ALICE resonance analysis package documentation

A. Pulvirenti

alberto.pulvirenti@ct.infn.it

1 Introduction

This small guide will drive the user through the resonance analysis package (we'll call it RSN), in order to learn how to configure an analysis with it and how to personalize some of its aspects. A slightly more advanced reference guide will also be provided at the end; it is of interest especially to all people interested in contributing to the package development, in order to go in a major detail through the code.

The purpose of the package is to set up an AnalysisTask object for resonance analysis. Then, all of its objects aim at configuring it properly for a specific resonance study. Keeping in mind this main purpose, we will explain how to configure a typical analysis, and this will give the possibility to introduce the user to all package classes he will have to manage directly. This will be done just going through a typical configuration macro, and discussing its lines.

This guide assumes that the reader roughly knows how a typical aliroot AnalysisTask object should be initialized, so we will not spend much time on its general aspects (how to configure input handlers and output containers, how to setup the AnalysisManager etc.). Anyway, in the`PWG2/RESONANCES/macros/test` path (inside the aliroot soruce code tree), it is available a set of macroes which can be used to steer an analysis from an aliroot session.

We will then give the initial description following a building scheme where we go through all aspects of analysis initialization. The package is based on four independent pieces which all contribute to the final analysis: the interface classes which are used to uniformize the way it accesses the informations taken from whatever kind of input source (namely, ESD, AOD or MC); the work-flow classes which just execute the loops on the events and build up the output objects returned by the analysis task; the cut classes which are the fundamental step for track/pair selection and can determine the difference between one analysis and another the output classes which implement the computation of all output values of interest It must be specified that, in its current version, the package is built essentially in order to make computations on pairs of particles, when they are accepted as a candidate resonance decay. This document will give a panoramic of all the classes which are currently available in the package, specifying what they are implemented for. Anyway, not all of the package classes need to be perfectly known by the user, since most of them implement just the necessary internal steps to loop on the event and produce the output. Whenever a class enters the necessary configuration and needs to be managed by the user, more details will be given to it. Finally, we must mention that the package is currently built in such a way that it allows an advanced user to customize some aspects of the analysis.

```
//
// This function configures the entire task for all resonances the user is interested in.
// This is done by creating all configuration objects which are defined in the package.
//
// Generally speaking, one has to define the following objects for each resonance:
//
//  1 - an AliRsnPairDef to define the resonance decay channel to be studied
//  2 - an AliRsnPair{Ntuple|Functions} where the output is stored
//  3 - one or more AliRsnCut objects to define track selections
//      which will have then to be organized into AliRsnCutSet objects
//  4 - an AliRsnCutManager to include all cuts to be applied (see point 3)
//  5 - definitions to build the TNtuple or histograms which are returned
//
// The return value is used to know if the configuration was successful
//
Bool_t RsnConfigTask(AliRsnAnalysisSE* &task, const char *dataLabel)
{
  // for safety, return if no task is passed
```

```
if (!task)
{
  Error("ConfigTaskRsn", "Task not found");
  return kFALSE;
}

// interpret the useful information from second argument
TString strDataLabel(dataLabel);
Bool_t isESD   = strDataLabel.Contains("ESD");
Bool_t isAOD   = strDataLabel.Contains("AOD");
Bool_t isSim   = strDataLabel.Contains("sim");
Bool_t isData  = strDataLabel.Contains("data");
Bool_t isPass1 = strDataLabel.Contains("pass1");
Bool_t isPass2 = strDataLabel.Contains("pass2");

//
// -- Set cuts for events (applied to all analyses) ------------------------------------------------
//

// primary vertex range
AliRsnCutPrimaryVertex *cutVertex   = new AliRsnCutPrimaryVertex("cutVertex", 10.0, 0, kFALSE);
AliRsnCutSet           *cutSetEvent = new AliRsnCutSet("eventCuts", AliRsnCut::kEvent);
//cutSetEvent->AddCut(cutVertex);
//cutSetEvent->SetCutScheme("cutVertex");
//task->SetEventCuts(cutSetEvent);

//
// -- Setup pairs ----------------------------------------------------------------------------------
