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

Nicolas Lurkin

December 13, 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 algorithm 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 particle. Even if the tracks are supposed to originate from the same point they are in practice never intersecting because of the finite measurement precision. As long as the tracks are not parallel the CDA will find the unique point $\vec{V} = (V_x, V_y, V_z)$ where the distance between $\vec{V}$ and the first track and the distance between $\vec{V}$ and the second track are minimums. For this point only, the vector $\vec{l}$ passing through $\vec{V}$ and joining both tracks is perpendicular to them.

If the track origins $\vec{x}_{1,2}$ and their momenta $\vec{p}_{1,2}$ are defined, any point $\vec{A}_{1,2}$ on track can be described by the parameter $s_{1,2}$ such as

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

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

$$\vec{l}(s_1, s_2) = \vec{A}_1 - \vec{A}_2 = \vec{x}_1 + s_1\vec{p}_1 - \vec{x}_2 - s_2\vec{p}_2 \tag{2}$$

$$= \vec{l}_0 + s_1\vec{p}_1 - s_2\vec{p}_2 \tag{3}$$

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 its length is minimum gives the following equations:

$$\vec{l}(s_{1c}, s_{2c}) \cdot \vec{p}_1 = \vec{l}_0 \cdot \vec{p}_1 + s_{1c}|\vec{p}_1|^2 - s_{2c}(\vec{p}_1 \cdot \vec{p}_2) = 0 \tag{4}$$

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

They can be solved for s_{1c} and s_{2c}:

$$s_{1c} = \frac{(\vec{p}_1 \cdot \vec{p}_2)(\vec{l}_0 \cdot \vec{p}_2) - (\vec{l}_0 \cdot \vec{p}_1)|\vec{p}_2|^2}{|\vec{p}_1|^2|\vec{p}_2|^2 - (\vec{p}_1 \cdot \vec{p}_2)^2} \tag{6}$$

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

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

$$\vec{V} = \vec{A}_1 + 0.5\vec{l}(s_{1c}, s_{2c}) \tag{8}$$

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
```

The source code of the newly created analyzer can be found in **Analyzers/include/VertexCDA.hh** and **Analyzers/src/VertexCDA.cc**. Every standard method of the analyzer and each section of code will be described thereafter.

1.1.1 Constructor

As the GigaTracker and Spectrometer information needs to be accessed, it is necessary to start by requesting 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 at the beginning of the file

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

1.1.2 InitHist

In this method all the histograms that will be needed during the processing are created and registered to the framework:

- Histograms for plotting the vertex x,y,z positions

```
//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));
```

- Histograms to compare the reconstructed vertex with the true Monte Carlo vertex

```
//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));
```

- Histograms for the particle multiplicity in detectors

```
//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));
```

- A serie of histogram to plot the kaon decay profile in 20 z sections

```
//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));
}
```

Several counters to keep track of the number of events passing the selection are created as well. 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. The *fVertex* member object is first declared 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 independent 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. A specific partial event signature of the type $K^+ \to \pi^+ X$ is defined 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 to easily referencing 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 some variable declaration:

```
//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. The use of simulated data is therefore not enforced but it is checked anyway and the *withMC* flag is set 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 therefore only events with the value *KComplete* are relevant.

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

Then the events retrieved from the GigaTracker and Spectrometer TTrees are retrieved

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

And the counter that keeps track of the total number of processed events is incremented

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

The next step is handling the GigaTracker information. The multiplicity histogram are filled, and the Kaon variables are set only if 1 candidate has been reconstructed. The Kaon position is the position on station 3 but as the coordinates are relative to the centre of the GigaTracker detector, it has to be translated into the experiment frame. Only the z coordinate is impacted. The counter keeping the count of events passing the GigaTracker selection is incremented as well. The event is flagged as bad if 0 or more than 1 candidates are reconstructed.

```
//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 reconstructed Spectrometer events. The multiplicity histogram is filled then the position and momentum are set if only one candidate is reconstructed and the counter for the Spectrometer selection is incremented. 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, the state of the output is set 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, the vertex is finally computed with the CDA algorithm (whose implementation is described in section 1.1.9), the state of the output is set to *kOValid* to signal other analyzers that they can use it, and the three Vertex histograms are filled.

