Documentation for the `gluex_root_analysis` software

The GlueX Software Group

July 30, 2020

Abstract

The *GlueX* ROOT analysis software is used to analyze data the standard PART (Physics Analysis ROOT Tree) format based on the *DSelector* software concept. The raw experimental data in EVIO format are input to the *GlueX* reconstruction software that reconstructs charged particle tracks and neutral particle showers and stores this information into the REST data format. Using the *ReactionFilter* plugin, the *GlueX* Analysis library will build combinations of final state particles and apply standard loose event selections to generate skims of the reconstructed data based on the event reaction supplied by the analyzer. The output of *ReactionFilter* is a ROOT file which can be read by the *DSelector* software package and provides the base for the a *GlueX* ROOT analysis that is the subject of this document. We will discuss basic concepts of the *DSelector*, the fields available in the PART format, and methods to perform various typical analysis activities. In the following discussion it is assumed you have configured a working *GlueX* software environment.

1 Introduction

The *GlueX* ROOT analysis software package provides an executable named `MakeDSelector` that allow the user to generate a basic framework of *DSelector* program files for the analysis of a particular reaction. This code will be different for each reaction. The location of this executable is defined by the environment variable `ROOT_ANALYSIS_HOME`, which is configured by the standard GlueX software configuration scrripts. An example of how to use this is shown below:

```
MakeDSelector input-root-tree-file-name root-tree-name program-name
```

where `input-root-tree-file-name` is the name of an arbitrary input ROOT file, `root-tree-name` is the name of the `TTree` contained in this file `program-name` can be any label you choose. The command above will create two files in the current directory with the name `DSelector_program-name` with the extensions `.C` and `.h` and contains the basic template to read and analyze the specified ROOT tree. The files contain the general functionality with comments and example code for the user to expand on.

2 The DSelector class structure

In the `.h` header file you find a class defined as `DSelector_program-name` that inherits from the `DSelector` class that is defined in the *GlueX* root analysis package library and provides all necessary methods to read the ROOT tree. The `DSelector` class also inherits from the ROOT `TSelector` class that provides the methods `Init(), Process()` and `Finalize()` that form the base of the generated `.C` file and the structure to loop over all of the entries in the ROOT tree.

The `DSelector_program-name` can be run with a root script that has the following form:

```
TChain chain("name-of-the-root-tree");
chain.Add("root-file-name");
gROOT->ProcessLine(".x $ROOT_ANALYSIS_HOME/scripts/Load_DSelector.C");
chain.Process("DSelector_program-name.C++");
```

Note, that you should supply either one or two plus signs at the end of the `DSelector` file name to increase the processing speed of your analysis. If you specify `+`, the DSelector will only be recompiled if you modify

either of the DSelector files. The preferred option is ++, which will force recompilation at every execution, which can be help protect against errors that occur when switching between software versions.

Alternatively, the DSelector_program-name can be run in parallel on multiple cores with PROOF-Lite:

```
int NThreads = 16; // or as many as you want to use
TChain *chain = new TChain("name-of-the-root-tree");
chain.Add("root-file-name");
gROOT->ProcessLine(".x $ROOT_ANALYSIS_HOME/scripts/Load_DSelector.C");
DPROOFLiteManager *dproof = new DPROOFLiteManager();
dproof->Process_Chain(chain, "DSelector_program-name.C+", NThreads, "outputHistFileName",
    "outputTreeFileName");
```

2.1 The Wrapper concept

Each piece of information is stored as a branch in the ROOT tree. To provide an interface to access this information corresponding more closely to the reconstructed physics object, various types of wrappers are implemented. These wrappers are provided by the *DSelector* and are defined in ROOT_ANALYSIS_HOME/libraries/DSelector. There are four basic type of objects in the tree – charged particle tracks, neutral particles, combos and beam photons – which leads to the definition of four standard wrapper objects:

```
  // OBJECT ARRAYS: RECONSTRUCTED
  DChargedTrackHypothesis* dChargedHypoWrapper;
  DNeutralParticleHypothesis* dNeutralHypoWrapper;
  DBeamParticle* dBeamWrapper;
  DParticleCombo* dComboWrapper;
```

The first three wrappers are used to access the reconstructed final state particles, charged and neutral, and the reconstructed initial state beam photons. The forth wrapper represents the output of the standard analysis library used by *ReactionFilter*, which combines the initial and final state particles, with kinematic fitting by default, to generate the full event. The resulting final state particles and intermediate states (if so defined) from the *ReactionFilter*are accessed by additional wrappers specific to the reaction at hand. As an example, in the case of an exclusive final state like $\pi^+ + \pi^- + \pi^0 + p$ with the π^0 decaying into two photons and its mass being constrained in the kinematic fit will lead to 8 additional wrappers in the generated *DSelector* that include a wrapper for the initial state beam photon and the decay of the π^0 with its mass being constrained by the fit:

```
  //Step 0
DParticleComboStep* dStep0Wrapper;
DBeamParticle* dComboBeamWrapper;
DChargedTrackHypothesis* dPiPlusWrapper;
DChargedTrackHypothesis* dPiMinusWrapper;
DChargedTrackHypothesis* dProtonWrapper;

//Step 1
DParticleComboStep* dStep1Wrapper;
DKinematicData* dDecayingPi0Wrapper;
DNeutralParticleHypothesis* dPhoton1Wrapper;
DNeutralParticleHypothesis* dPhoton2Wrapper;
```

For example, to loop over all charged particle tracks of an event and perform a calculation using the value of the start counter energy-loss one would think to do something like the following:

```
 // check if any charged track has a ST hit by looking at its dEdx value
int NChargedTracksHypos = (int)Get_NumChargedHypos();
int FoundStartCounterHits = 0;
for (int k=0; k<NChargedTracksHypos; k++){
  dChargedHypoWrapper->Set_ArrayIndex(k);
  float ST_dEdx = dChargedHypoWrapper->Get_dEdx_ST();
```

```
  if (ST_dEdx>0.){
    FoundStartCounterHits++; // NOTE THAT THIS CODE OVERCOUNTS THE NUMBER OF HITS!
  }
}
```

The key to this loop is that one sets pointer of the wrapper to the right index by using the method `Set_ArrayIndex(k)`. Now that the wrapper points to the desired object one can access the desired value from that object.

