The FLOW Analysis Package

a short manual

July 1, 2014

Redmer Alexander Bertens
(rbertens @ cern.ch)
with excerpts from other manuals, authors of those are mentioned in text

Contents

Chapter 1

Introduction

The ALICE flow package[a] contains most known flow analysis methods. The package itself consists of two parts

1. The 'tasks' library, which can be considered to be the ALICE interface to the package and takes care of e.g. track cuts, event cuts, etc;
2. The 'base' library, which is the core of the package and contains the actual implementation of flow analysis methods such as the scalar product method, Q-cumulant method, etc. This part of the package has no dependencies other than ROOT and can be used on any type of input data.

1.1 This manual

This manual is designed to get you started with using the flow package. It is written in the following way:

- Chapter 2 is designed to get you started on a short Monte Carlo example. In this example you will use the flow package to generate toy Monte Carlo events and analyze them;
- Chapter 3 describes the flow package itself in detail. This includes a brief discussion on the structure of the package, sections on track and event cuts, an explanation of some relevant code sections and ending with an example analysis of $v_2(p_t)$ of charged pions with the Q-cumulant method. Most of this chapter pertains to the 'tasks (the AliROOT)' part of the flow package (i.e. event cuts, track cuts, PID, etc), but it is also explained how to do flow analysis in ROOT only on a TTree;
- Chapter 4 gives an overview of the available flow analysis methods. For the theory behind the methods references to papers are given. Settings relevant to the specific implementation are given as well.
- Lastly, chapter 5 explains how the flow package can be put to use in more 'exotic' environments, such as an invariant mass method estimate of flow of rapidly decaying particles.

1.2 Disclaimer

What this manual is *not* designed for is letting the analyzer use the flow package as a 'black box'. It is supposed to be a starting point, to give an overview of the design of the software and point you to relevant classes, but in the end, the analyzer is responsible for understanding what is happening and using the software in a proper way. Configurations of the package which may work on a technical level (i.e. produce output) do not necessarily mean that the output is what you expect it to be! Always make sure that you understand what you are doing, and when in doubt, browse through the source code or consult an expert. The package is not a static entity, users are encouraged to make additions, be it track cuts, bug fixes, additional analysis methods, etc, etc. If you have suggestions, questions, commit requests, send an email to the flow-pag mailing list or to rbertens @ cern.

[a] The ALICE flow package is part of AliROOT, the ALICE extension of the ROOT framework, which can be obtained from http://git.cern.ch/pub/AliRoot. The flow package itself is located in the folder $ALICE_ROOT/PWG/FLOW/, where $ALICE_ROOT refers to the source directory of AliROOT.

Chapter 2

A Quick Start

We'll begin with a hands-on exercise in which you'll get acquainted with some aspects of the flow package in a few minutes. We'll do this by generating a few simple toy Monte Carlo events and performing a flow analysis on these simulated events without writing them (the events) to disk, a so called 'flow analysis on-the-fly'[a].

2.1 On the fly - getting started on a Toy MC

The steps which will be followed in this example will be the same as the steps we take when performing an analysis on data[b]:

1. Prepare your `(Ali)ROOT` session by loaded the necessary libraries
2. Create the analysis method objects
3. Initialize the methods (which creates their histograms)
4. Define track cuts
5. Create flow events, which is a container class holding all necessary information (e.g. tracks) for the flow analysis of an event (collision) and actually do the analysis
6. Finish the analysis, which will calculate the final v_n values
7. Write the results to an output file

In this Monte Carlo exercise, the flow event class will not receive data from a detector, but instead generate toy events itself.

We will now go through these step one-by-one. All the code that is used can also be found in the macro `runFlowOnTheFlyExample.C`[c].

1. To use the flow code the flow library needs to be loaded. In `AliROOT`:

```
gSystem->Load("libPWGflowBase");
```

   In `root` additional libraries need to be loaded:

```
gSystem->Load("libGeom");
gSystem->Load("libVMC");
gSystem->Load("libXMLIO");
gSystem->Load("libPhysics");
gSystem->Load("libPWGflowBase");
```

2. We need to instantiate the flow analysis methods which we want to use. In this example we will instantiate two methods: one which calculates the flow versus the Monte Carlo event plane (this our reference value: as the event plane orientation is known by this method, the v_2 value we retrieve should be equal to the input v_2 by definition) and as a second method the so called Q-cumulant analysis.

```
AliFlowAnalysisWithMCEventPlane *mcep = new AliFlowAnalysisWithMCEventPlane();
AliFlowAnalysisWithQCumulants *qc = new AliFlowAnalysisWithQCumulants();
```

[a] In this example the `AliFlowEventSimple` class will be used to generate toy events (which is described in detail in section 3). Another on-the-fly routine is available in the `AliFlowEventSimpleMakerOnTheFly`, the original on-the-fly manual for that class is reprinted in the appendix (see B) of this document.

[b] In data, some of these steps are actually taken care of by an analysis task, but this will be described in more detail in the next chapter.

[c] In aliroot, this macro can be found at
`$ALICE_ROOT/PWGCF/FLOW/Documentation/examples/manual/runFlowOnTheFlyExample`

3. Each of the methods needs to be initialized (e.g. to define the histograms):

```
mcep->Init();
qc->Init();
```

4. To define the particles we are going to use as Reference Particles (RP's, particles used for the **Q** vector) and the Particles Of Interest (POI's, the particles of which we calculate the differential flow) we have to define two track cut objects:

```
AliFlowTrackSimpleCuts *cutsRP = new AliFlowTrackSimpleCuts();
AliFlowTrackSimpleCuts *cutsPOI = new AliFlowTrackSimpleCuts();
cutsPOI->SetPtMin(0.2);
cutsPOI->SetPtMax(2.0);
```

Particles will be selected as either POI or RP depending on whether or not they pass these cuts.

5. Now we are ready to start the analysis. For a quick start we create a toy Monte Carlo event, tag the reference particles and particles of interest (which means that, if a particle passes the POI or RP cuts, it is flagged as 'POI' or 'RP') and pass it to the two flow methods.