```
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 Monte Carlo are available, the reconstructed vertex is compared to the real one component by component and the results are inserted in 1D and 2D histograms. The vertex X-Y profile along the experiment is also plotted 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());
  }
}
```

1.1.6 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.

1.1.7 ExportPlot

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

```
SaveAllPlots();
```

1.1.8 DrawPlot

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

```
DrawAllPlots();
```

1.1.9 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 *l0* vector is determined 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;
```

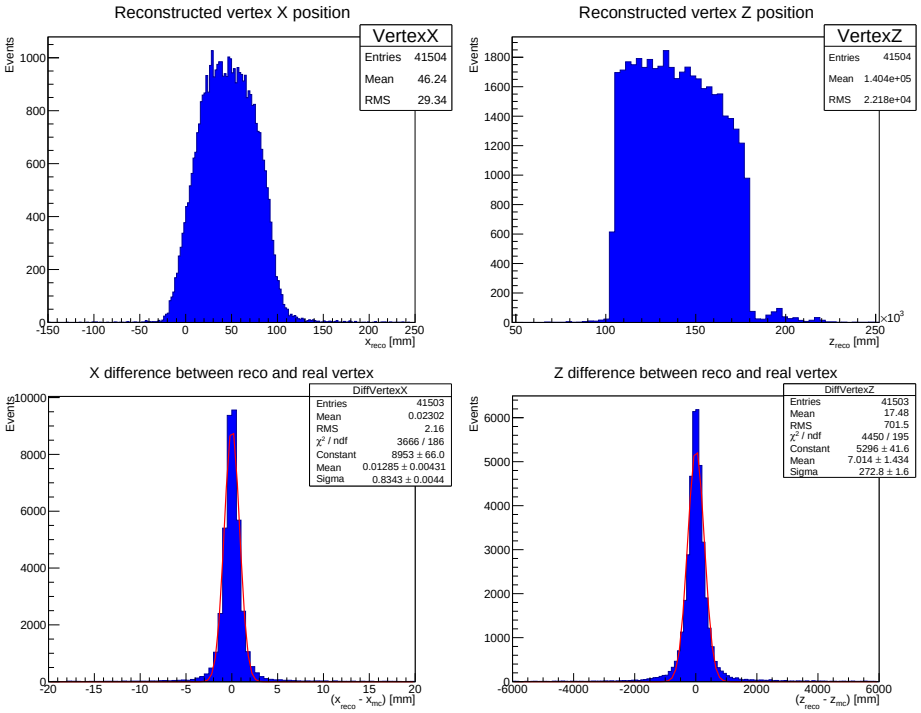

Figure 1: Top: histograms of the reconstructed vertex x and z positions. Bottom: histograms of the difference between the reconstructed and the real vertex x and z positions

```
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 $\vec{V}$ being in the middle of this vector is returned:

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

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

1.2 Results

The figure 1 shows one of the results of the **VertexCDA** analyzer. The two top histograms display the x and z values of the reconstructed vertex and the bottom ones display a gaussian fit of the difference between the reconstructed and the real vertex x and z values. The reconstructed x axis gaussian is centered around 0 with a very small standard deviation of less than 1mm. The y axis is similar to x. The z axis is slightly displaced of (7 ± 1)mm with a variance of (27.3 ± 0.2)cm. The tracks measured points being spaced of more than 100m and the scale of the decay zone being of 65m the relative error is smaller than the 1% level.

2 Pi0Reconstruction

This analyzer is implementing the reconstruction of a π^0 candidate from two photon candidates in the Liquid Krypton calorimeter. The eventual reconstructed candidate is then made available to further analyzer as a **KinePart** object.

The simple algorithm implemented works under strong hypothesis:

- The clusters in LKr are all due to individual photons.
- The event contains only a single π^0 that decayed into two photons, both in the acceptance of the LKr.

Every cluster is transformed to a photon candidate. The energy being the cluster energy and the momentum being computed using the reconstructed vertex. Every possible combination of candidate pair is tried and the pair with the invariant mass closest to m_{π^0} is selected to reconstruct the π^0.

2.1 Pi0Reconstruction 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 Pi0Reconstruction
```

The source code of the newly created analyzer can be found in **Examples/include/Pi0Reconstruction.hh** and **Examples/src/Pi0Reconstruction.cc**. Every standard method of the analyzer and each section of code will be described thereafter.