Note that this code will not do what the user desires for several reasons. First, by default *ReactionFilter*will only write out the initial and final state particles used in the combos saved to the ROOT tree, unless the `U1` option is specified. Second, each charged track is reconstructed under several different mass hypotheses (e, π, K, $p/\bar{p}$ by default), and it is these specific hypotheses which are used in the reconstructed combos which are saved to the ROOT tree. The mapping between charged particle hypotheses and reconstructed tracks is given by the `DChargedTrackHypothesis::Get_TrackID()` function, so one could in principle keep track of the number of unique track IDs, but in general this is an issue to be very careful about.

2.2 `Init(TTree *locTree)` Method

This function is used to initialize histograms and any other class variables which will be used by the analysis code for every event. This method can and most likely will be called more than once, in particular when using a `TChain` with more than one root file or when using PROOF in the context of multi-threading. This method is inherited from the ROOT `TSelector` class and overridden in the `DSelector` class. The code snippet in the `Init()` method shown below will ensure that the code that follows after will be executed only once:

```
bool locInitializedPriorFlag = dInitializedFlag; //save whether have been initialized previously
DSelector::Init(locTree); //This must be called to initialize wrappers for each new TTree
//gDirectory now points to the output file with name dOutputFileName (if any)
if(locInitializedPriorFlag)
  return; //have already created histograms, etc. below: exit
```

Therefore any initialization of variables or histogram definitions intended for the whole analysis need to be done after this part of the code. This includes any definitions of Analysis Actions and Cut Actions as well as custom branches for an output ROOT tree.

The call to the method `Get_ComboWrappers()` initializes all instances of wrappers available for the analysis of the ROOT tree. This call will configure a wrapper for each particle in the reaction. For example if the reaction is $\gamma p \to p\pi^0\pi^+\pi^-$, $\pi^0 \to \gamma\gamma$ the following pointers defined in the class header will be initialized

```
//Step 0
dComboBeamWrapper = static_cast<DBeamParticle*>(dStep0Wrapper->Get_InitialParticle());
dPiPlusWrapper = static_cast<DChargedTrackHypothesis*>(dStep0Wrapper->Get_FinalParticle(1));
dPiMinusWrapper = static_cast<DChargedTrackHypothesis*>(dStep0Wrapper->Get_FinalParticle(2));
dProtonWrapper = static_cast<DChargedTrackHypothesis*>(dStep0Wrapper->Get_FinalParticle(3));

//Step 1
dStep1Wrapper = dComboWrapper->Get_ParticleComboStep(1);
dDecayingPi0Wrapper = dStep1Wrapper->Get_InitialParticle();
dPhoton1Wrapper = static_cast<DNeutralParticleHypothesis*>(dStep1Wrapper->Get_FinalParticle(0));
dPhoton2Wrapper = static_cast<DNeutralParticleHypothesis*>(dStep1Wrapper->Get_FinalParticle(1));
```

These wrapper objects are instances of classes as shown in the above example and are pointers to the corresponding instances of the particles for a given combo and information about the combo itself. All instances are defined in the *DSelector* library and provide methods to access any data within the tree for a given combo. So for example with the line

```
TLorentzVector xv4 = dPhoton1Wrapper->Get_X4_Shower();
```

you will access the Lorentz 4-vector for the calorimeter shower associated with the first photon used in this example reaction for a given combo within the combo loop. This will be discussed in more detail below when

describing the `Process()` method.

The full particle combination corresponding to the reconstructed reaction is described by a `DParticleCombo` object, which contains one or more `DParticleComboStep` objects. Each `DParticleComboStep` corresponds to an intermediate decay step in the overall reaction. For example, the reaction $\gamma p \to p\pi^+\pi^-$ has only one step (`Step0`), while $\gamma p \to p\pi^0\pi^+\pi^-$, $\pi^0 \to \gamma\gamma$ has two steps, with `Step0` being the primary photoproduction reaction and `Step1` being the $\pi^0 \to \gamma\gamma$ decay.

Each final state particle is either charged or neutral and is accessed through a wrapper of type `DNeutralParticleHypoth` or `DChargedTrackHypothesis`. There are many methods associated with these two classes and provide access to all relevant variables in the tree associated with these particles within a combo. Intermediate particles that decay into some final state particles will also be represented by a wrapper class of type `DKinenaticData` but only if the mass of this intermediate state particle is constrained in the kinematic fit when requested by the reaction filter. If the intermediate particle mass is not constrained in the kinematic fit, then its properties (e.g. 4-momentum) can be constructed by the user based on the other reconstructed particles. All beam photons are represented by the wrapper class `DBeamParticle`. Note that the beam photon energies are not altered by the kinematic fitter. More details about these wrappers and their methods will be discussed further below when discussing the loop over all combos of an event.

