeloTrackDataMonitor in the file. If you don't have VeloTrackDataMonitor in the file it is probably because no tracks where found in the run.

If you turn on the "Detailed" view you will see a description to each histogram. The histogram descriptions are hopefully self explanatory. Here's an example:



There are many, many histograms here... too many to cover here. One that can be handy to know about is `nEvents`, which contains as many entries as the number of events that were processed.

A brief root example

It is not the intention of this article to explain root, but here's a very short beginners example of how to find out what the cluster size distribution looks like for a certain sensor, without using the browser.

Start root with the file (in this example Beam.root):

```
> root Beam.root
```

When root starts there will be a pointer created for you that points to the open file. It's default name is `_file0`. Now we have to go to the VeloClusterDataMonitor directory:

```
> _file0.cd("/VeloClusterDataMonitor")
```

The histogram ClusterSizeVsSensor contains what we need. You can find it with the `.ls` command:

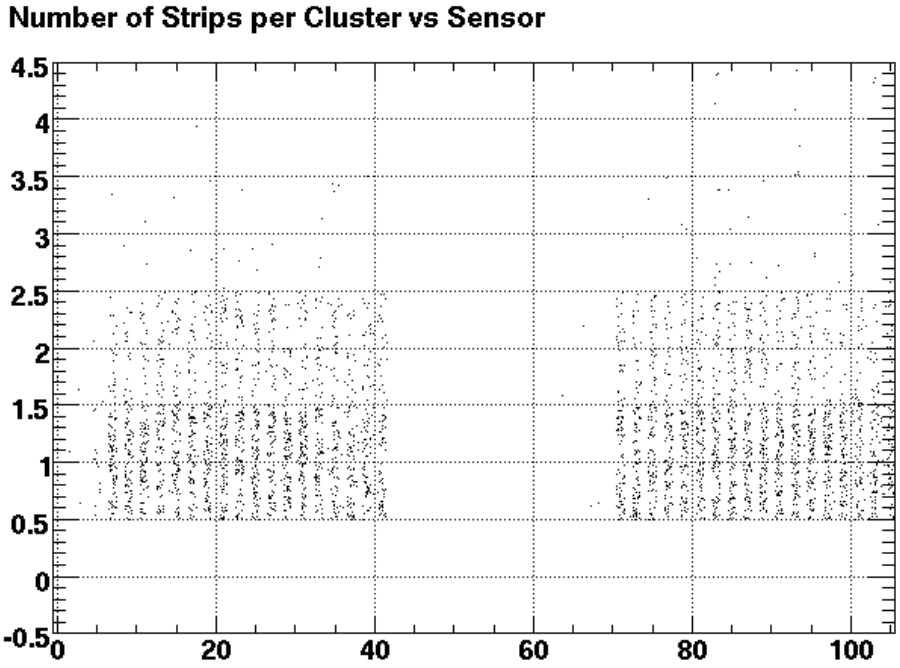
```
> .ls *Size*
```

You will now see something like:

```
TDirectory*     VeloClusterDataMonitor  VeloClusterDataMonitor
 KEY: TH1D      ClusterSize;1   Number of Strips per Cluster
 KEY: TH2D      ClusterSizeVsSensor;1   Number of Strips per Cluster vs Sensor
```

Try to draw ClusterSizeVsSensor:

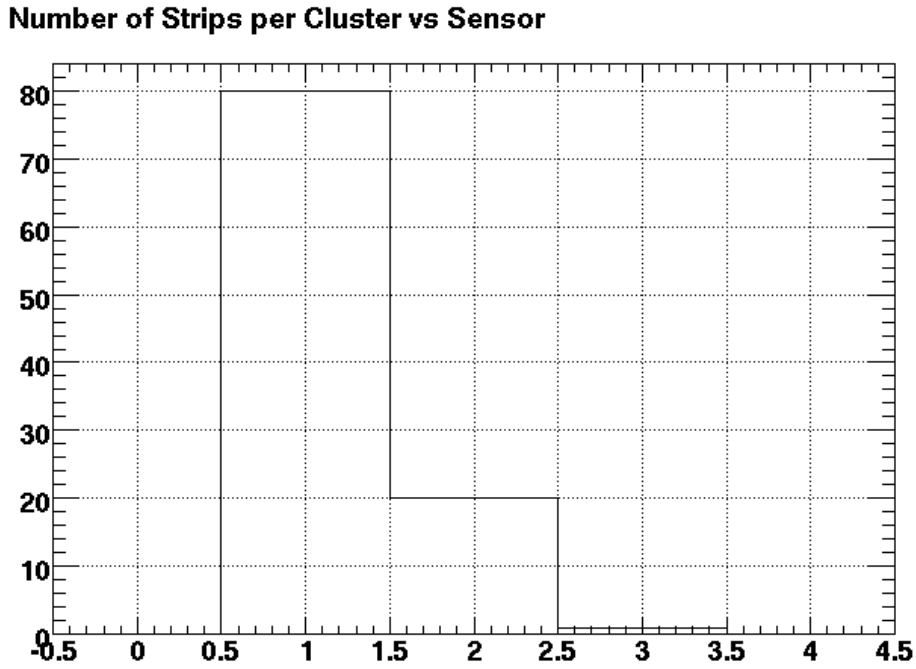
```
> ClusterSizeVsSensor.Draw()
```



As you see, this is a 2D histogram where the cluster size is plotted verses the sensor number. To see the cluster size distribution for sensor 7 we need to project the contents of bin 8 (sensor number + 1 since root counts the bins starting from 1) on y-axis:

```
> ClusterSizeVsSensor.ProjectionY("proj",8,8)

> proj.Draw()
```



