A tutorial for NA62Analysis: creating the VertexCDA and Pi0Reconstruction analyzers.

December 10, 2013

This document will describe the process of creating the VertexCDA and Pi0Reconstruction analyzers within the NA62Analysis framework. It is intended to guide future analyzer authors by describing the complete procedure. Assumption is done that the environment is already configured, NA62Analysis is installed and the User directory has been created. The installation and configuration procedure is described at `http://sergiant.web.cern.ch/sergiant/NA62FW/html/analysis.html`. Before starting the framework environment should be set by sourcing the env.(c)sh file in the **scripts/** folder of the user directory. Every shell command given in this document is to be executed from the top of the user directory.

1 VertexCDA

The aim of this analyzer is to implement an algorithm that will compute the Kaon decay vertex and make it available to further analyzers. Though different methods are available for this purpose, the focus is given on a closest distance of approach (CDA) algorithm. The algoritm implemented by this analyzer can be used for the class of processes $K^\pm \to C^\pm X$ where $K^\pm$ is the incoming charged kaon measured in GigaTracker, $C^\pm$ is a charged particle measured in the Spectrometer and X is any kind and any number of additional particule. Even if the tracks are supposed to originate from the same point they are in practice never intersecting because of the finite measurement precision. The CDA will find the unique point $\vec{v} = (v_x, v_y, v_z)$ where the distance between v and the first track and the distance between v and the second track are minimums. For this point only, the line l passing through v and joining both tracks is perpendicular to them.

We define $\vec{x}_{1,2}$ being the track origin and $\vec{p}_{1,2}$ it's momentum. Any point $\vec{A}_{1,2}$ on track can be described by a single parameter $s_{1,2}$.

$$\vec{A}_{1,2} = \vec{x}_{1,2} + s_{1,2}\vec{p}_{1,2} \tag{1}$$

The vector joining $\vec{A}_1$ and $\vec{A}_2$ is then

$$\begin{aligned} \vec{l}(s_1, s_2) = \vec{A}_1 - \vec{A}_2 &= \vec{x}_1 + s_1\vec{p}_1 - \vec{x}_2 - s_2\vec{p}_2 & (2)\\ &= \vec{x}_1 - \vec{x}_2 + s_1\vec{p}_1 - s_2\vec{p}_2 & (3)\\ &= \vec{l}_0 + s_1\vec{p}_1 - s_2\vec{p}_2 & (4) \end{aligned}$$

with $\vec{l}_0 = \vec{x}_1 - \vec{x}_2$. Using the property that $\vec{l}$ is perpendicular to $\vec{p}_1$ and $\vec{p}_2$ when it's length is minimum gives the following equations:

$$\begin{aligned} \vec{l}(s_{1c}, s_{2c}) \cdot \vec{p}_1 = \vec{l}_0 \cdot \vec{p}_1 + s_{1c}|\vec{p}_1|^2 - s_{2c}(\vec{p}_1 \cdot \vec{p}_2) &= 0 & (5)\\ \vec{l}(s_{1c}, s_{2c}) \cdot \vec{p}_2 = \vec{l}_0 \cdot \vec{p}_2 + s_{1c}(\vec{p}_2 \cdot \vec{p}_1) - s_{2c}|\vec{p}_2|^2 &= 0 & (6) \end{aligned}$$

That we solve for s_{1c} and s_{2c}:

$$s_{1c} = \frac{(\vec{p}_1 \cdot \vec{p}_2)(\vec{l}_0 \cdot \vec{p}_2) - (\vec{l}_0 \cdot \vec{p}_1)|\vec{p}_2|^2}{|\vec{p}_1|^2|\vec{p}_2|^2 - (\vec{p}_1 \cdot \vec{p}_2)^2} \tag{7}$$

$$s_{2c} = \frac{|\vec{p}_1|^2(\vec{l}_0 \cdot \vec{p}_2) - (\vec{p}_1 \cdot \vec{p}_2)(\vec{l}_0 \cdot \vec{p}_1)}{|\vec{p}_1|^2|\vec{p}_2|^2 - (\vec{p}_1 \cdot \vec{p}_2)^2} \tag{8}$$

The vertex being in the middle of the $\vec{l}(s_{1c}, s_{2c})$ vector:

$$v = \vec{A}_1 + 0.5\vec{l}(s_{1c}, s_{2c}) \tag{9}$$

1.1 VertexCDA analyzer implementation

The first step is to create the skeleton of the new analyzer. This is automatically done with the framework python script:

```
NA62AnalysisBuilder.py new VertexCDA
```