2.2.1 Histogram definitions

Histograms can be defined easily as in any root script. The user will need to define histograms as needed for their analysis. Note that you should define the histogram variables in the class definition in the `.h` file before creating them in the `DSelector::Init()` function.

2.2.2 AnalysisAction

NEED TEXT TO EXPLAIN

2.2.3 CutAction

NEED TEXT TO EXPLAIN

2.3 `Process(Long64_t locEntry)` Method

This is the main event loop which is called for each event. There are three main sections in the default code which is provided. The first is for any event-level initialization or calculations. The second comprises the loop over all available combinations in the event. The third is for additional event-level calculations and output.

The data for the event is read from the tree with the following code section:

```
//CALL THIS FIRST
 DSelector::Process(locEntry); //Gets the data from the tree for the entry
```

Note that this method `Process()` is inherited from the ROOT class `TSelector` This method is overridden by the `DSelector` class and calls the method `Get_Entry()` defined in the class `DTreeInterface`, which reads the event data from file into memory. Remember that in the case a `TChain` is used and a new tree file is opened, the method `Init()` will be executed again first.

If you are using analysis actions, then they need to be initialized for each event:

```
 /*** SETUP UNIQUENESS TRACKING ****/
 //ANALYSIS ACTIONS: Reset uniqueness tracking for each action
 //For any actions that you are executing manually, be sure to call Reset_NewEvent() on them here
 Reset_Actions_NewEvent();
 dAnalyzeCutActions->Reset_NewEvent(); // manual action, must call Reset_NewEvent()
```

If you are filling histograms yourself, the combinatorics must be carefully handled to avoid double-counting, either by "uniqueness tracking" or some other method. The discussion of these techniques is the focus of another document. [Add reference here when it's ready]

2.3.1 Looping over combos in an event

At this point the loop over all combos for this event is stared. A combo is a combinations of a beam photon with the final state particles, which pass some loose event selection criteria. Note that even if you have a unique combination of final state particles in an event, you will have multiple combinations if there are multiple beam photons sufficiently in time with this event, which is often the case in the intense GlueX photon beam. Typically, combos are kept if the reconstructed final state particles are in time with the tagged beam photons within 3 or 4w of the 4 ns beam bunches. For example, if you are looking at the reaction $\gamma p \to p\pi^+\pi^-$, and find an event with only one candidate each for a negatively charged pion, a postively charged pion, and a proton, and 3 tagged photons within the required timing window, then you will end up with 3 combos for an event. All necessary wrappers are configured by calling the `dComboWrapper->Set_ComboIndex(UInt_t)` function, as shown below. qa Note that the *DSelector* can be run not just over the output of the *ReactionFilter* plugin, but the *DSelector* can also write out ROOT trees which contain a set of reduced events and/or branches containing additional information. When these trees are created, if additional selections are being applied on the combos being analyzed, an event will still be written out as long as least one combo in the event passes the selections. To keep track of which combos have been rejected and which are kept for further analysis, we use the "ComboCut" flag. This can be set using `dComboWrapper->Set_IsComboCut(bool)` and its value retrieved using `dComboWrapper->Get_IsComboCut()`.

An example beginning of the combo loop is shown below:

```
for(UInt_t loc_i = 0; loc_i < Get_NumCombos(); ++loc_i)
{
  //Set branch array indices for combo and all combo particles
  dComboWrapper->Set_ComboIndex(loc_i);

  // Is used to indicate when combos have been cut
  if(dComboWrapper->Get_IsComboCut()) // Is false when tree originally created
    continue; // Combo has been cut previously

  // ********************************************
  // now from here on out you can do your stuff!
  // ********************************************
```

A detailed discussion of the data available through the different particle wrappers is given in Section 3. By default, code is added to configure variables to store the unique ID numbers of each particle in the combination, along with their measured and kinematically-fitted momentum 4-vectors. Equipped with the data access methods described below and the wrappers that provide the access to these methods, we can get any data within the tree. Continuing with the above example of three pions and a recoil proton in the final state, we can calculate the invariant mass of the 3-pion system using either the measured momentum 4-vectors or those determined by the kinematic fitter. In the below example, we also give examples of how to access other information, in particular the shower quality factor for a neutral particles and and the z-position of the associated shower in the calorimeter:

```
// These quantities are from the kinematic fitter
TLorentzVector locBeamP4 = dComboBeamWrapper->Get_P4();
TLorentzVector locPiPlusP4 = dPiPlusWrapper->Get_P4();
TLorentzVector locPiMinusP4 = dPiMinusWrapper->Get_P4();
TLorentzVector locProtonP4 = dProtonWrapper->Get_P4();
TLorentzVector locPhoton1P4 = dPhoton1Wrapper->Get_P4();
TLorentzVector locPhoton2P4 = dPhoton2Wrapper->Get_P4();

// 3-Pion Final state based on kinematic fit quantities:
TLorentzVector ThreePiFinalState = locPiPlusP4 + locPiMinusP4 + locPhoton1P4 + locPhoton2P4;

// These quantities are the measured quantities
TLorentzVector locBeamP4_Measured = dComboBeamWrapper->Get_P4_Measured();
TLorentzVector locPiPlusP4_Measured = dPiPlusWrapper->Get_P4_Measured();
TLorentzVector locPiMinusP4_Measured = dPiMinusWrapper->Get_P4_Measured();
TLorentzVector locProtonP4_Measured = dProtonWrapper->Get_P4_Measured();
```

```
TLorentzVector locPhoton1P4_Measured = dPhoton1Wrapper->Get_P4_Measured();
TLorentzVector locPhoton2P4_Measured = dPhoton2Wrapper->Get_P4_Measured();

// 3-Pion Final state based on measured quantities:
TLorentzVector ThreePiFinalState_Measured = locPiPlusP4_M + locPiMinusP4_M + locPhoton1P4_M +
    locPhoton2P4_M;

// Get the shower quality factor of the first photon and find the z-position of the shower
float qf = dPhoton1Wrapper->Get_Shower_Quality();
TLorentzVector Gamma1_Shower = dPhoton1Wrapper->Get_X4_Shower();
float Shower1Zposition = Gamma1_Shower.Z();
```