//

// decay channels
AliRsnPairDef         *pairDefpm = new AliRsnPairDef(AliPID::kKaon, '+', AliPID::kKaon, '-', 333, 1.019455);

// computation objects
AliRsnPairFunctions   *pairPMhist = new AliRsnPairFunctions("pairPMHist", pairDefpm);
AliRsnPairNtuple      *pairPMntp  = new AliRsnPairNtuple   ("pairPMNtp" , pairDefpm);

//
// -- Setup cuts -----------------------------------------------------------------------------------
//

// -- track cut --
// --> global cuts for 2010 analysis
AliRsnCutESD2010 *cuts2010 = new AliRsnCutESD2010("cuts2010");
// ----> set the flag for sim/data management
cuts2010->SetMC(isSim);
// ----> require to check PID
cuts2010->SetCheckITS(kFALSE);
cuts2010->SetCheckTPC(kFALSE);
cuts2010->SetCheckTOF(kFALSE);
// ----> set TPC ranges and calibration
cuts2010->SetTPCrange(5.0, 3.0);
cuts2010->SetTPCpLimit(0.35);
cuts2010->SetITSband(4.0);
if (isSim) cuts2010->SetTPCpar(2.15898 / 50.0, 1.75295E1, 3.40030E-9, 1.96178, 3.91720);
else       cuts2010->SetTPCpar(1.41543 / 50.0, 2.63394E1, 5.0411E-11, 2.12543, 4.88663);
// ----> set standard quality cuts for TPC global tracks
//cuts2010->GetCutsTPC()->SetRequireTPCStandAlone(kTRUE); // require to have the projection at inner TPC wall
cuts2010->GetCutsTPC()->SetMinNClustersTPC(70);
cuts2010->GetCutsTPC()->SetMaxChi2PerClusterTPC(4);
cuts2010->GetCutsTPC()->SetAcceptKinkDaughters(kFALSE);
cuts2010->GetCutsTPC()->SetRequireTPCRefit(kTRUE);
cuts2010->GetCutsTPC()->SetRequireITSRefit(kTRUE);
cuts2010->GetCutsTPC()->SetClusterRequirementITS(AliESDtrackCuts::kSPD, AliESDtrackCuts::kAny);
cuts2010->GetCutsTPC()->SetMaxDCAToVertexXYPtDep("0.0350+0.0420/pt^0.9"); // DCA pt dependent: 7*(0.0050+0.0060/pt0.9)
cuts2010->GetCutsTPC()->SetMaxDCAToVertexZ(1e6); // disabled
cuts2010->GetCutsTPC()->SetDCAToVertex2D(kFALSE); // each DCA is checked separately
cuts2010->GetCutsTPC()->SetRequireSigmaToVertex(kFALSE);
// ----> set standard quality cuts for ITS standalone tracks
cuts2010->GetCutsITS()->SetRequireITSStandAlone(kTRUE, kTRUE);
cuts2010->GetCutsITS()->SetRequireITSRefit(kTRUE);
cuts2010->GetCutsITS()->SetMinNClustersITS(4);
cuts2010->GetCutsITS()->SetClusterRequirementITS(AliESDtrackCuts::kSPD, AliESDtrackCuts::kAny);
cuts2010->GetCutsITS()->SetMaxChi2PerClusterITS(1.);
cuts2010->GetCutsITS()->SetMaxDCAToVertexXYPtDep("0.0595+0.0182/pt^1.55"); // DCA pt dependent
cuts2010->GetCutsITS()->SetMaxDCAToVertexZ(1e6); // disabled
cuts2010->GetCutsITS()->SetDCAToVertex2D(kFALSE); // each DCA is checked separately
// ----> set the configuration for TOF PID checks
if (isData && (isPass1 || isPass2))
{
  cuts2010->SetTOFcalibrateESD(kTRUE);
  //if (isPass2) cuts2010->SetTOFcalibrateESD(kFALSE); // potrebbe anche essere kFALSE
  cuts2010->SetTOFcorrectTExp(kTRUE);
  cuts2010->SetTOFuseT0(kTRUE);
  cuts2010->SetTOFtuneMC(kFALSE);
  cuts2010->SetTOFresolution(100.0);
}
else if (isSim)
{
  cuts2010->SetTOFcalibrateESD(kFALSE);
  cuts2010->SetTOFcorrectTExp(kTRUE);
  cuts2010->SetTOFuseT0(kTRUE);
  cuts2010->SetTOFtuneMC(kTRUE);
  cuts2010->SetTOFresolution(100.0);
}

// -- tracks --> PID
AliRsnCutPID *cutPID = new AliRsnCutPID("cutPID", AliPID::kKaon, 0.0, kTRUE);

// cut sets
AliRsnCutSet *cutSetDaughterCommon = new AliRsnCutSet("commonDaughterCuts", AliRsnCut::kDaughter);

// --> add related cuts
```

```
  //cutSetDaughterCommon->AddCut(cuts2010);
  cutSetDaughterCommon->AddCut(cutPID);

  // --> define schemes
  cutSetDaughterCommon->SetCutScheme("cutPID");

  // cut managers
  // define a proper name for each mult bin, to avoid omonyme output histos
  pairPMhist->GetCutManager()->SetCommonDaughterCuts(cutSetDaughterCommon);
  pairPMntp ->GetCutManager()->SetCommonDaughterCuts(cutSetDaughterCommon);

  // function axes
  Double_t ybins[] = {-0.8, -0.7, -0.6, -0.5, 0.5, 0.6, 0.7, 0.8};
  AliRsnValue *axisIM   = new AliRsnValue("IM"  , AliRsnValue::kPairInvMass, 50,  0.9,  1.4);
  AliRsnValue *axisPt   = new AliRsnValue("PT"  , AliRsnValue::kPairPt,      0.0, 20.0, 0.1);
  AliRsnValue *axisY    = new AliRsnValue("Y"   , AliRsnValue::kPairY,       sizeof(ybins)/sizeof(ybins[0]), ybins);
  AliRsnValue *axisQinv = new AliRsnValue("QInv", AliRsnValue::kQInv,      100,  0.0, 10.0);

  // functions for TH1-like output
  AliRsnFunction *fcnPt    = new AliRsnFunction;
  // --> add axes
  fcnPt   ->AddAxis(axisIM);
  fcnPt   ->AddAxis(axisPt);
  fcnPt   ->AddAxis(axisY);
  fcnPt   ->AddAxis(axisQinv);

  // add functions to TH1-like output
  pairPMhist->AddFunction(fcnPt);
  //pairPMhist->SetOnlyTrue();

  // add values to TNtuple-like output
  pairPMntp->AddValue(axisIM);
  pairPMntp->AddValue(axisPt);
  pairPMntp->AddValue(axisY);
  pairPMntp->AddValue(axisQinv);

  // add everything to analysis manager
  task->GetAnalysisManager()->Add(pairPMhist);
  task->GetAnalysisManager()->Add(pairPMntp);

  return kTRUE;
}
```