Since we want to analyze more than one event, this step is performed in loop. First define the number of events that need to be created, their multiplicity, and a value v_2 value, which can either be supplied as a fixed number (no p_t dependence) of a function (to generate p_t differential flow[d]

```
Int_t nEvents = 1000; // generate 1000 events
Int_t mult = 2000;    // use track multiplicity of 2000
Double_t v2 = .05;    // 5 pct integrated flow
// or sample differential flow
TF1* diffv2 = new TF1("diffv2", "((x<1.)*(0.1/1.)*x+(x>=1.)*0.1)", 0., 20.);
```

Now we have all the ingredients to our first flow analysis

```
for(Int_t i=0; i<nEvents; i++) {
    // make an event with mult particles
    AliFlowEventSimple* flowevent = AliFlowEventSimple(mult,AliFlowEventSimple::kGenerate);
    // modify the tracks adding the flow value v2
    flowevent->AddV2(diffv2);
    // select the particles for the reference flow
    flowevent->TagRP(cutsRP);
    // select the particles for differential flow
    flowevent->TagPOI(cutsPOI);
    // do flow analysis with various methods:
    mcep->Make(flowevent);
    qc->Make(flowevent);
    // delete the event from memory
    delete flowevent;
}
```

6. To fill the histograms which contain the final results we have to call Finish for each method:

```
mcep->Finish();
qc->Finish();
```

7. This concludes the analysis and now we can write the results into a file. Two options for writing the input to a file are available:

- Create a new output file and write the output to this file

```
TFile *outputFile = new TFile("outputMCEPanalysis.root","RECREATE");
mcep->WriteHistograms();
TFile *outputFile = new TFile("outputQCanalysis.root","RECREATE");
qc->WriteHistograms();
```

Please note that this will create a new output file, and overwrite any existing file called `AnalysisResults.root`.

- To write the output of multiple analyses into sub-directories of one file, one can do the following:

```
TFile *outputFile = new TFile("AnalysisResults.root","RECREATE");
TDirectoryFile* dirQC = new TDiretoryFile("outputQCanalysis", "outputQCanalysis");
qc->WriteHistograms(dirQC);
TDirectoryFile* dirMCEP = new TDiretoryFile("outputMCEPanalysis", "outputMCEPanalysis");
mcep->WriteHistograms(dirMCEP);
```

[d] The on the fly event generator is not limited to the generation of the second harmonic v_2, but to get started, this is a nice example.

Note that `AnalysisResults.root` is the default name given to analyses in `AliROOT`. Many macros in `AliROOT` will expect a file `AnalyisResults.root` as input, so for most users it will be convenient to follow this convention.

When done with running the analysis, do not forget to write the file to disk by calling

```
TFile::Close(); // write the buffered file to disk
```

2.2 What is in the output file ?

Now we have written the results into a file, but what is in there?

Although the output of different flow analysis techniques might differ slightly as a result of their different approaches at estimating v_2, the output files containers are always constructed in a similar way.

2.2.1 AliFlowCommonHists - Output objects

Objects of two types are stored in the output of the flow analysis[e]

1. `AliFlowCommonHist`, which is a class that contains common histograms for the flow analysis (e.g. QA histograms and histograms that contain the analysis flags which were used). Depending on the type of flow analysis that was used, this object contains histograms from the following list:

```
  Bool_t     fBookOnlyBasic;        // book and fill only control histos needed for all methods
  TH1F*      fHistMultRP;           // multiplicity for RP selection
  TH1F*      fHistMultPOI;          // multiplicity for POI selection
  TH2F*      fHistMultPOIvsRP;      // multiplicity for POI versus RP
  TH1F*      fHistPtRP;             // pt distribution for RP selection
  TH1F*      fHistPtPOI;            // pt distribution for POI selection
  TH1F*      fHistPtSub0;           // pt distribution for subevent 0
  TH1F*      fHistPtSub1;           // pt distribution for subevent 1
  TH1F*      fHistPhiRP;            // phi distribution for RP selection
  TH1F*      fHistPhiPOI;           // phi distribution for POI selection
  TH1F*      fHistPhiSub0;          // phi distribution for subevent 0
  TH1F*      fHistPhiSub1;          // phi distribution for subevent 1
  TH1F*      fHistEtaRP;            // eta distribution for RP selection
  TH1F*      fHistEtaPOI;           // eta distribution for POI selection
  TH1F*      fHistEtaSub0;          // eta distribution for subevent 0
  TH1F*      fHistEtaSub1;          // eta distribution for subevent 1
  TH2F*      fHistPhiEtaRP;         // eta vs phi for RP selection
  TH2F*      fHistPhiEtaPOI;        // eta vs phi for POI selection
  TProfile* fHistProMeanPtperBin; // mean pt for each pt bin (for POI selection)
  TH2F*      fHistWeightvsPhi;      // particle weight vs particle phi
  TH1F*      fHistQ;                // Qvector distribution
  TH1F*      fHistAngleQ;           // distribution of angle of Q vector
  TH1F*      fHistAngleQSub0;       // distribution of angle of subevent 0 Q vector
  TH1F*      fHistAngleQSub1;       // distribution of angle of subevent 1 Q vector
  TProfile* fHarmonic;             // harmonic
  TProfile* fRefMultVsNoOfRPs;     // <reference multiplicity> versus # of RPs
  TH1F*      fHistRefMult;          // reference multiplicity distribution
  TH2F*      fHistMassPOI;          // mass distribution for POI selection
```

This information is from the header file of the AliFlowCommonHist object[f]

2. `AliFlowCommonHistResults` is an object designed to hold the common results of the flow analysis[g]. The possible common histograms stored in this object are

```
  TH1D* fHistIntFlow; // reference flow
  TH1D* fHistChi;     // resolution
  // RP = Reference Particles:
  TH1D* fHistIntFlowRP;      // integrated flow of RPs
  TH1D* fHistDiffFlowPtRP;   // differential flow (Pt) of RPs
  TH1D* fHistDiffFlowEtaRP; // differential flow (Eta) of RPs
  // POI = Particles Of Interest:
  TH1D* fHistIntFlowPOI;      // integrated flow of POIs
  TH1D* fHistDiffFlowPtPOI;   // differential flow (Pt) of POIs
  TH1D* fHistDiffFlowEtaPOI; // differential flow (Eta) of POIs
```

The titles of the histograms in the output object differ from the names of the pointers given in the two lists printed above, but the lists give an overview of what is available; the easiest way however of getting acquainted with where to find histograms in the output is browsing them in `ROOT's TBrowser` (see figure 2.2).

[e] Make sure that `libPWGflowBase.so` is loaded in your `(Ali)ROOT` session, otherwise these objects will be unknown.