2.3.2 Handling Accidental Beam Photon Contributions

The reconstructed final state particle combination is associated with the timing of a particular accelerator beam bunch (which we will call the "RF time"). The measured initial state beam photon time will then either be consistent, or "in-time" with this selected beam bunch, or "out-of-time" with it. While signal events are generally always associated with in-time beam photons, there is also an in-time contribution due to combinations where the beam photon is not the real photon which created the reconstructed final state particles (for example, due to multiple in-time photons or due to the correct photon not being reconstructed). There are several methods which can be used to subtract this background contribution, the most common of which is to estimate it using out-of-time beam photon combos and subtract the background by filling histograms with appropriate weights, or propagating these weights to later analysis stages.

In this method, in-time beam photons are assigned a weight of 1, while out-of-time beam photons are assigned a weight of $1/n$, where n is the number of out-of-time beam bunches that are considered in the analysis. The RF time is determined from the combo using the `dComboBeamWrapper()->Get_RFTime()` method. The following code shows an example of how to determine the RF time of the beam photon used in a given combo and how to define the weight for the combo based on the RF time of the beam photon. In this particular example it is expected that 4 beam bunches on either side of the in-time beam bunch (at $t = 0$) are available leading to a $-1/8$ weight factor. An additional run-dependent multiplicitive factor should be applied, to correct for non-uniformities in the out-of-time peaks, which can be acccessed through `DAnalysisUtilities::Get`$_{A}accidentalScalingFactor()$.

```
TLorentzVector locVertex = dComboBeamWrapper->Get_X4_Measured();
float locRFTime = dComboWrapper->Get_RFTime();
TLorentzVector locBeamP4 = dComboBeamWrapper->Get_P4();
.....
float DT_RF = locVertex.T() - (locRFTime + (locVertex.Z() - dTargetCenter.Z())/29.9792458);

// Now an event weight can be assigned:
//    if DT_RF = +/- 2ns within zero the beam photon is in time
//                      within +-4, +-8, +-12, ... the beam photon is out of time
double weight = 1.;
if (TMath::Abs(DT_RF)>2.){ // TOFIX: should actually pull the real beam bunch spacing for this
    weight = -0.125;       // -1/8
}
weight *= dAnalysisUtilities->Get_AccidentalScalingFactor(); // correct for correlation effects

// the weight then can be used to fill a histogram with the weight
// thereby automatically subtract accidental background
// for example, dEBeam->Fill(locBeamP4.E(), weight);
```

2.3.3 Kinematic Fit Quality

The *ReactionFilter* ROOT trees are generated with no constraint on the quality of the kinematic fit by default. This quantity can be very powerful for separating signal from background. This information can be accessed in one of two ways: by looking at the χ^2 of the kinematic

fit (suggested), which should have some distribution peaking at 1 (when normalized by the number of degrees of freedom in the fit) and falling towards higher values for the signal, or the confidence level of the fit, which should have a roughly flat distribution of the signal, with the background piling up around 0. There is no preferred cut value for a general case; you should choose a value by studying these distributions for your signal and some background contributio that you would like to reject. This information can be accessed in the following way:

```
// Kinematic fit confidence level and chi^2/d.o.f.
double chi2dof = dComboWrapper->Get_ConfidenceLevel_KinFit("") /
    dComboWrapper->Get_NDF_KinFit("");
double chi2dof = dComboWrapper->Get_ChiSq_KinFit("") / dComboWrapper->Get_NDF_KinFit("");
  // reject events with a poor chi^2 for this reaction
if(chi2dof > 5.) {
   dComboWrapper->Set_IsComboCut(true); // reject this combo for future consideration
   continue;
}
```

2.3.4 Long-Lived Particles

As previously discussed, particle combinations can have multiple steps in order to model the effect of additional decays of narrow resonances. These decaying particles can be short-lived $(\omega, \eta, \ldots)$ or long-lived $(K_S, \Lambda, \ldots)$. In the case of long-lived particles, additional event selections can be made to separate them from combinatorial background. The most accurate information on the displacement of the secondary vertex from the primary vertex is obtained from the kinematic fit, particularly when a vertex constraint is included. Two common variables to cut on are the magnitude of this displacement, or ''flight distance'', and this magnitude divided by the uncertainty in the position of these vertices, or ''flight significance''. Below, we give an example for the reaction $\gamma p \to K^+K^+\Xi^-(1320)$, which has two detatched vertices: $\Xi^- \to \pi^-\Lambda^0$, $\Lambda^0 \to p\pi^-$.

