Documentation to the gluex-root-analysis software

The GlueX bunch

April 26, 2019

Abstract

The *GlueX* root analysis software is used to analyze a gluex root tree based on the *DSelector* software idea. 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 these information into a REST-file in hddm format. Using the *Reaction Filter* software package will analyze these data based on the event reaction supplied by the analyzer. The output of *Reaction Filter* is a root file which can be read by the *DSelector* software package and represents the base for the a *GlueX* root analysis that is the subject of this document. In the following it is assumed you have setup up a working *GlueX* environment.

1 Introduction

The *GlueX* root analysis software package provides a few binaries that allow the user to generate a basic framework of *DSelector* program files. The location of these binaries is defined by the environment variable `ROOT_ANALYSIS_HOME`. An example of how to use one of the provide binaries is shown below:

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

where `program-name` is the label of your choice. This binary example assumes single threaded analysis only. 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.

ADD TEXT FOR USAGE OF MakePROOFPackage AND tree_to_amptools....

2 DSelector

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 software package library and provides all necessary methodes to read the root tree. The `DSelector` class does inherit from the root `TSelector` class that provides the methods `Init()`, `Process()` and `Finalize()` that form the base of the generated `.C` file.

The *DSelector* 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, the `++` at the end of the file forces the compiler to recompile the code regardless of the existence of an already compiled code.

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

```
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 `Init(TTree *locTree)` Method

Contrary to its name this method can and most likely will be called more than once. In particular when using `TChain` with more than one root file or when using PROOF in the context of multi-threading. This method is inherited from 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 past this part of the code. This includes any definitions of analysis-action and cut-actions as well as custom branches for an output root tree which can be done as a main tree or a flat tree.

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

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

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

These wrapper objects are instances of classes as shown in the above example and are pointers to the corresponding class instances the represent the particle for a given combo. 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 get access to the Lorentz 4-vector for the 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.

HERE ADD TEXT TO EXMPLAIN the concept of Step0 Step1 ... ect. BASED ON THE EXAMPLE ABOVE.

Each final state particle is either charged or neutral and is represented by a wrapper as either of type `DNeutralParticleHypothesis` 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. 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.

2.1.1 Histogram definitions

Histograms can be defined easily as in any root script. The user will need to define histograms as needed for his/her analysis.

2.1.2 AnalysisAction

NEED TEXT TO EXPLAIN

2.1.3 CutAction

NEED TEXT TO EXPLAIN

2.2 `Process(Long64_t locEntry)` Method

This is the main event loop called for each event. 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 fro the cern root library class `TSelector` Processing stops when this function returns kFALSE. This method is overridden by the HallD `DSelector` class and calls the method `Get_Entry()` defined in the class `DTreeInterface` and reads the event data from file into memory. Remember that in case `TChain` is used and a new tree file is opened the method `Init()` will be executed again first.

The second step is to prepare the analysis action:

```
/*** 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()
```

In the following some examples are suggested on how to prepare for uniqueness test but crucial part of the explanation of what is really meant for the analyzer to do is missing.

NEED MORE/CLEAR TEXT HERE. WHAT IS MEANT BY FOR EXAMPLE: FILL CUSTOM OUTPUT BRANCHES ?????

2.2.1 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 photo with the final state particle tracks. Each additional beam photon constitutes a different combo. The same is true for an additional charged track that passes the basic timing cuts and the kinematic fitter did converge based on the reaction requirements using this track in the final state.

```
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 COMMENT IS NEED HERE WHAT IsComboCut() REALLY MEANS, AND WHAT THE CONSEQUENCES ARE?

All final state particle tracks are required to have a vertex time that is in line with the RF beam bunch time. The initial beam photon timing can be either in time with the RF beam bunch or out of time (side band). In case of in time beam photons an event weight of 1 can be assigned while in case of out of time beam photons a weight of 1/n can be assigned where n is the number of side beam bunches that are considered in the analysis. The RF time is determined from the beam photon time using the `dComboBeamWrapper()` 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 in the analysis (tree) leading to a 1/8 weight factor.

```
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.){
   Weight = -0.125;
}

// the weight then can be used to fill a histogram with the weight
// thereby automatically subtract accidental background
.....
```

All the quantities in the root file are accessed using some `wrapper` method defined in your `DSelector` version. The variable names in the root tree have distinct qualifiers to help indicating its meaning and are explained here in more detail.

Root Tree Variables

`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`	Reconstructed measured quantity as determined by the track 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.
`PiZero_`	Final state π^0 for a given combo if the mass was constrained in by the kinematic fitter.

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. Therefor, 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 get access to these variables in the tree you are going to 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` and `DNeutralParticleHypothesis.h` located in `gluex_root_analysis` library. These methods are discussed in the following in more detail.