[f] The headers of both output objects can be found in `$ALICE_ROOT/PWG/FLOW/Base/`.

[g] The word common here is used to indicate histograms that hold observables which are evaluated in all flow analysis methods. Specific analysis methods may however store additional histograms which are not covered in this list!

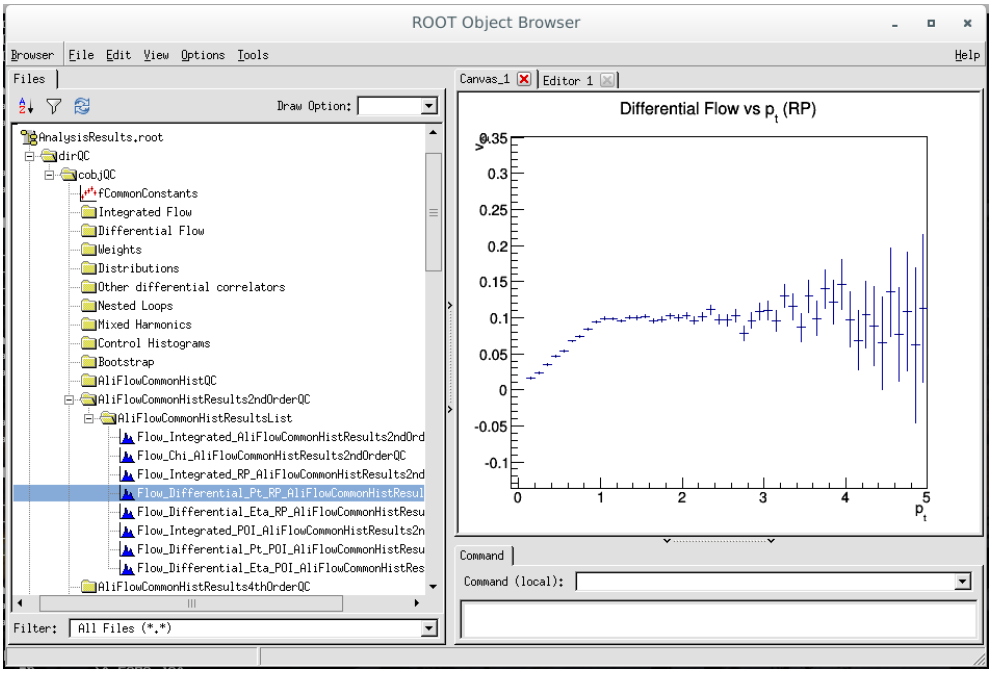

Figure 2.1: Example of output file opened in a TBrowser, results of differential v_2 analysis with second order Q-cumulant analysis are shown.

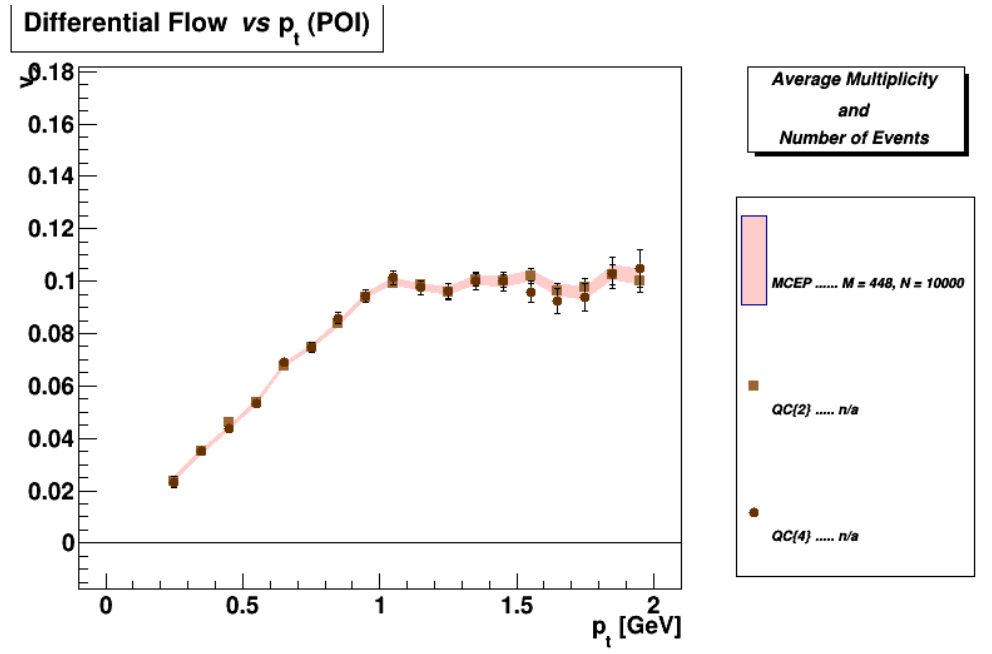

Figure 2.2: Example of inspecting the output file of the on the fly analysis with the `compareFlowResults.C` macro.

```
new TBrowser();
```

The `AliFlowCommonHist` and `AliFlowCommonHistResults` classes are derived from the generic `TNamed` `ROOT` object and can be written to a `ROOT` file. The flow analysis tasks will, as output, write the complete `AliFlowCommonHist` and `AliFlowCommonHistResults` objects to file at the end of an analysis. To read the content of these objects, the `libPWGflowBase` library must be loaded in your `ROOT` session.

Comparing flow results

A convenient way of comparing the results of the different flow analysis strategies that have been used is invoking the macro `compareFlowResults.C`[h]. This macro will read the analysis output file `AnalysisResults.root`, extract the requested results from it and plot them. For a full overview of what can be done with the macro, the reader is referred to the macro itself and its ample documentation. To run the macro on the data-set that we have just generated, simply do

```
.L compareFlowResults.C
compareFlowResults(TSring(""))  // the empty suffix indicates on the fly events
```

[h] `$ALICE_ROOT/PWGCF/FLOW/macros/compareFlowResults.C`

Chapter 3

The Program

The basic idea behind the flow package is that from whatever input you have, a *flow event* is constructed, which is then passed to one or more flow analysis methods (e.g. the scalar product method or Q-cumulant method). The flow event is a collection of *flow tracks*, which are simple objects carrying only the kinematic information that is necessary to do flow analysis. By setting up the flow package in this way, the flow analysis methods can analyze input from various sources, be it ALICE data, Monte Carlo events, STAR data, etc, etc, as long as the flow event is properly filled . This might all sound a bit abstract at this point; this chapter however will explain all details and relevant classes in detail. For those who are impatient and prefer seeing the flow package in action, section 3.2.7 gives a step-by-step example of doing a $\pi^{\pm}$ v_2 analysis in the `AliROOT` analysis framework.