The cases in which the decaying particle is and is not mass constrained in the fit are handled differently. If no mass constraint is applied in the fit, first obtain the production X4 (vertex spatial position and time) and the X4 for the decaying particle. The pathlength of the decaying particle is the magnitude of the difference between the two vertices. The uncertainty of the pathlength is saved directly to the tree, but must be extracted by hand. The flight significance is then the pathlength divided by the uncertainty in the pathlength.

```
TLorentzVector locProdSpacetimeVertex = dComboBeamWrapper->Get_X4();
TLorentzVector locDecayingXiX4 =
    dTreeInterface->Get_TObject<TLorentzVector>("DecayingXiMinus__X4",loc_i); // change if the
    mass is constrained in the fit
TLorentzVector locDeltaSpacetimeXi = locProdSpacetimeVertex - locDecayingXiX4;
double locPathLengthXi = locDeltaSpacetimeXi.Vect().Mag();
float locPathLengthSigmaXi = Get_Fundamental<Float_t>("DecayingXiMinus__PathLengthSigma", loc_i);
double locPathLengthSignificanceXi = locPathLengthXi/locPathLengthSigmaXi;
```

If a mass constraint on the decaying particle was applied, then a wrapper for the decaying particle is configured and should be used to obtain the vertex position instead:

```
TLorentzVector locDecayingXiX4 = dDecayingXiMinusWrapper->Get_X4();
```

For a secondary detached vertex, for example the Λ decay in $\Xi^- \to \Lambda\pi^-$, the only difference in the calculation is to compare it to the parent detatched vertex:

```
TLorentzVector locDecayingLambX4 =
    dTreeInterface->Get_TObject<TLorentzVector>("DecayingLambda__X4",loc_i);
TLorentzVector locDeltaSpacetimeLamb = locDecayingXiX4 - locDecayingLambX4;
double locPathLengthLamb = locDeltaSpacetimeLamb.Vect().Mag();
```

```
float locPathLengthSigmaLamb = Get_Fundamental<Float_t>("DecayingLambda__PathLengthSigma", loc_i);
double locPathLengthSignificanceLamb = locPathLengthLamb/locPathLengthSigmaLamb;
```

2.3.5 Analyzing Thrown Information

When analyzing simulated data (MC), information about the particles thrown by the physics event generator is also stored for each event. The data is stored in objects of type DMCThrown and is also accessible through a wrapper interface. This information can be accessed in several different ways. Each DChargedTrackHypothesis and DNeutralParticleHypothesis has a member function named Get_ThrownIndex(), which can be passed as an argument to dThrownWrapper->Set_ArrayIndex(). The list of thrown particles can also be studied directly, such as in the following example:

```
//Thrown beam: just use directly
if(dThrownBeam != NULL)
   double locEnergy = dThrownBeam->Get_P4().E();

//Loop over throwns
for(UInt_t loc_i = 0; loc_i < Get_NumThrown(); ++loc_i)
{
   //Set branch array indices corresponding to this particle
   dThrownWrapper->Set_ArrayIndex(loc_i);

   //Do stuff with the wrapper here ...
   //Example - fill a histogram of the momentum distribution of protons
   if(dThrownWrapper->Get_PID() == Proton)
       dProtonPmag->Fill(dThrownWrapper->Get_P4().Vect().Mag());
}
```

2.4 ROOT Tree Output

Besides histograms, the *DSelector* can also output TTrees for further analysis, either by another *DSelector*or by another program. Two types of TTree output are support: the standard *ReactionFilter* PART format, which stores data mostly in various arrays of objects, and a ‘‘flat’’ tree with a simpler structure.

The code to handle ROOT tree output is in several locations, and generally works through the (thread-safe) DTreeInterface class. In the Init() function, besides defining the names of output files, you can optionally add branches to store additional event/combo information. The following shows an example of these features:

```
 // In Init() function...

//USERS: SET OUTPUT FILE NAME //can be overriden by user in PROOF
dOutputFileName = "ggg.root"; //"" for none
dOutputTreeFileName = ""; //"" for none
dFlatTreeFileName = ""; //output flat tree (one combo per tree entry), "" for none
dFlatTreeName = ""; //if blank, default name will be chosen

/************************* EXAMPLE USER INITIALIZATION: CUSTOM OUTPUT BRANCHES - MAIN TREE
    *************************/

//EXAMPLE MAIN TREE CUSTOM BRANCHES (OUTPUT ROOT FILE NAME MUST FIRST BE GIVEN!!!! (ABOVE: TOP)):
//The type for the branch must be included in the brackets
//1st function argument is the name of the branch
//2nd function argument is the name of the branch that contains the size of the array (for
    fundamentals only)
dTreeInterface->Create_Branch_Fundamental<Int_t>("my_int"); //fundamental = char, int, float,
    double, etc.
```

```
dTreeInterface->Create_Branch_FundamentalArray<Int_t>("my_int_array", "my_int");
dTreeInterface->Create_Branch_FundamentalArray<Float_t>("my_combo_array", "NumCombos");
dTreeInterface->Create_Branch_NoSplitTObject<TLorentzVector>("my_p4");
dTreeInterface->Create_Branch_ClonesArray<TLorentzVector>("my_p4_array");

/************************* EXAMPLE USER INITIALIZATION: CUSTOM OUTPUT BRANCHES - FLAT TREE
    ************************/

//EXAMPLE FLAT TREE CUSTOM BRANCHES (OUTPUT ROOT FILE NAME MUST FIRST BE GIVEN!!!! (ABOVE: TOP)):
//The type for the branch must be included in the brackets
//1st function argument is the name of the branch
//2nd function argument is the name of the branch that contains the size of the array (for
    fundamentals only)
dFlatTreeInterface->Create_Branch_Fundamental<Int_t>("flat_my_int"); //fundamental = char, int,
    float, double, etc.
dFlatTreeInterface->Create_Branch_FundamentalArray<Int_t>("flat_my_int_array", "flat_my_int");
dFlatTreeInterface->Create_Branch_NoSplitTObject<TLorentzVector>("flat_my_p4");
dFlatTreeInterface->Create_Branch_ClonesArray<TLorentzVector>("flat_my_p4_array");
```

