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 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.

xxxDescription of the algorithmxxx

1.1 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.

1.1.1 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"
```

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 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));
BookHisto("pi0MCMass",
    new TH1I("pi0MCMass", "MC 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));
```

1.1.3 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);
```

1.1.4 DefineMCSimple

In this method, the specific event signature $K^+ \rightarrow \pi^+ X \pi^0 \rightarrow \gamma\gamma X$ 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 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
```

1.1.5 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 cluster 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 $M = \sqrt{p_{g1}^2 + p_{g2}^2}$ 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 are 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 wether 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()));
FillHisto("pi0MCMass", sqrt(pow(fMCSimple[22][iLead]->GetInitialEnergy() + fMCSimple
     [22][iTrail]->GetInitialEnergy(), 2) - (fMCSimple[22][iLead]->
     GetInitialMomentum() + fMCSimple[22][iTrail]->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();
}
```

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();
```

1.2 Validation