2.1.1 Constructor

As this analyzer only needs information from the LKr, this is the only requested TTree. The analyzer will run some acceptance checks as well and need access to the global instance of the **DetectorAcceptance** object:

```
RequestTree("LKr", new TRecoLKrEvent);
fDetectorAcceptanceInstance = GetDetectorAcceptanceInstance();
```

Without forgetting to include the header for the LKr reconstructed events

```
#include "TRecoLKrEvent.hh"
```

2.1.2 InitHist

In this method all the histograms that will be needed during the processing are created and registered to the framework:

- Histograms for the photon candidates reconstructed energy

```
BookHisto("g1Energy",
    new TH1I("G1Energy", "Energy of g1", 100, 0, 75000));
BookHisto("g2Energy",
    new TH1I("G2Energy", "Energy of g2", 100, 0, 75000));
```

- 2D histograms to compare reconstructed photon candidates energy with the true Monte Carlo energy

```
BookHisto("g1Reco",
    new TH2I("g1Reco", "g1 Reco vs. Real", 100, 0, 75000, 100, 0, 75000));
BookHisto("g2Reco",
    new TH2I("g2Reco", "g2 Reco vs. Real", 100, 0, 75000, 100, 0, 75000));
```

- 2D histograms to compare the reconstructed photon candidates positions on LKr with the extrapolated position of the true MC photons

```
BookHisto("g1px",
    new TH2I("g1px", "g1 px Reco vs. Real", 200, 0, 2000, 200, 0, 2000));
BookHisto("g2px",
    new TH2I("g2px", "g2 px Reco vs. Real", 200, 0, 2000, 200, 0, 2000));
BookHisto("g1py",
    new TH2I("g1py", "g1 py Reco vs. Real", 200, 0, 2000, 200, 0, 2000));
BookHisto("g2py",
    new TH2I("g2py", "g2 py Reco vs. Real", 200, 0, 2000, 200, 0, 2000));
BookHisto("g1pz",
    new TH2I("g1pz", "g1 pz Reco vs. Real", 10, 240000, 250000, 10, 240000,
     250000));
BookHisto("g2pz",
    new TH2I("g2pz", "g2 pz Reco vs. Real", 10, 240000, 250000, 10, 240000,
     250000));
```

- Histograms for the reconstructed π^0 candidate properties

```
BookHisto("pi0Energy",
    new TH1I("pi0Energy", "Energy of pi0", 100, 0, 75000));
BookHisto("pi0Mass",
    new TH1I("pi0Mass", "Reconstructed mass of pi0", 200, 0, 200));
```

- Histograms for LKr Monitoring

```
BookHisto("clusterPosition",
    new TH2I("clusterPosition",
        "Cluster position on LKr", 500, -2000, 2000, 500, -2000, 2000));
BookHisto("photonsNbr",
    new TH1I("photonsNbr", "Photons number/event", 10, 0, 10));
BookHisto("energyCalib",
    new TGraph());
BookHisto("g1EnergyFraction",
    new TH1I("g1EnergyFraction",
        "Fraction between real energy and reco energy", 1000, 0, 100));
BookHisto("g2EnergyFraction",
    new TH1I("g2EnergyFraction",
        "Fraction between real energy and reco energy", 1000, 0, 100));
```

- Histograms for the pair selection algorithm

```
BookHisto("gPairSelected",
    new TH1I("gPairSelected", "Pair of gamma selected for Pi0", 10, 0, 10));
```

- Histograms specific to Monte Carlo events

```
BookHisto("g1FirstVol",
    new TH1I("g1FirstVol", "First touched volume for g1", 15, 0, 15));
BookHisto("g2FirstVol",
    new TH1I("g2FirstVol", "First touched volume for g2", 15, 0, 15));
BookHisto("pdgID",
    new TH1I("pdgID", "Non complete events : pdgID", 0, 0, 0));
```