In the Process() function, these trees can be filled either once per event (i.e. before the combo loop)

```
/************************************* EXAMPLE: FILL CUSTOM OUTPUT BRANCHES
    *************************************/
Int_t locMyInt = 7;
dTreeInterface->Fill_Fundamental<Int_t>("my_int", locMyInt);

TLorentzVector locMyP4(4.0, 3.0, 2.0, 1.0);
dTreeInterface->Fill_TObject<TLorentzVector>("my_p4", locMyP4);

for(int loc_i = 0; loc_i < locMyInt; ++loc_i)
  dTreeInterface->Fill_Fundamental<Int_t>("my_int_array", 3*loc_i, loc_i); //2nd argument =
      value, 3rd = array index
```

Or once per combo (i.e. inside the combo loop)

```
  /************************************* EXAMPLE: FILL CUSTOM OUTPUT BRANCHES
      *************************************/
  TLorentzVector locMyComboP4(8.0, 7.0, 6.0, 5.0);
  //for arrays below: 2nd argument is value, 3rd is array index
  //NOTE: By filling here, AFTER the cuts above, some indices won't be updated (and will be
      whatever they were from the last event)
    //So, when you draw the branch, be sure to cut on "IsComboCut" to avoid these.
  dTreeInterface->Fill_Fundamental<Float_t>("my_combo_array", -2*loc_i, loc_i);
  dTreeInterface->Fill_TObject<TLorentzVector>("my_p4_array", locMyComboP4, loc_i);
```

For the normal tree output, the tree should be written out at the end of Process():

```
/********************************** EXAMPLE: FILL CLONE OF TTREE HERE WITH CUTS APPLIED
    **********************************/
Bool_t locIsEventCut = true;
for(UInt_t loc_i = 0; loc_i < Get_NumCombos(); ++loc_i) {
  //Set branch array indices for combo and all combo particles
  dComboWrapper->Set_ComboIndex(loc_i);
  // Is used to indicate when combos have been cut
  if(dComboWrapper->Get_IsComboCut())
    continue;
  locIsEventCut = false; // At least one combo succeeded
  break;
}
```

```
if(!locIsEventCut && dOutputTreeFileName != "")
  Fill_OutputTree();
```

For a flat tree, this format has one entry per combo, and so it should be filled and written out right at the end of the combo loop, for example:

```
/****************************************** FILL FLAT TREE (IF DESIRED)
    ******************************************/
//FILL ANY CUSTOM BRANCHES FIRST!!
Int_t locMyInt_Flat = 7;
dFlatTreeInterface->Fill_Fundamental<Int_t>("flat_my_int", locMyInt_Flat);

TLorentzVector locMyP4_Flat(4.0, 3.0, 2.0, 1.0);
dFlatTreeInterface->Fill_TObject<TLorentzVector>("flat_my_p4", locMyP4_Flat);

for(int loc_j = 0; loc_j < locMyInt_Flat; ++loc_j)
{
  dFlatTreeInterface->Fill_Fundamental<Int_t>("flat_my_int_array", 3*loc_j, loc_j); //2nd
      argument = value, 3rd = array index
  TLorentzVector locMyComboP4_Flat(8.0, 7.0, 6.0, 5.0);
  dFlatTreeInterface->Fill_TObject<TLorentzVector>("flat_my_p4_array", locMyComboP4_Flat,
      loc_j);
}

//FILL FLAT TREE
Fill_FlatTree(); //for the active combo
```

3 DSelector Data Classes

All the quantities in the ROOT file are accessed through some wrapper objects defined in your DSelector. The variable names used in the ROOT trees and the generated *DSelector* code have distinct qualifiers to help to indicate their meaning and are explained here in more detail.

Variable names prefixes and suffixes

Beam_	Beam photon related quantity/ies for a given event before Kinematic Fit
NeutralHypo_	Neutral Particle related quantity/ies for a given event before Kinematic Fit
ChargedHypo_	Charged Particle related quantity/ies for a given event before Kinematic Fit
_Measured	A measured quantity as determined by the reconstruction software
_KinFit	A measured quantity that has been altered by the Kinematic Fitter.
_X4	4-vector containing position and time (TLorentzVector).
_P4	4-vector containing momentum and energy (TLorentzVector).
BeamCombo_	Initial state beam photon for a given combo.
Proton_	Final state proton for a given combo.
PiPlus_	Final state π^+ for a given combo.
PiMinus_	Final state π^- for a given combo.
Photon1_	First final state photon for a given combo. Subsequent photons are labeled Photon2_, Photon3_, ...
PiZero_	Final state π^0 for a given combo if the mass was constrained in by the kinematic fitter. Other particles, such as Eta, Kshort, and Lambda can also be specified.

Note, in the above example the π^0 is not a final state particle, as it does decay into photons and only those are detected directly. Therefore, such a particle is only explicitly listed as a separate particle in the ROOT tree if its mass was constrained by the kinematic fit to a fixed value, usually its mass as given by the PDG value.