2.2.2 `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 based on the kinematic fitter

`Get_X4()` Get the 4-vector position of this particle based on the kinematic fitter

`Get_P4_Measured()` Same as above but the measured quantities

`Get_PX_Measured()` Same as above but the measured quantities, see example code above using the wrapper `dComboBeamWrapper` 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.

2.2.3 `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 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_TrackID()`

TRACKING INFO

`Get_NDF_Tracking()` Returns uint

`Get_ChiSq_Tracking()` Returns float

`Get_NDF_DCdEdx()` Returns uint

`Get_ChiSq_DCdEdx()` Returns float

`Get_dEdx_CDC()` Returns float

`Get_dEdx_FDC()` Returns float

TIMING INFO

`Get_HitTime()` Returns float

`Get_RFDeltaTVar()` Returns float

`Get_NDF_Timing()` Returns uint

`Get_Beta_Timing()` Returns float

Get_ChiSq_Timing() Returns float

Get_ConfidenceLevel_Timing() Returns float

Get_Beta_Timing_Measured() Returns float

Get_ChiSq_Timing_Measured() Returns float

idenceLevel_Timing_Measured() Returns float

Get_Detector_System_Timing() Returns `DetectorSystem_t`

HIT ENERGY

Get_dEdx_TOF() Returns float

Get_dEdx_ST() Returns float

Get_Energy_BCAL() Returns float

Get_Energy_BCALPreshower() Returns float

Get_Energy_BCALLayer2() Returns float

Get_Energy_BCALLayer3() Returns float

Get_Energy_BCALLayer4() Returns float

Get_SigLong_BCAL() Returns float

Get_SigTheta_BCAL() Returns float

Get_SigTrans_BCAL() Returns float

Get_RMSTime_BCAL() Returns float

Get_Energy_FCAL() Returns float

Get_E1E9_FCAL() Returns float

Get_E9E25_FCAL() Returns float

Get_SumU_FCAL() Returns float

Get_SumV_FCAL() Returns float

SHOWER MATCH

Get_TrackBCAL_DeltaPhi() Returns float, 999.0 if not matched, units are radians

Get_TrackBCAL_DeltaZ() Returns float, 999.0 if not matched

Get_TrackFCAL_DOCA() Returns float, 999.0 if not matched

2.2.4 `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()	Returns float, timing in order of preference: BCAL/TOF/FCAL/ST
Get_PhotonRFDeltaTVar()	Returns float, variance of `X4_Measured.T() - RFTime`
Get_NDF_Timing()	Returns uint
Get_Beta_Timing()	Returns float, is kinfit if kinfit, else is measured
Get_ChiSq_Timing()	Returns float, is kinfit if kinfit, else is measured
Get_ConfidenceLevel_Timing()	Returns float, is kinfit if kinfit, else is measured
Get_Beta_Timing_Measured()	Returns float
Get_ChiSq_Timing_Measured()	Returns float
idenceLevel_Timing_Measured()	Returns float, is kinfit if kinfit, else is measured
Get_Detector_System_Timing()	Returns `DetectorSystem_t`
	SHOWER INFO
Get_Shower_Quality()	Returns float, value between 0 and 1, recommended cut > 0.5 for photons.
Get_Energy_BCAL()	Returns float, shower energy in units of GeV
Get_Energy_BCALPreshower()	Returns float, preshower energy in units of GeV
Get_Energy_BCALLayer2()	Returns float
Get_Energy_BCALLayer3()	Returns float
Get_Energy_BCALLayer4()	Returns float
Get_SigLong_BCAL()	Returns float
Get_SigTheta_BCAL()	Returns float
Get_SigTrans_BCAL()	Returns float
Get_RMSTime_BCAL()	Returns float
Get_Energy_FCAL()	Returns float
Get_E1E9_FCAL()	Returns float
Get_E9E25_FCAL()	Returns float
Get_SumU_FCAL()	Returns float
Get_SumV_FCAL()	Returns float
Get_X4_Shower()	Returns TLorentzVector, location of the shower in the calorimeter
	TRACK MATCHING
Get_TrackBCAL_DeltaPhi()	Returns float, 999.0 if not matched, units are radians
Get_TrackBCAL_DeltaZ()	Returns float, 999.0 if not matched
Get_TrackFCAL_DOCA()	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`
Equipped with these methods 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 the 4-vectors based on the measured quantities or using the kinematic fitter results also in the example is shown how to get the shower quality factor and for a neutral particle 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_M = dComboBeamWrapper->Get_P4_Measured();
TLorentzVector locPiPlusP4_M = dPiPlusWrapper->Get_P4_Measured();
TLorentzVector locPiMinusP4_M = dPiMinusWrapper->Get_P4_Measured();
TLorentzVector locProtonP4_M = dProtonWrapper->Get_P4_Measured();
TLorentzVector locPhoton1P4_M = dPhoton1Wrapper->Get_P4_Measured();
TLorentzVector locPhoton2P4_M = 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();
```