3.1 Overview

Figure 3.1 gives a simple schematic representation of the flow package. Input events (in the case of the figure this is either ESDs or AODs) pass a set of event cuts (the common cuts) and are then converted to a flow event (stored as an `AliFlowEventSimple` object). This flow event holds a collection of flow tracks (`AliFlowTrackSimple` objects) which are passed to flow analysis methods. The only steps of this flow chart which depend on `AliROOT` libraries are the ones handling `ALICE` data types (the ESDs or AODs). The rest of the analysis chain (the `AliFlowEventSimle` and the analysis methods) have no specific `AliROOT` dependence and are just simple `c++` objects. Therefore, the flow package is split into two libraries

libPWGflowBase The base library, which has no specific `AliROOT` dependencies. This library holds objects such as the `AliFlowEventSimple` and `AliFlowTrackSimple`, and analysis methods classes. The analysis methods classes follow the naming scheme: `AliFlowAnalysisWith`$*$ where $*$ denotes a specific analysis method. All classes which end up in the libPWGflowBase.so shared object can be found in `$ALICE_ROOT/PWG/FLOW/Base`;

libPWGflowTasks The tasks library, which has specific `AliROOT` dependencies. Contrary to what the name suggests, this library does not just hold tasks, but actually comprises all classes of the flow package which need to include `AliROOT` specific classes. This ranges from classes to read the AOD or ESD input data (important examples are the `AliFlowEvent` and `AliFlowTrackCuts`, which will be discussed later on in this chapter) and the `AliAnalysisTask`$*$ classes, which are analysis tasks, derived from `AliAnalysisTaskSE` which can be used in the `AliROOT` analysis framework and are actually just interface classes to the underlying flow analysis methods of libPWGflowBase. The classes which are bundled into the libPWGflowTasks.so shared object can be found in `$ALICE_ROOT/PWG/FLOW/Tasks`;

Some tools, such as the flow event or track cuts, have a 'base' component which name ends with the suffix 'simple', and an 'tasks' (`AliROOT`) component which does not have this suffix. The 'tasks' class in these cases inherits from the 'base' class.

Every flow analysis in the flow package starts with the flow event. As mentioned earlier, the flow event is a simple container class which holds a collection of flow tracks, which are in turn fed to the flow analysis methods. In the next section it will be explained how the flow event can be filled with `ALICE` data in the `AliROOT` analysis framework. The section after that will explain how the flow event can be filled with *any* type of data using just `ROOT`

3.2 Analysis in the ALICE analysis framework

In this section, you will see how a flow analysis can be performed in the `AliROOT` analysis framework.

3.2.1 Input data

Before passing the flow event to the flow analysis methods, it needs to be filled with a set of flow tracks. In general, a distinction is made between *reference particles* (or *RP's*), which are particles that are used to build the **Q** vector(s), and

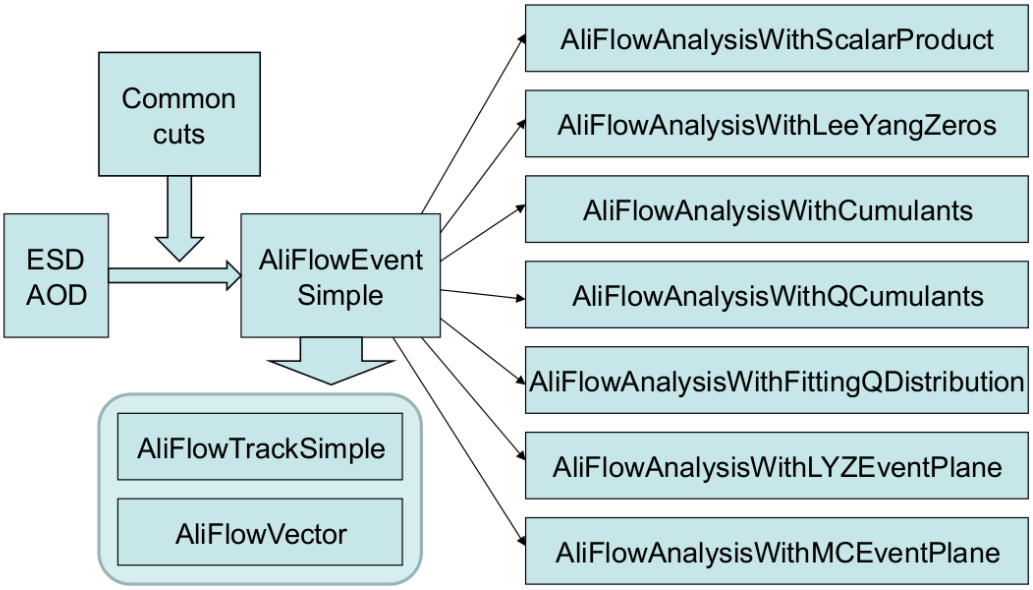

Figure 3.1: Schematic presentation of the organization of the flow package. Input, which can come from any kind of source, is converted to a generic `AliFlowEventSimple` object, which in turn is passed to the analysis methods.

particles of interest (or *POI's*), which are the particles of which you'll calculate the differential flow. The flow event and the flow analysis methods are designed to keep track of which flow tracks are POI's, RP's (or even both at the same time), which is important to avoid auto-correlation effects which can distort the v_n measurement. The user of the flow package however is responsible for properly setting up the analysis!

The flow event can be filled with input from many sources. In the second chapter of this manual, a simple method has been shown where the flow event (the `AliFlowEventSimple` object) fills itself by generating a set of Monte Carlo tracks by sampling kinematic variables from supplied p.d.f.'s. Using this method is a very effective tool for testing and developing new flow analysis methods (if you generate events with a certain $v_2(p_t)$ and then retrieve the same $v_2(p_t)$ from your flow analysis method, you can use that as a tool to proof the validation of your analysis method) but if you want to do a data analysis, a somewhat more advanced - but not difficult - approach is necessary.