In order to access the variables in the tree, one must use the above mentioned wrappers and their methods. The detailed functionality and implementations of all these wrapper methods can be found in DBeamParticle.h, DChargedTrackHypothesis.h, DNeutralParticleHypothesis.h, and DParticleCombo.h located in gluex_root_analysis library. These methods are discussed in the following in more detail.

3.0.1 DBeamParticle.h

Most of the methods needed by the incoming photon are inherited from DKinematicData.h

PID & KINEMATIC DATA

Get_PID() Returns type Particle_t defined in halld_recon/src/libraries/include/particleType.h

Get_P4() Get the 4-momentum of this particle

Get_X4() Get the 4-vector position (time and position) of this particle at the production vertex from the kinematic fit

Get_P4_Measured() Same as above

Get_X4_Measured() Same as above but the measured quantities, see example code above using the wrapper dComboBeamWr to get the vertex 4-vector.

Note that since the kinematic fitter does not change the 4-momentum of the beam photon _P4() and _P4_Measured() are the same in for the incident beam particle! The vertex position still can change though.

3.0.2 DChargedTrackHypothesis.h

Note that this class does inherit from DKinematicData.h and as such has the above listed methods as well the following ones:

IDENTIFIERS

Get_TrackID() Each physical particle has its own identifier but is fit by multiple mass hypotheses, refers back to the parent particle, returns int.

Get_ThrownIndex() Array index of thrown particle if match found, returns int (-1 otherwise). MC data only.

Get_ID() Same as Get_TrackID()

TRACKING INFO

Get_ChiSq_Tracking() The χ^2 from the charged particle fit (Kalman filter), returns float

Get_NDF_Tracking() The number of degrees of freedom in the fit, equal to the number of hits - 5, returns uint

Get_ChiSq_DCdEdx() The χ^2 between the measured energy loss in the drift chamber and the value expected for the given mass hypotheses, where the most appropriate value between the CDC and FDC dE/dx's is used, returns float

Get_NDF_DCdEdx() The number of degrees of freedom in the fit, returns uint

Get_dEdx_CDC() The calculated energy lose in the CDC, returns float

Get_dEdx_FDC() The calculated energy loss in the FDC, returns float

TIMING INFO

Get_HitTime() The time of the hit in whatever timing detector the charged particle is matched to, returns float

Get_RFDeltaTVar() The variance of the Δt_{RF} variable using the above hit time, returns float

Get_Beta_Timing() The calculated β from the above hit time, returns float

Get_ChiSq_Timing() The corresponding χ^2 from the above hit time for the given mass hypothesis, returns float

Get_NDF_Timing() The number of degrees of freedom used in this χ^2, returns uint

Get_ConfidenceLevel_Timing() The corresponding confidence level, returns float

Get_Beta_Timing_Measured() The same as above, but using the measured momentum, returns float

Get_ChiSq_Timing_Measured() The same as above, but using the measured momentum, returns float

idenceLevel_Timing_Measured() The same as above, but using the measured momentum, returns float

Get_Detector_System_Timing() The detector in which the matched timing hit was measured (BCAL/FCAL/SC/TOF), returns DetectorSy

HIT ENERGY

Get_dEdx_TOF() Measured energy loss in the TOF paddles, returns float

Get_dEdx_ST() Measured energy loss in the Start Counter paddles, returns float

Get_Energy_BCAL() Energy of a matched BCAL shower, returns float

Get_Energy_BCALPreshower() Energy deposited in the first layer of a matched BCAL shower, returns float

Get_Energy_BCALLayer2() Energy deposited in the second layer of a matched BCAL shower, returns float

Get_Energy_BCALLayer3() Energy deposited in the third layer of a matched BCAL shower, returns float

Get_Energy_BCALLayer4() Energy deposited in the fourth layer of a matched BCAL shower, returns float

Get_SigLong_BCAL() The r.m.s. deviation (?) of the BCAL shower energy in the logitudinal direction, returns float

Get_SigTheta_BCAL() The r.m.s. deviation (?) of the BCAL shower energy in the polar angle, returns float

Get_SigTrans_BCAL() The r.m.s. deviation (?) of the BCAL shower energy in the transverse direction, returns float

Get_RMSTime_BCAL() The r.m.s. deviation (?) of the BCAL shower time, returns float

Get_Energy_FCAL() Energy of a matched FCAL shower, returns float

Get_E1E9_FCAL() The $E1/E9$ ratio for the matched FCAL shower, returns float

Get_E9E25_FCAL() The $E9/E25$ ratio for the matched FCAL shower, returns float

Get_SumU_FCAL() The SumU value for the matched FCAL shower, returns float

Get_SumV_FCAL() The SumV value for the matched FCAL shower, returns float

SHOWER MATCH

Get_TrackBCAL_DeltaPhi() Difference in azimuthal positions of the shower and matched track along the face of the BCAL, returns float, 999.0 if not matched, units are radians

Get_TrackBCAL_DeltaZ() Difference in z positions of the shower and matched track along the face of the BCAL, returns float, 999.0 if not matched

Get_TrackFCAL_DOCA() Difference in positions of the shower and matched track along the face of the FCAL, returns float, 999.0 if not matched

DIRC INFO

Get$_{T}rack_{N}umPhotons_{D}IRC()$ Number of DIRC photons associated with the track, returns float, 999.0 if no DIRC info

Get$_{T}rack_{T}hetaC_{D}IRC()$ Measured Cherenkov angle, returns float, 999.0 if no DIRC info

Get$_{T}rack_{L}ele_{D}IRC()$ Measured Likelihood for the electron hypothesis, returns float, 999.0 if no info

Get$_{T}rack_{L}pi_{D}IRC()$ Measured Likelihood for the pion hypothesis, returns float, 999.0 if no info

Get$_{T}rack_{L}k_{D}IRC()$ Measured Likelihood for the kaon hypothesis, returns float, 999.0 if no info

Get$_{T}rack_{L}p_{D}IRC()$ Measured Likelihood for the proton hypothesis, returns float, 999.0 if no info

3.0.3 DNeutralParticleHypothesis.h

Note that this class does inherit from DKinematicData.h and as such has those methods as well the following ones:

IDENTIFIERS

Get_NeutralID() Each physical particle has its own number, returns int.