2.1.3 InitOutput

This analyzer should provide further analyzers with a π^0 candidate if any is found. The output object is first declared in the header:

```
KinePart pi0;
```

And then registered in the framework under the name "*pi0*" in the **InitOutput** method. It should be noted that to avoid collisions between independent 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 "*Pi0Reconstruction.pi0*"

```
RegisterOutput("pi0", &pi0);
```

2.1.4 DefineMCSimple

In this method, the specific event signature $K^+ \to \pi^+ X \pi^0 \to \gamma\gamma X$ is defined where X can be any kind and any number (including 0) of additional particle. This will allow to do extra-processing to assess the performances of the analyzer when running on this kind of simulated events.

```
int kID = fMCSimple->AddParticle(0, 321); //ask for beam Kaon
fMCSimple->AddParticle(kID, 211); //ask for positive pion from initial kaon decay
int pi0ID = fMCSimple->AddParticle(kID, 111); //ask for positive pion from initial
    kaon decay
fMCSimple->AddParticle(pi0ID, 22); //ask for positive pion from initial kaon decay
fMCSimple->AddParticle(pi0ID, 22); //ask for positive pion from initial kaon decay
```

2.1.5 Process

All the necessary variables are first declared

```
//Temporary variables
vector<KinePart*> photons;
KinePart *part;
TRecoLKrCandidate *lkrCand;
vector<pair<KinePart*, KinePart*> > candidates;
int i;

//Maps
map<double, int> photonOrder;
map<double, int>::reverse_iterator photonIt, photonIt2;
multimap<double, int> cID;
multimap<double, int>::iterator cIDIterator;

//For input from VertexCDA
TVector3 vertex, direction;
OutputState state;


//Variables for reconstruction
int LKrStartPos = 240413;
KinePart *g1,*g2;
double pi0Mass;
int iLead=-1;
int iTrail=-1;

//Variables for MC checks
double g1EnergyFrac = 0;
double g2EnergyFrac=0;
DetectorAcceptance::volume g1Vol, g2Vol;
bool g1LKr, g2LKr;
TVector3 g1reco, g2reco, g1real, g2real;
```

The event is retrieved from the LKr TTree and two calibration constants are declared for the LKr cluster energy correction:

```
//Get LKr event from TTree
TRecoLKrEvent *LKrEvent = (TRecoLKrEvent*)GetEvent("LKr");
```

```
//Calibration constants
double calibMult = 0.9744;
double calibConst = -366.5;
```

As stated previously some supplementary work can be done if running on specific Monte Carlo events. The use of simulated events is not enforced but the *withMC* flag is set accordingly:

```
bool withMC = true;
if(fMCSimple.fStatus == MCSimple::kMissing){
  for(int i=0; i<MCTruthEvent->GetNKineParts(); i++){
    FillHisto("pdgID",
        ((KinePart*)MCTruthEvent->GetKineParts()->At(i))->GetParticleName(), 1);
  }
  return;
}
if(fMCSimple.fStatus == MCSimple::kEmpty) withMC = false;
```

The vertex is required to reconstruct the photons momenta and is retrieved from the **VertexCDA** analyzer and its validity if verified.

```
//Get vertex from VertexCDA analyzer
vertex = *(TVector3*)GetOutput("VertexCDA.Vertex", state);
//Check we got the vertex. We cannot work without the vertex
if(state!=kOValid) return;
```