Filling a flow event from data can be performed either 'by-hand' (which is covered in section 5 on more exotic analyses), but the most commonly used method of filling a flow event in the `AliROOT` analysis framework is using the dedicated task `AliAnalysisTaskFlowEvent`.

The idea behind this is the following:

1. Setup the `AliAnalysisTaskFlowEvent` task to receive input events (e.g. `AODs`, `ESDs`, `MC`, ...;
2. Define two sets of track selection criteria (colloquially referred to as *track cuts*), one for POI's and one for RP's;
3. Pass these two sets of track cuts to the `AliAnalysisTaskFlowEvent`;
4. The `AliAnalysisTaskFlowEvent` will convert the tracks of each input event to a set of `AliFlowSimpleTracks`. Depending on whether or not a track passes the track selection for POI's or RP's, the `AliFlowSimpleTrack` is labeled as a POI or RP (or both. In the case where a track does not meet any of the track selection criteria, it is omitted from the `AliFlowSimpleTrack` collection and not added to the flow event);
5. All the `AliFlowSimpleTracks` are added to the flow event which is passed to the flow analysis methods.

3.2.2 Event selection

When using the `AliAnalysisTaskFlowEvent` task to create your flow event, the `AliAnalysisTaskFlowEvent` task is responsible for ensuring that only good quality tracks enter into your analysis by making sensible track selections. The first step however at safeguarding track quality is making sure that the events that are accepted by `AliAnalysisTaskFlowEvent` pass sane event selection criteria.

Trigger selection

A certain combination a of detector signals (a *trigger*) is required for an event to be written to storage. Different types of analyses might require different types of events, and hence, different types of triggers.

You can set a trigger by calling

```
1 AliAnalysisTaskFlowEvent::SelectCollisionCandidates(UInt_t offlineTriggerMask);
```

where `offlineTriggerMask` is the trigger mask corresponding to the desired trigger. A list of all available triggers, with a short descrption, can be found in the header file of the `AliVEvent` class[a]. This function, however, is not implement in the `AliAnalysisTaskFlowEvent` itself, but rather in the base class of which most of the analysis task classes within `AliROOT` are derived: the `AliAnalysisTaskSE` class (which is designed to handle a single event, hence the suffix 'SE'). For each event that is written from a file, but function `AliAnalysisTaskSE::Exec()` is called, which - among other things - checks if an event passes the requested trigger selection, and if so, calls the `UserExec()` function of your analysis task. In the case of the `AliAnalysisTaskFlowEvent` this is the `AliAnalysisTaskFlowEvent::UserExec()`, which creates `AliFlowSimpleTracks` and fills the flow event.

A general remark about trigger selection in flow analyses is that the non-uniform acceptance correction methods that are implemented in the flow package assume a flat **Q** vector distribution. Specific triggers (e.g. EMCal triggers) result in a **Q** vector bias which should *not* be corrected as they invalidate that assumption. A safe approach is therefore using a minimum bias trigger for your analysis (such as `AliVEvent::kMB`), other triggers selections will not a-priori lead to problems, but use them with caution!

Event cuts

In addition to trigger selection, generally one wants to perform additional event (quality) selection. The flow package contains an event cuts class which can be used to perform event selection, the `AliFlowEventCuts` object[b].

To use the event cuts object in combination with the `AliAnalysisTaskFlowEvent` task, simply create the event cuts object, configure it and pass it to the `AliAnalysisTaskFlowEvent`:

```
AliFlowEventCuts* cutsEvent = new AliFlowEventCuts("EventCuts");
// configure some event cuts, e.g. centrality
cutsEvent->SetCentralityPercentileRange(20., 30.);
// pass it to the flow event task via the setter
AliAnalysisTaskFlowEvent::SetCutsEvent(cutsEvent);
```

The available cut parameters in the flow event cuts object are

```
Bool_t fCutNumberOfTracks;//cut on # of tracks
Int_t fNumberOfTracksMax;  //limits
Int_t fNumberOfTracksMin;  //limits
Bool_t fCutRefMult; //cut on refmult
refMultMethod fRefMultMethod; //how do we calculate refmult?
Bool_t fUseAliESDtrackCutsRefMult; //use AliESDtrackCuts for refmult calculation
AliESDtrackCuts::MultEstTrackType fRefMultMethodAliESDtrackCuts;
Int_t fRefMultMax; //max refmult
Int_t fRefMultMin; //min refmult
AliFlowTrackCuts* fRefMultCuts; //cuts
AliFlowTrackCuts* fMeanPtCuts; //mean pt cuts
AliFlowTrackCuts* fStandardTPCcuts; //Standard TPC cuts
AliFlowTrackCuts* fStandardGlobalCuts; //StandardGlobalCuts
Bool_t fCutPrimaryVertexX; //cut on x of prim vtx
Double_t fPrimaryVertexXmax; //max x prim vtx
Double_t fPrimaryVertexXmin; //min x prim vtx
Bool_t fCutPrimaryVertexY; //cut on y of prim vtx
Double_t fPrimaryVertexYmax; //max y prim vtx
Double_t fPrimaryVertexYmin; //min y prim vtx
Bool_t fCutPrimaryVertexZ; //cut on z of prim vtx
Double_t fPrimaryVertexZmax; //max z prim vtx
Double_t fPrimaryVertexZmin; //min z prim vtx
Bool_t fCutNContributors; //cut on number of contributors
Int_t fNContributorsMax; //maximal number of contrib
Int_t fNContributorsMin; //minimal number of contrib
Bool_t fCutMeanPt; //cut on mean pt
Double_t fMeanPtMax; //max mean pt
Double_t fMeanPtMin; //min mean pt
Bool_t fCutSPDvertexerAnomaly; //cut on the spd vertexer anomaly
Bool_t fCutSPDTRKVtxZ; //require compatibility between SPDvertexz TRKvertexz
Bool_t fCutTPCmultiplicityOutliers; //cut TPC multiplicity outliers
Bool_t fCutTPCmultiplicityOutliersAOD; // cut TPC outliers in 10h or 11h aod
Bool_t fUseCentralityUnchecked; //use the unchecked method
refMultMethod fCentralityPercentileMethod; //where to get the percentile from
Bool_t fCutZDCtiming;   //cut on ZDC timing
AliTriggerAnalysis fTrigAna; //trigger analysis object
Bool_t fCutImpactParameter; //cut on impact parameter (MC header)
Double_t fImpactParameterMin; // min impact parameter
Double_t fImpactParameterMax; // max impact parameter
TH2F *fhistTPCvsGlobalMult; //!correlation between TPCMult and GlobalMult
Bool_t fData2011; //2011 data is used
```