Get_ThrownIndex() Array index of thrown particle if match found, returns int (-1 otherwise). MC data only.

Get_ID() Same as Get_NeutralID()

TIMING

Get_HitTime() The reconstructed shower time, returns float

Get_RFDeltaTVar() The variance of the Δt_{RF} variable using the above shower time, returns float

Get_Beta_Timing() The calculated β from the above hit time, returns float

Get_ChiSq_Timing() The corresponding χ^2 from the above hit time for the given mass hypothesis, returns float

Get_NDF_Timing() The number of degrees of freedom used in this χ^2, returns uint

Get_ConfidenceLevel_Timing() The corresponding confidence level, returns float

Get_Beta_Timing_Measured() The same as above, but using the measured momentum, returns float

Get_ChiSq_Timing_Measured() The same as above, but using the measured momentum, returns float

idenceLevel_Timing_Measured() The same as above, but using the measured momentum, returns float

Get_Detector_System_Timing() The detector in which the matched timing hit was measured (BCAL/FCAL/SC/TOF), returns DetectorSy

SHOWER INFO

Get_Energy_BCAL() Energy of the BCAL shower, returns float

Get_Energy_BCALPreshower() Energy deposited in the first layer of the BCAL shower, returns float

Get_Energy_BCALLayer2() Energy deposited in the second layer of the BCAL shower, returns float

Get_Energy_BCALLayer3() Energy deposited in the third layer of the BCAL shower, returns float

Get_Energy_BCALLayer4() Energy deposited in the fourth layer of the BCAL shower, returns float

Get_SigLong_BCAL() The r.m.s. deviation (?) of the BCAL shower energy in the logitudinal direction, returns float

Get_SigTheta_BCAL() The r.m.s. deviation (?) of the BCAL shower energy in the polar angle, returns float

Get_SigTrans_BCAL() The r.m.s. deviation (?) of the BCAL shower energy in the transverse direction, returns float

Get_RMSTime_BCAL() The r.m.s. deviation (?) of the BCAL shower time, returns float

Get_Energy_FCAL() Energy of a FCAL shower, returns float

Get_E1E9_FCAL() The $E1/E9$ ratio for the FCAL shower, returns float

Get_E9E25_FCAL() The $E9/E25$ ratio for the FCAL shower, returns float

Get_SumU_FCAL() The SumU value for the FCAL shower, returns float

Get_SumV_FCAL() The SumV value for the FCAL shower, returns float

SHOWER MATCH

Get_TrackBCAL_DeltaPhi() Difference in azimuthal positions of the shower and matched track along the face of the BCAL, returns float, 999.0 if not matched, units are radians

Get_TrackBCAL_DeltaZ() Difference in z positions of the shower and matched track along the face of the BCAL, returns float, 999.0 if not matched

Get_TrackFCAL_DOCA() Difference in positions of the shower and matched track along the face of the FCAL, returns float, 999.0 if not matched

As an important reminder note that in order to make the Energy_BCALLayerX data available in the root tree the reaction filter requires to be run with the following command line parameter:
-PANALYSIS:BCAL_VERBOSE_ROOT_OUTPUT=1
and similarly for FCAL if the E1E9, E9E25, SumU and SumV data should be in the root tree the following command line parameter is required on the command line when running the reaction filter:
-PANALYSIS:FCAL_VERBOSE_ROOT_OUTPUT=1

3.0.4 DParticleCombo.h

SHOWER MATCH

Get_RFTime_Measured() RF bucket time for the given combination, measured at the center of the target, returns float.

Get_RFTime() Same as above, returns float.

TARGET

Get_TargetPID() Particle type of target, returns Particle_t.

Get_TargetCenter() Position of target center, returns TVector3.

KINFIT

Get_ChiSq_KinFit() χ^2 of kinematic fit of this combo to the assumed reaction, returns float.

Get_NDF_KinFit() Number of degrees of freedom for the above kinematic fit, returns float.

Get_ConfidenceLevel_KinFit() Confidence level determined from χ^2 and number of degrees of freedom, returns float.

UNUSED ENERGY

Get_Energy_UnusedShowers() Total energy of unmatched showers that are in time with the combo, but are not part of the combo, returns float.

UNUSED TRACKS

Get_NumUnusedTracks() Number of charged particle tracks in the event and not used in the combo, returns UChar_t.

Get_NumUnusedShowers() Number of unmatched showers in the event and not used in the combo, returns UChar_t.

EVENT INFO

Get_RunNumber() Run number of the event, returns UInt_t.

Get_EventNumber() Event number of the event, returns ULong64_t.

Get_L1TriggerBits() Trigger bit information, returns UInt_t.

Get_MCWeight() Optional event weight factor for MC events, returns float. =

3.0.5 DParticleComboStep.h

Get_X4() Vertex position and time for this step of the reaction, returns TLorentzVector.