We can open the source code of the newly created analyzer in **Analyzers/include/VertexCDA.hh** and **Analyzers/src/VetexCDA.cc**, go through every standard method of the analyzer and describe each section of code

1.1.1 Constructor

As we will need to access the GigaTracker and Spectrometer information we first start by requesting the access to these ROOT TTrees:

```
//Request TTree with name GigaTracker with elements of class TRecoGigaTrackerEvent
RequestTree("GigaTracker", new TRecoGigaTrackerEvent);
//Request TTree with name Spectrometer with elements of class TRecoSpectrometerEvent
RequestTree("Spectrometer", new TRecoSpectrometerEvent);
```

Without forgetting to include the relevant header files

```
#include "TRecoGigaTrackerEvent.hh"
#include "TRecoSpectrometerEvent.hh"
```

1.1.2 InitHist

In this method we will create and register all the histograms that we will need during the processing.

```
//Booking 3 histograms to plot the reconstructed vertex x,y,z positions
//Book Histogram VertexX (name that will be used to access this histogram later) and
//create it with new TH1I (see ROOT TH1I for the syntax).
BookHisto("VertexX",
  new TH1I("VertexX", "Reconstructed vertex X position", 250, -250, 250));
BookHisto("VertexY",
  new TH1I("VertexY", "Reconstructed vertex Y position", 150, -150, 150));
BookHisto("VertexZ",
  new TH1I("VertexZ", "Reconstructed vertex Z position", 100, 0, 300000));

//Booking 3 histograms to plot the difference between the reconstructed vertex and
     the real vertex components
BookHisto("DiffVertexX",
  new TH1I("DiffVertexX", "X difference between reco and real vertex", 200, -50, 50)
     );
BookHisto("DiffVertexY",
  new TH1I("DiffVertexY", "Y difference between reco and real vertex", 200, -50, 50)
     );
BookHisto("DiffVertexZ",
  new TH1I("DiffVertexZ", "Z difference between reco and real vertex", 200, -10000,
     10000));

//Booking 3 2D histograms to plot the reconstructed vertex components vs. the real
     vertex components
BookHisto("VertexRecoRealX",
  new TH2I("VertexRecoRealX", "Reconstructed vs. Real (X)", 250, -250, 250, 250,
     -250, 250));
BookHisto("VertexRecoRealY",
  new TH2I("VertexRecoRealY", "Reconstructed vs. Real (Y)", 150, -150, 150, 150,
     -150, 150));
BookHisto("VertexRecoRealZ",
  new TH2I("VertexRecoRealZ", "Reconstructed vs. Real (Z)", 200, 0, 300000, 200, 0,
     300000));

//Booking histograms to plot the reconstructed candidates multiplicity in
     GigaTracker and Spectrometer
BookHisto("GTKMultiplicity",
  new TH1I("GTKMultiplicity", "Multiplicity in GTK", 11, -0.5, 10.5));
BookHisto("StrawMultiplicity",
  new TH1I("StrawMultiplicity", "Multiplicity in Straw", 11, -0.5, 10.5));

//Booking a serie of 20 histogram to plot the MC kaon decay vertex X-Y profile in
     different sections
of the detector length
for(int i=0; i<20; i++){
  BookHisto(TString("BeamXY") + (Long_t)i,
    new TH2I(TString("BeamXY") + (Long_t)i,
        TString("BeamXY") + (Long_t)(100+i*5) + TString("->") + (Long_t)(100+(i+1)
     *5),
        100, -100, 100, 100, -100, 100));
}
```

We can also create several counters to keep track of the number of events passing the selection for this analyzer. The counters are grouped in an **EventFraction** table and the sample size for this table is defined as the "*Total_Events*" counter.

```
BookCounter("Total_Events");
BookCounter("Good_GTK_Mult");
BookCounter("Good_Straw_Mult");

NewEventFraction("Selection");
AddCounterToEventFraction("Selection", "Total_Events");
AddCounterToEventFraction("Selection", "Good_GTK_Mult");
AddCounterToEventFraction("Selection", "Good_Straw_Mult");
DefineSampleSizeCounter("Selection", "Total_Events");
```