all of which are accessible via dedicated setters,

[a] `$ALICE_ROOT/...`

[b] `$ALICE_ROOT/PWG/FLOW/Tasks/AliFlowEventCuts.cxx`

```
  void SetNumberOfTracksMax(Int_t value) {fNumberOfTracksMax=value;fCutNumberOfTracks=kTRUE;}
  void SetNumberOfTracksMin(Int_t value) {fNumberOfTracksMin=value;fCutNumberOfTracks=kTRUE;}
  void SetNumberOfTracksRange(Int_t min, Int_t max) {fNumberOfTracksMin=min;fNumberOfTracksMax=max;
    fCutNumberOfTracks=kTRUE;}
  void SetRefMultMax(Int_t value) {fRefMultMax=value;fCutRefMult=kTRUE;}
  void SetRefMultMin(Int_t value) {fRefMultMin=value;fCutRefMult=kTRUE;}
  void SetRefMultRange(Int_t min, Int_t max) {fRefMultMin=min;fRefMultMax=max;fCutRefMult=kTRUE;}
  void SetImpactParameterMax(Double_t value) {fImpactParameterMax=value;fCutImpactParameter=kTRUE;}
  void SetImpactParameterMin(Double_t value) {fImpactParameterMin=value;fCutImpactParameter=kTRUE;}
  void SetImpactParameterRange(Double_t min, Double_t max) {fImpactParameterMin=min;
    fImpactParameterMax=max;fCutImpactParameter=kTRUE;}
  void SetPrimaryVertexXrange(Double_t min, Double_t max)
  void SetPrimaryVertexYrange(Double_t min, Double_t max)
  void SetPrimaryVertexZrange(Double_t min, Double_t max)
  void SetNContributorsRange(Int_t min, Int_t max=INT_MAX)
  void SetMeanPtRange(Double_t min, Double_t max) {fCutMeanPt=kTRUE; fMeanPtMax=max; fMeanPtMin=min
    ;}
  void SetCutSPDvertexerAnomaly(Bool_t b=kTRUE) {fCutSPDvertexerAnomaly=b;}
  void SetCutZDCtiming(Bool_t c=kTRUE) {fCutZDCtiming=c;}
  void SetCutSPDTRKVtxZ(Bool_t b=kTRUE) {fCutSPDTRKVtxZ=b;}
  void SetCutTPCmultiplicityOutliers(Bool_t b=kTRUE) {fCutTPCmultiplicityOutliers=b;}
  void SetCutTPCmultiplicityOutliersAOD(Bool_t b=kTRUE) {fCutTPCmultiplicityOutliersAOD=b;}
  void SetRefMultMethod(refMultMethod m) {fRefMultMethod=m;}
  void SetRefMultMethod(AliESDtrackCuts::MultEstTrackType m) { fRefMultMethodAliESDtrackCuts=m;
  void SetRefMultCuts( AliFlowTrackCuts* cuts ) {fRefMultCuts=static_cast<AliFlowTrackCuts*>(cuts->
    Clone());}
  void SetMeanPtCuts( AliFlowTrackCuts* cuts ) {fMeanPtCuts=static_cast<AliFlowTrackCuts*>(cuts->
    Clone());}
  void SetQA(Bool_t b=kTRUE) {if (b) DefineHistograms();}
  void SetCentralityPercentileMethod( refMultMethod m) {fCentralityPercentileMethod=m;}
  void SetUseCentralityUnchecked(Bool_t b=kTRUE) {fUseCentralityUnchecked=b;}
  void SetUsedDataset(Bool_t b=kTRUE) {fData2011=b;}     // confusing name, better use different
    interface
  void SetLHC10h(Bool_t b=kTRUE) {fData2011=(!b);}       // TODO let cut object determine runnumber
    and period
  void SetLHC11h(Bool_t b=kTRUE) {fData2011=b;}          // use this only as 'manual override'
```

Caveats and remarks

Some caveats and remarks about using the event cuts object

Default behavior By default, the event cuts object accepts all events. All desired cuts have to be set by the user. This is also reflected in the design of the setters: most of the setters will, when called, set a `Bool_t` to true which enables a cut on a certain parameter;

Applicability of cuts to different data types Not all the cuts can be applied to all input data types. In e.g. the process of filtering `AODs` from `ESDs`, 'technical' event cuts are made and not all events are stored in the `AOD` format. Because of this, information that can be required from `ESDs` might not be available (as it is not necessary) in `AODs`. To see whether or not a cut you set is actually applied to the data type you're using, take a look at

```
Bool_t AliFlowEventCuts::PassesCuts(AliVEvent *event, ALIMCEvent *mcevent)
```

This function determines whether or not an event is accepted: it starts by converting the virtual event type that is passed as argument to either an `ESD` or `AOD` event, and goes through selection criteria accordingly.

Event cuts outside of the `AliAnalysisTaskFlowEvent` class When you perform a flow analysis without using the `AliAnalysisTaskFlowEvent` class (which is done e.g. in the analyses explained in section 5), you can still use the event cuts class by creating an instance of the object, passing it to your analysis class and 'manually' checking the return value of the function

```
Bool_t AliFlowEventCuts::PassesCuts(AliVEvent *event, ALIMCEvent *mcevent)
```

Data taking period Most event cuts will be tuned specifically to the LHC10h or LHC11h data taking periods. The event cuts class might need to be updated to accommodate specific cuts for different periods - do not hesitate write patches for this!

for e.g. each event that is passed to your `::UserExec()` function.

3.2.3 Track cuts and the track cuts object

As explained in the previous subsection, flow events are filled with tracks which fulfill certain track selection criteria. These criteria are checked using the `AliFlowTrackCuts` class. The `AliFlowTrackCuts` class can handle different types of

input from different data-types (e.g. ESD or AOD) and information from different sub-detector systems. All input is in the end converted to AliFlowSimpleTracks which are added to the flow event. To understand how the AliFlowTrackCuts object works and how it should be configured, it is good to make a few distinctions and remarks.