Then all the LKr clusters are transformed into photon candidates. The hypothesis is made that all clusters are due to a photon and that they all originate from the vertex. In which case the particle momenta is $\vec{p} = E_{cl} * \frac{(\vec{x}_{cl} - \vec{V})}{|\vec{x}_{cl} - \vec{V}|}$ where E_{cl} is the corrected cluster energy, $\vec{x}_{cl}$ is the position of the cluster on the LKr surface and $\vec{V}$ is the vertex. The processing can only continue if at least two photons candidates are found.

```
//Loop over the LKr clusters and create a KinePart photon candidate
for(int i=0; i<LKrEvent->GetNCandidates(); i++){
  lkrCand = (TRecoLKrCandidate*)LKrEvent->GetCandidate(i);
  part = new KinePart();
  //Fill the cluster position histogram
  FillHisto("clusterPosition",
      lkrCand->GetClusterX()*10, lkrCand->GetClusterY()*10);
  //Set the LKr position (*10 to go from cm to mm)
  direction.SetXYZ(lkrCand->GetClusterX()*10,
           lkrCand->GetClusterY()*10, LKrStartPos);
  //Set the direction
  direction = direction - vertex;
  //Set the magnitude of the direction to the corrected cluster energy to form the
     momentum (photon hypothesis)
  direction.SetMag(lkrCand->GetClusterEnergy()*1000*calibMult + calibConst);
  //Assign the properties to the KinePart
  part->SetProdPos(TLorentzVector(vertex, 0));
  part->SetInitialMomentum(direction);
  part->SetInitialEnergy(lkrCand->GetClusterEnergy()*1000*calibMult + calibConst);
  //Push the candidate in the list
  photonOrder.insert(pair<double, int>(
    lkrCand->GetClusterEnergy()*1000*calibMult + calibConst, photons.size()));
  photons.push_back(part);
}
//Fill the photon multiplicity histogram
FillHisto("photonsNbr", photonOrder.size());

i=0;
//We need at leat 2 photons to reconstruct the pi0
if(photonOrder.size()>=2){
```

In most of the case more than two photon candidates are found. This is mainly due to the contribution of the charged pion creating a cluster and the photons sometimes creating more than one cluster. The ratio between the invariant mass of every possible photon candidate pair and the π^0 mass is computed. The invariant mass is computed as $M = \sqrt{p_{g_1}^2 + p_{g_2}^2}$ where $p_{g_{1,2}}$ are the quadri-momenta of the photon candidates. The pair whose ratio is closest to 1 (the pair with the invariant mass closest to the π^0 mass) is chosen.

```
//Looping over possible photon pairs and computing invariant mass for each
photonIt2 = photonOrder.rbegin();
for(photonIt = photonOrder.rbegin(); photonIt != photonOrder.rend(); photonIt++){
  g1 = photons[photonIt->second];
  photonIt2 = photonIt;
  for(photonIt2++; photonIt2 != photonOrder.rend(); photonIt2++){
    g2 = photons[photonIt2->second];

    candidates.push_back(pair<KinePart*, KinePart*>(g1, g2));
    pi0Mass = sqrt(pow(g1->GetInitialEnergy() + g2->GetInitialEnergy(),2)
            - (g1->GetInitialMomentum() + g2->GetInitialMomentum()).Mag2());
    //insert the invariant mass divided by the pi0 mass in the map
    cID.insert(pair<double, int>(fabs((pi0Mass/134.9766)-1), i));
    i++;
  }
}
//Select the photon pair closest to the pi0 mass
FillHisto("gPairSelected", cID.begin()->second);
g1 = candidates[cID.begin()->second].first;
g2 = candidates[cID.begin()->second].second;
```

If Monte Carlo data is available the energy of the selected photon candidates are compared with the simulated photons coming from the π^0 decay. Tests are also done to see whether they are in the acceptance of the LKr or not.

```
//Are we working with MC?
if(withMC){

  //Select the most energetic photons coming from the pi0 decay
  if(fMCSimple["gamma"][0]->GetInitialEnergy() >= fMCSimple["gamma"][1]->
     GetInitialEnergy()) iLead = 0;
  else iLead = 1;
  iTrail = !iLead;

  //Compare the energy with the selected pair
  g1EnergyFrac = g1->GetInitialEnergy()/fMCSimple["gamma"][iLead]->GetInitialEnergy
     ();
  g2EnergyFrac = g2->GetInitialEnergy()/fMCSimple["gamma"][iTrail]->GetInitialEnergy
     ();

  //Are the 2 real photons in the LKr acceptance?
  fDetectorAcceptanceInstance->FillPath(fMCSimple["gamma"][iLead]->GetProdPos().Vect
     (), fMCSimple["gamma"][iLead]->GetInitialMomentum());
  g1Vol = fDetectorAcceptanceInstance->FirstTouchedDetector();
  g1LKr = fDetectorAcceptanceInstance->GetDetAcceptance(DetectorAcceptance::kLKr);
  fDetectorAcceptanceInstance->CleanDetPath();
  fDetectorAcceptanceInstance->FillPath(fMCSimple["gamma"][iTrail]->GetProdPos().
     Vect(), fMCSimple["gamma"][iTrail]->GetInitialMomentum());
  g2Vol = fDetectorAcceptanceInstance->FirstTouchedDetector();
  g2LKr = fDetectorAcceptanceInstance->GetDetAcceptance(DetectorAcceptance::kLKr);

  FillHisto("g1FirstVol", g1Vol);
  FillHisto("g2FirstVol", g2Vol);
}
```