1.1.3 InitOutput

This analyzer is meant to provide a vertex to other analyzers. We first declare the *fVertex* member object in the header of the analyzer:

```
protected:
  TVector3 fVertex;
```

before declaring it to the framework under the name "*Vertex*" in the **InitOutput** method. It should be noted that to avoid collisions between independant analyzers, this name is automatically prepended with the name of the analyzer and a dot. In this case, to access this object from another analyzer, one will have to request "*VertexCDA.Vertex*"

```
RegisterOutput("Vertex", &fVertex);
```

1.1.4 DefineMCSimple

This method is specific to simulated events. We will define a specific partial event signature of the type $K^+ \to \pi^+ X$ where X can be any number (0 included) of any particle. When reading simulated events, the list of simulated particles (**KineParts**) will be scanned and the particles corresponding to the definition given here will be stored in a structure that will allow us to easily reference them later.

```
//We ask for the initial Kaon and store it's ID
int kID = fMCSimple->AddParticle(0, 321);
// We ask for the positive pion whose parent is the initial Kaon (kID as parent)
fMCSimple->AddParticle(kID, 211);
```

1.1.5 Process

This method is the main process loop. Every event will be processed by this method. First we declare some variables:

```
//Flags for rejected/accepted events
bool badEvent = false;
//Will contain the kaon position on GigaTracker station 3 and it's reconstructed
    momentum
TVector3 KaonPosition, KaonMomentum;
//Will contain the charged pion position returned by Spectrometer and it's
    reconstructed momentum
TVector3 PipPosition, PipMomentum;
```

This analyzer does not need simulated data but can do extra-work for consistency checks if available. We therefore don't enforce the use of simulated data but we simply check for them and set the *withMC* flag accordingly. The *fMCSimpe.fStatus* flag can take 3 different values:

- *kEmpty*: No Monte Carlo data have been found
- *kMissing*: Monte Carlo data have been found but the current event does not correspond to the signature given in **DefineMCSimple**
- *kComplete*: Monte Carlo data have been found and the event corresponds to the signature given in **DefineMCSimple**. There could be additional particles in the event.

The extra-work can only be realized with a complete event and we will therefore only check for the value *KComplete* of the status flag.

```
//Declare the flag
bool withMC = true;
//For the additional jobWe are only interested in complete
if(fMCSimple.fStatus != MCSimple::kComplete) withMC = false;
```

Then we request the events retrieved from the GigaTracker and Spectrometer TTrees

```
TRecoGigaTrackerEvent *GTKEvent = (TRecoGigaTrackerEvent*)GetEvent("GigaTracker");
TRecoSpectrometerEvent *SpectrometerEvent = (TRecoSpectrometerEvent*)GetEvent("
    Spectrometer");
```

We increment to counter that keeps track of the total number of processed events:

```
IncrementCounter("Total_Events");
```

The next step is handling the GigaTracker information. We fill the multiplicity histogram and if only 1 candidate has been reconstructed we fill the kaon variables. The kaon position is the position on station 3 but as the coordinates are relative to the center of the GigaTracker detector, we have to translate it into the experiment frame. Only the z coordinate is impacted. We also increment the counter keeping the count of events passing the GigaTracker selection. If 0 or more than 1 candidate is reconstructed, we flag the event as bad.

```
//Filling GigaTracker multiplicity histogram
FillHisto("GTKMultiplicity", GTKEvent->GetNCandidates());
if(GTKEvent->GetNCandidates()==1){
  //If only 1 reconstructed candidate, set KaonPosition to the position on station 3
      (index 2)
  KaonPosition = ((TRecoGigaTrackerCandidate*)GTKEvent->GetCandidate(0))->
      GetPosition(2);
  //Move Z to the experiment frame. Z position of GigaTracker centre is 90932.5mm
  KaonPosition.SetZ(KaonPosition.Z()+90932.5);
  //Set the kaon momentum as measured by GigaTracker
  KaonMomentum = ((TRecoGigaTrackerCandidate*)GTKEvent->GetCandidate(0))->
      GetMomentum().Vect();
  //Increment the GigaTracker selection counter
  IncrementCounter("Good_GTK_Mult");
}
else badEvent = true;
```