The term 'track' is generally used for reconstructed particle trajectories which are constructed from information coming from the tracking detectors in central barrel of the ALICE detector (more specifically from information from the ITS and TPC detectors). Tracks are the most commonly used data source, and the translation from 'track' to AliFlowTrackSimple is trivial, as it merely comprises copying kinematic information (p_t, φ, η) from the barrel track to the AliFlowTrackSimple object.

When using information that is not coming from tracking detectors, e.g. information from the VZERO system, this procedure of simply copying variables is not suitable as the VZERO system does not measure p_t, φ, η of particles, but is an array of scintillators with limited spatial resolution. Nevertheless, the AliFlowTrackCuts class converts the VZERO signal to AliFlowTrackSimples which are, to the flow event, indistinguishable from barrel tracks. As the procedure of accepting these tracks is very different from the procedure of accepting barrel tracks, they will be treated separately in the following subsections.

ESD tracks as data source

The safest and most convenient way of using ESD tracks as a data source is by using one of the pre-defined track cuts sets that are available in the AliFlowTrackCuts class. These sets of track cuts mimic the cuts that are defined in the AliESDtrackCuts class[c]. The following default track cuts sets are available:

```
1  static AliFlowTrackCuts* GetStandardTPCStandaloneTrackCuts();
2  static AliFlowTrackCuts* GetStandardTPCStandaloneTrackCuts2010();
3  static AliFlowTrackCuts* GetStandardGlobalTrackCuts2010();
4  static AliFlowTrackCuts* GetStandardITSTPCTrackCuts2009(Bool_t selPrimaries=kTRUE);
5  static AliFlowTrackCuts* GetStandardMuonTrackCuts(Bool_t isMC=kFALSE, Int_t passN=2);
6
```

All these are static methods which create a new track cuts object and configure it properly, so to use these track cuts it suffices to type e.g.

```
1 AliFlowTrackCuts* myCuts = AliFlowTrackCuts::GetStandardGlobalTrackCuts2010();
```

To get a better understanding of what the AliFlowTrackCuts class actually does, let's take a look at what how the cut object is configured in this case:

```
1  AliFlowTrackCuts* AliFlowTrackCuts::GetStandardGlobalTrackCuts2010()
2  {
3    //get standard cuts
4    AliFlowTrackCuts* cuts = new AliFlowTrackCuts("standard Global tracks");
5    cuts->SetParamType(kGlobal);
6    cuts->SetPtRange(0.2,5.);
7    cuts->SetEtaRange(-0.8,0.8);
8    cuts->SetMinNClustersTPC(70);
9    cuts->SetMinChi2PerClusterTPC(0.1);
10   cuts->SetMaxChi2PerClusterTPC(4.0);
11   cuts->SetMinNClustersITS(2);
12   cuts->SetRequireITSRefit(kTRUE);
13   cuts->SetRequireTPCRefit(kTRUE);
14   cuts->SetMaxDCAToVertexXY(0.3);
15   cuts->SetMaxDCAToVertexZ(0.3);
16   cuts->SetAcceptKinkDaughters(kFALSE);
17   cuts->SetMinimalTPCdedx(10.);
18   return cuts;
19 }
```

The configuration falls into three categories:

1. A number of track quality cuts is set;
2. Some kinematic cuts are set;
3. The parameter type is set by calling AliFlowTrackCuts::SetParamType() (in this case to AliFlowTrackCuts::kGlobal). This last step is of particular importance as it takes care disentangling the POI and RP selection and removing a v_n bias due to auto-correlations. When the flow event is filled (the relevant piece of code is printed under section 3.2.5), a check is done to see if the POI's and RP's are of the same type. If not, a track cannot be a POI and RP at the same time (as they are from different sources). However, if POI's and RP's originate from the same source, an AliFlowTrackSimple can be both a POI and RP at the same time if it satisfies both the POI and RP track selection criteria. By specifying the parameter type by calling AliFlowTrackCuts::SetParamType() the flow event is configured to properly deal with overlapping or exclusive POI and RP selections. A wrongly configured parameter type can lead to double counting of tracks and nonsensical analysis results! The following list of track parameter types is available as an enum in AliFlowTrackCuts.h

[c] $ALICE_ROOT/ANALYSIS/AliESDtrackCuts.cxx

```
enum trackParameterType { kMC,
                          kGlobal,
                          kTPCstandalone,
                          kSPDtracklet,
                          kPMD,
                          kV0,     //neutral reconstructed v0 particle
                          kVZERO, //forward VZERO detector
                          kMUON,
                          kKink,
                          kAODFilterBit,
                          kUserA,      // reserved for custom cuts
                          kUserB       // reserved for custom cuts
                        };
```

Note that `kV0` is reserved to denote a decay vertex of a neutral particle, and `kVZERO` is used to indicate the VZERO detector system. kUserA and kUserB are additional flags which can selected for 'custom' track selection sets.

AOD tracks as data source

`AOD` tracks are derived from `ESD` tracks via process called 'filtering'. If an `ESD` track meets a pre-defined set of track cuts, it is converted to an `AOD` track which is stored in an `AOD` event. The `AOD` track carries a specific flag (called `filterbit`) which corresponds to the specific set of cuts that was applied to create accept the track. A full list of track selection criteria corresponding to distinct filterbits can be found here. Note that different `AOD` productions might have different filterbit definitions!

In `AOD` analysis it generally suffices to select tracks of a certain filterbit, instead of checking quality criteria 'by-hand' as is done in `ESD` analyses (some variables which one would cut on in `ESD` tracks might not even be available in the `AOD` tracks as the `AOD` is designed to be a light-weight 'end-user' data format). To get an instance of the `AliFlowTrackCuts` object which only selects tracks based on a specific filterbit, one can call

```
static AliFlowTrackCuts* GetAODTrackCutsForFilterBit(UInt_t bit = 1);
```

which is defined as

```
AliFlowTrackCuts* AliFlowTrackCuts::GetAODTrackCutsForFilterBit(UInt_t bit)
{
  // object which in its default form only cuts on filterbit (for AOD analysis)
  AliFlowTrackCuts* cuts = new AliFlowTrackCuts(Form("AOD fitlerbit %i", (int)bit));
  cuts->SetMinimalTPCdedx(-999999999);
  cuts->SetAODfilterBit(bit);
  cuts->SetParamType(AliFlowTrackCuts::kAODFilterBit);
  return cuts;
}
```

The `SetMinimalTPCdedx(-999999999);` is kept here for backward-compatibility.