If no Monte Carlo are available the photons are simply assumed to be in the LKr acceptance and the rest of the processing is only done if the photons are inside LKr.

```
else{
  g1Vol = DetectorAcceptance::kLKr;
  g2Vol = DetectorAcceptance::kLKr;
  g1LKr = true;
  g2LKr = true;
}

if( (g1Vol != DetectorAcceptance::kLAV && g1LKr == true)
  && (g2Vol != DetectorAcceptance::kLAV && g2LKr == true)){
```

Again if simulated data are available, the positions of the simulated photons are extrapolated at the LKr surface and compared with the selected candidates and the comparison histograms are filled.

```
//Expected position on LKr
g1reco = propagate(g1->GetProdPos().Vect(), g1->GetInitialMomentum(), LKrStartPos);
g2reco = propagate(g2->GetProdPos().Vect(), g2->GetInitialMomentum(), LKrStartPos);
```

```
if(withMC){
  //Comparison reconstructed momenta and energies between reconstructed and real
  g1real = propagate(fMCSimple["gamma"][iLead]->GetProdPos().Vect(), fMCSimple["
     gamma"][iLead]->GetInitialMomentum(), LKrStartPos);
  g2real = propagate(fMCSimple["gamma"][iTrail]->GetProdPos().Vect(), fMCSimple["
     gamma"][iTrail]->GetInitialMomentum(), LKrStartPos);

  FillHisto("g1px", g1reco.X(), g1real.X());
  FillHisto("g2px", g2reco.X(), g2real.X());
  FillHisto("g1py", g1reco.Y(), g1real.Y());
  FillHisto("g2py", g2reco.Y(), g2real.Y());
  FillHisto("g1pz", g1reco.Z(), g1real.Z());
  FillHisto("g2pz", g2reco.Z(), g2real.Z());
  FillHisto("g1Reco", g1->GetInitialEnergy(), fMCSimple["gamma"][iLead]->
     GetInitialEnergy());
  FillHisto("g2Reco", g2->GetInitialEnergy(), fMCSimple["gamma"][iTrail]->
     GetInitialEnergy());
  //Dont do it if we are likely to have selected the wrong photon
  if(g1EnergyFrac>0.95) FillHisto("energyCalib", g1->GetInitialEnergy(), fMCSimple["
     gamma"][iLead]->GetInitialEnergy());
  if(g2EnergyFrac>0.95) FillHisto("energyCalib", g2->GetInitialEnergy(), fMCSimple["
     gamma"][iTrail]->GetInitialEnergy());
  FillHisto("g1EnergyFraction", g1EnergyFrac);
  FillHisto("g2EnergyFraction", g2EnergyFrac);
}

FillHisto("g1Energy", g1->GetInitialEnergy());
FillHisto("g2Energy", g2->GetInitialEnergy());
```