2 Cuts

Cuts are implemented in the package through the **AliRsnCut** base class, which is the basic scheme from which all specific cut implementations must inherit. Its general structure is given here:

```
class AliRsnCut : public TNamed
{
  public:

    ...

    virtual Bool_t   IsSelected(TObject *obj1, TObject *obj2 = 0x0);

    ...

  protected:

    Bool_t  OkValue();
    Bool_t  OkRange();
    Bool_t  OkValueI();
    Bool_t  OkRangeI();
    Bool_t  OkValueD();
    Bool_t  OkRangeD();

    ...

    Int_t     fMinI;       // lower edge of INT range or ref. value for INT CUT
    Int_t     fMaxI;       // upper edge of INT range (not used for value cuts)
    Double_t  fMinD;       // lower edge of DOUBLE range or ref. value for INT CUT
    Double_t  fMaxD;       // upper edge of DOUBLE range (not used for value cuts)

    Int_t     fCutValueI;  // cut value INT
    Double_t  fCutValueD;  // cut value DOUBLE

    ...
};
```

The heart of the class definition is the **IsSelected()** function, which is called when the cut needs to be checked along the work-flow of the analysis and *must* be overloaded by each specific cut implementations inheriting from this base class. Due to this structure, a user can implement his personalized version of the cuts by creating a class inheriting from **AliRsnCut** and compiling it on the fly, and using it in the standard analysis.

Most cuts usually consist in checking that a value equals a given reference or stays inside a given range. Then, we provide directly in the base class the instruments to make these checks with integer or floating-point variables. One thing that needs to be considered here is that, for monitoring purposes, these value or range

check functions (**OkValueD()**, **OkValueI()**, **OkRangeD()** and **OkRangeI()**) do not accept arguments, since the value to be checked is expected to be stored in an apposite data member of the class itself (**fCutValueI** and **fCutValueD**). Then, all cuts which exploit this facility must first store the value to be checked in these data members and then call the corresponding range or value check function. The advantage of this structure is that the cut value can be kept in memory and, for example, monitored using debug messages.

Since usually a user does many checks and sometimes the way to link cuts together is not exactly their simple "and", then we introduced a strategy which allows to implement each single check separately, with the advantage of having simpler code, easier to debug. This strategy is based on the **AliRsnCutSet** class, which acts as a collector of cuts. All points where a cut check is done throughout the analysis work-flow uses object of this type, instead of accessing direcly to the single cuts. With this object a user has to do two things: first, add all cuts he wants to check; second, defining a logic by means of a string containing an expression where the names of the single cuts are combined using the C++ logic operators (and, or not). Both these steps are necessary, since if one just adds the cuts but does not define a logic, then no cut check is taken into account. On the other side, for ease of use, one can add cuts and not use them, since it is not mandatory to include all cut names inside the logic. Anywaym the user must keep in mind that all cuts added to a set are checked always, so, adding a lot of unused cuts could unnecessarily slow down the analysis execution.

The uppermost step of the cut structure is the **AliRsnCutManager**, which combines a set for each possible object to check.