The same processing is applied for the recostructed Spectrometer events. We fill the multiplicity histogram then get the position and momentum if only one candidate is reconstructed and we increment the counter for the Spectrometer selection. If no or more than one event is found, the *badEvent* flag is raised.

```
FillHisto("StrawMultiplicity", SpectrometerEvent->GetNCandidates());
if(SpectrometerEvent->GetNCandidates()==1){
  PipPosition = ((TRecoSpectrometerCandidate*)SpectrometerEvent->GetCandidate(0))->
      GetPosition();
  PipMomentum = ((TRecoSpectrometerCandidate*)SpectrometerEvent->GetCandidate(0))->
      GetMomentum().Vect();
  IncrementCounter("Good_Straw_Mult");
}
else badEvent = true;
```

Finally if the event is tagged as bad, we set the state of the output to *kOInvalid* without further treatment to signal other analyzers that the value of this output should not be used.

```
if(badEvent){
  //Signal the state of the Vertex output as kOInvalid if the event did not pass the
      selection
  SetOutputState("Vertex", kOInvalid);
}
```

If the event passed the selection, we finally compute the vertex with the CDA algorithm (whose implementation is described in section 1.4.1), set the state of the output to *kOValid* to signal other analyzers that they can use it and fill the three Vertex histograms.

```
else{
  //Get the vertex from the CDA algorithm
  fVertex = GetIntersection(KaonPosition, KaonMomentum, PipPosition, PipMomentum);
  \\Signal the state of the Vertex Output as kOValid
  SetOutputState("Vertex", kOValid);

  //Fill the histograms with the vertex components
  FillHisto("VertexX", fVertex.X());
  FillHisto("VertexY", fVertex.Y());
  FillHisto("VertexZ", fVertex.Z());
```

To finish the event processing and if MonteCarlo are available, we compare the reconstructed vertex to the real one component by component and insert the result in 1D and 2D histograms. We also plot the vertex X-Y profile along the experiment in 20 Z sections starting at Z=100m.

```
  if(withMC){
    FillHisto("DiffVertexX", fVertex.X()-fMCSimple["pi+"][0]->GetProdPos().X());
    FillHisto("DiffVertexY", fVertex.Y()-fMCSimple["pi+"][0]->GetProdPos().Y());
    FillHisto("DiffVertexZ", fVertex.Z()-fMCSimple["pi+"][0]->GetProdPos().Z());
    FillHisto("VertexRecoRealX", fVertex.X(), fMCSimple["pi+"][0]->GetProdPos().X())
     ;
    FillHisto("VertexRecoRealY", fVertex.Y(), fMCSimple["pi+"][0]->GetProdPos().Y())
     ;
    FillHisto("VertexRecoRealZ", fVertex.Z(), fMCSimple["pi+"][0]->GetProdPos().Z())
     ;
    int cat = ((fMCSimple["pi+"][0]->GetProdPos().Z()/1000)-100)/5;
    FillHisto(TString("BeamXY") + (Long_t)cat, fMCSimple["pi+"][0]->GetProdPos().X()
     , fMCSimple["pi+"][0]->GetProdPos().Y());
  }
}
```

1.2 PostProcess

This method is meant to destroy temporary objects that have been allocated during **Process** and that shall be destroyed only after all the analyzers finished processing the event. This is typically the case for memory allocated for the output of the analyzer. In the specific case of this analyzer, no such memory has been allocated and this method will stay empty.

1.3 ExportPlot

All the histograms previously booked with **BookHisto** are saved in the output ROOT file with

```
SaveAllPlots();
```

1.4 DrawPlot

Similarly if the analysis is running in graphical mode, all the histograms previously booked with **BookHisto** should be displayed on screen:

```
DrawAllPlots();
```

1.4.1 GetIntersection

This is the implementation of the CDA algorithm described earlier in this document. As input it receive two 3-vectors *x1* and *x2* being respectively a point belonging to the first track and a point belonging to the second track and two 3-vector *p1* and *p2* representing the directions of the tracks.

The $\vec{l}_0$ vector is computed from the positions:

```
TVector3 l0 = x1-x2;
```

Then the different cross-products and the $s_{1,2}$ parameters are computed:

```
double a = p1.Mag2();
double b = p1*p2;
double c = p2.Mag2();
double d = p1*d0;
double e = p2*d0;
double s1 = (b*e-c*d)/(a*c-b*b);
double s2 = (a*e-b*d)/(a*c-b*b);
```

The vector linking the closest points ($\vec{l}(s_1, s_2)$ in the algorithm) is *vdist* and the point being in the middle of this vector is returned:

```
TVector3 vdist = l0 + (s1*p1 - s2*p2);

return x1 + s1*p1 - 0.5*vdist;
```