Finally the π^0 candidate is reconstructed. Its energy is the sum of the photons energies and its momentum is the sum of the photon momenta. The state of the output is set as "*kOValid*" to signal further analyzer that the output can be used. The histograms corresponding to the π^0 candidate values are filled.

```
//Reconstruct the pi0 candidate and create the KinePart
pi0.SetInitialEnergy(g1->GetInitialEnergy() + g2->GetInitialEnergy());
direction = g1->GetInitialMomentum();
direction.SetMag(g1->GetInitialEnergy());
g1->SetInitialMomentum(direction);
direction = g2->GetInitialMomentum();
direction.SetMag(g2->GetInitialEnergy());
g2->SetInitialMomentum(direction);
pi0.SetInitialMomentum(g1->GetInitialMomentum() + g2->GetInitialMomentum());

//Set the output state as valid (we have a candidate)
SetOutputState("pi0", kOValid);

//Fill the pi0 histograms
FillHisto("pi0Energy", pi0.GetInitialEnergy());
FillHisto("pi0Mass", sqrt(pow(pi0.GetInitialEnergy(),2) - pi0.GetInitialMomentum().
     Mag2()));
```

The last step is to release all the memory that has been allocated during the processing. This can safely be done here as this memory is not part of the output. If memory had been allocated for the output, the release would have been done in the **PostProcess** method.

```
//Delete all the created KinePart
while(photons.size()>0){
  delete photons.back();
  photons.pop_back();
}
```

2.1.6 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.

2.1.7 ExportPlot

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

```
|| SaveAllPlots();
```

2.1.8 DrawPlot

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

```
|| DrawAllPlots();
```

2.2 Results

The figure 2 shows some results of this analyzer run on simulated $K^+ \to \pi^+\pi^0 \to \gamma\gamma$ events. The top plots shows the reconstructed energy of the π^0 candidate when found and the invariant mass of the photon pair being selected as decay product of the pi^0. The mass peak is center on 129.6 MeV. The bottom plot is the graph of the reconstructed energy vs. the MC energy for the selected photons. A linear fit is done on these points to extract the calibration constants (or verify them in case they are alredy applied).

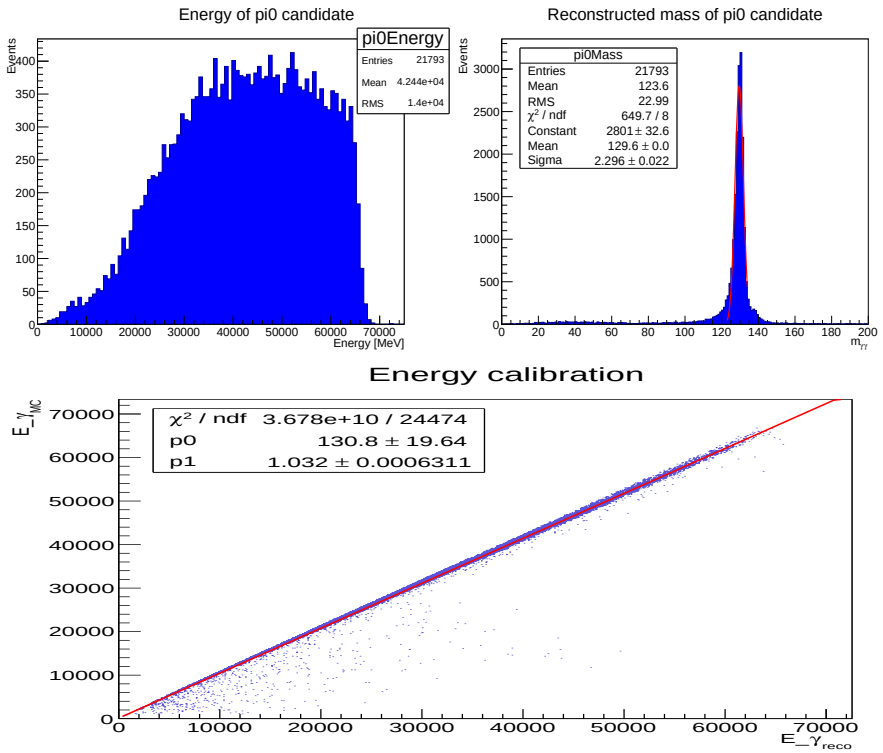

Figure 2: Top: histograms of the reconstructed π^0 candidate mass and energy. Bottom: Fit of the E_{reco} vs. E_{MC} slope for the verification of the calibration constants.