From this we learn that a majority of the clusters in sensor 7 consist of 1 strip. Actually, the average cluster size can be obtained with:

```
 > proj.GetMean()
```

If you want to have a look in the histograms created by the track monitor you can change directory again with:

```
> file0.cd("/VeloTrackDataMonitor")
```

This concludes this little tutorial.

The options files

By changing the options files you can control the output of VeloDataMonitor. By turning off the output of some histograms and the ntuple you can gain speed. In **VeloClusterDataMonitor.opts** you can change the following:

`VeloClusterDataMonitor.OutputNtuple` Turns on/off the output of a ntuple
`VeloClusterDataMonitor.OccupancyHistos` Turns on/off the average clusters occupancy histos (one histogram per sensor)
`VeloClusterDataMonitor.MultiplicityHistos` Turns on/off the average cluster multiplicity histos (one histogram per sensor)

In **VeloTrackDataMonitor.opts** you can change the following

`VeloTrackDataMonitor.EventDisplay` Turns on/off the 3D "event display" histos and the XY hitmap histos.
`VeloTrackDataMonitor.SelectedEvent` 1= Selects which event will be plotted in the "selected" 3D event display histogram.
`VeloTrackDataMonitor.ExtrapResidHistos` Turns on/off a lot of histograms dealing with residuals between track sensor intercept points and clusters locations.
`VeloTrackDataMonitor.TrackLocation ="Rec/Track/Velo"` Sets the location of the track container, in case you want to use a different track set for any reason.

Additionally, you need to add the following option when you are using PatVeloFilterClusters to exlude sensors from the pattern recognition (beam telescope mode):

`VeloTrackDataMonitor.TestSensors ={sensor1, sensor2, ... , sensorN}` (where sensorX is an integer corresponding to the sensor number that was excluded by PatVeloFilterClusters).

Adding your own histograms and hacking the code

If you aren't happy with the standard set of histograms it's very easy for you to add your own custom histograms or custom hacks to the monitoring source code. If we use VeloClusterDataMonitor as an example, you will find the source code in the `src` directory of `Velo/VeloClusterDataMonitor`.

The `VeloClusterDataMonitor` class is derived from the GaudiTupleAlg class, which provides an interface for creating histograms and ntuples. Here's how to fill a histogram:

```
plot1D(value, histogram name, histogram title, xlow, xhigh, xbins, weight)

plot2D(x-value, y-value, histgram name, histogram title, xlow, xhigh, ylow, yhigh, xbins, ybins)
```

and a 3D histogram using the LHCb coordinate system:

```
plot3D(z-value,x-value,y-value, ...fill in yourself...)
```

The histogram name can be an `int` or a `std::string` while the histogram title should be a std::string that describes the contents of the histogram in a sentence. There are plenty of examples of how this is done in the source files.

It might be a good idea to separate your own histograms from the rest of the code. In that way it will be much easier for you to migrate to new version of the package. Depending on which version of the monitoring packages you use, there are different ways of doing this:

- VeloClusterDataMonitor and VeloTrackDataMonitor **v1r3**: A method called userHistos() is available in the class .cpp file. There is an option called UserHistos with which you **MUST** set to **true** in order for this method to be executed.

- VeloClusterDataMonitor and VeloTrackDataMonitor **v1r4 or higher**: In the src directory you will find files for the class called Cluster(Track)DataMoniUserHistos. This class is derived from VeloCluster(Track)DataMonitor and has knowledge about the track and cluster containers. Add your own hacks to this class and **don't forget** to add the algorithm to the run sequence. Here is an example from **Vetra** where you can see how the algorithm `TrackDataMoniUserHistos` has been added to the run sequence:

```
ApplicationMgr.DLLs += { "VeloTrackDataMonitor"  };
ClusterAndTrackMoni.Members += { "VeloTrackDataMonitor"
                                ,"TrackDataMoniUserHistos"
};
```

Useful to know

To plot information about the clusters and tracks you need to familiarise yourself with classes like `DeVelo`, `DeVeloSensor`, `VeloCluster`, `Track` and `VeloChannelID`. Here's some documentation from the CVS repository and the Doxygen pages (*warning, this link may be pointing to an old version of LHCb*).

-- Aras Papadelis - 19 July 2006

This topic: LHCb > VeloDataMonitor
Topic revision: r11 - 2008-02-06 - ArasPapadelis