Note that also in the case of AOD analyses the parameter type is set to (if necessary) decouple POI and RP selections.

3.2.4 Additional options

As stated, input data needn't necessarily come in the form of barrel tracks - we can use other detector systems as well. When dealing with barrel tracks, quality criteria might not be the only thing you want to select your tracks on: perhaps you want to do analysis on identified particles. The following sub-sections explain how the `AliFlowTrackCuts` object can be used to achieve this.

Identified particles

The `AliFlowTrackCuts` object can do particle selection for a number of particles that are defined in the AliPID[d]. To enable particle identification as a selection criterion, call the function

```
void AliFlowTrackCuts::SetPID(
    AliPID::EParticleType pid,
    PIDsource s=kTOFpid,
    Double_t prob=0.9)
{fParticleID=pid; fPIDsource=s; fParticleProbability=prob; fCutPID=kTRUE; InitPIDcuts();
}
```

The first argument specifies the particle species that will be selected via the `EParticleType` enum. The total list of particles as defined in the `AliPID` class reads

[d] `$ALICE_ROOT/STEER/STEERBas/AliPID.h`

```
enum EParticleType {
  kElectron = 0,
  kMuon = 1,
  kPion = 2,
  kKaon = 3,
  kProton = 4,

  kDeuteron = 5,
  kTriton = 6,
  kHe3 = 7,
  kAlpha = 8,

  kPhoton = 9,
  kPi0 = 10,
  kNeutron = 11,
  kKaon0 = 12,
  kEleCon = 13,

  kUnknown = 14
};
```

Note that not all these particles may be available for selection via `AliFlowTrackCuts`!

The second argument tells the `AliFlowTrackCuts` class which particle identification method should be used. The available methods are

```
enum PIDsource {
                kTPCpid,       // default TPC pid (via GetTPCpid)
                kTOFpid,       // default TOF pid (via GetTOFpid)
                kTOFbayesian, // TOF bayesian pid (F.Noferini)
                kTOFbeta,      // asymmetric cuts of TOF beta signal
                kTPCdedx,       // asymmetric cuts of TPC dedx signal
                kTOFbetaSimple, //simple TOF only cut
                kTPCbayesian, //bayesian cutTPC
            kTPCNuclei,   // added by Natasha for Nuclei
                kTPCTOFNsigma // simple cut on combined tpc tof nsigma
                };
```

The third argument (with a default value of 0.9) gives the analyzer control over the purity of the particle sample by setting a lower bound on the probability that a particle is of a certain species (where 0 would mean no selection and 1 -theoretically - means a 100% pure sample). To see how - and if - this parameter is used in a certain identification routine, take a look at the source code.

The best way of understanding how particles are identified is by just browsing the relevant pieces of the code in the `AliFlowTrackCuts.cxx` file (look at the list of `Passes*Cuts()`, but to give a very short overview:

kTPCpid Return particle identity as stored in the `AliESDtrack`, TPC information only;

kTOFpid Return particle identify as stored in the `AliESDtrack`, TOF information only;

kTOFbayesian Combined TPC and TOF Bayesian PID method;

kTOFbeta PID based on asymmetric TOF β cut;

kTPCdedx PID cut using TPC $\frac{dE}{dx}$ measurements stored in the `AliESDtrack`,

kTOFbetaSimple PID cut based on TOF time stored in the `AliESDtrack`;

kTPCbayesian Bayesian cut based on TPC or TOF signal;

kTPCNuclei PID selection for heavy nuclei;

kTPCTOFNsigma Cut based in a simple combined cut on the n-σ signal from TPC and TOF, requires PID response object. The PID response object is created by the PID response task, and thus requires that the PID response task runs in an analysis train *before* the `AliFlowTrackCuts` class does its selection. To enable the PID response task, add the following lines to your run macro:

```
gROOT->LoadMacro("ANALYSIS/macros/AddTaskPIDResponse.C");
AddTaskPIDResponse();
```

The default value for n-σ is 3, but it can be set to a different value using

```
void AliFlowTrackCuts::SetNumberOfSigmas(Float_t val);
```

Caveats and notes

Applicability of cuts to different data types Just as not all event and track cuts that are available for all data types. For the track quality cuts this has been explained in the previous subsections, but one has to realize that in addition, not all particle identification methods are available for all types of data. At the time of writing, the `ESD` particle identification is more elaborate than the `AOD` counterpart. To see which PID methods exist for the different data types, check the `AliFlowTrackCuts::Passes*pidCut()` functions, printed below for your convenience.

```
Bool_t AliFlowTrackCuts::PassesAODpidCut(const AliAODTrack* track )
{
     if(!track->GetAODEvent()->GetTOFHeader()){
          AliAODPid *pidObj = track->GetDetPid();
          if (!pidObj) fESDpid.GetTOFResponse().SetTimeResolution(84.);
          else{
            Double_t sigmaTOFPidInAOD[10];
            pidObj->GetTOFpidResolution(sigmaTOFPidInAOD);
            if(sigmaTOFPidInAOD[0] > 84.){
              fESDpid.GetTOFResponse().SetTimeResolution(sigmaTOFPidInAOD[0]); // use the electron
     TOF PID sigma as time resolution (including the T0 used)
          }
        }
     }

 //check if passes the selected pid cut for ESDs
  Bool_t pass = kTRUE;
  switch (fPIDsource)
  {
   case kTOFbeta:
      if (!PassesTOFbetaCut(track)) pass=kFALSE;
      break;
  case kTOFbayesian:
      if (!PassesTOFbayesianCut(track)) pass=kFALSE;
      break;
  case kTPCbayesian:
      if (!PassesTPCbayesianCut(track)) pass=kFALSE;
      break;
  case kTPCTOFNsigma:
      if (!PassesTPCTOFNsigmaCut(track)) pass = kFALSE;
      break;
  default:
    return kTRUE;
    break;
 }
  return pass;

}
//-----------------------------------------------------------------------
Bool_t AliFlowTrackCuts::PassesESDpidCut(const AliESDtrack* track )
{
  //check if passes the selected pid cut for ESDs
  Bool_t pass = kTRUE;
  switch (fPIDsource)
  {
    case kTPCpid:
      if (!PassesTPCpidCut(track)) pass=kFALSE;
      break;
    case kTPCdedx:
      if (!PassesTPCdedxCut(track)) pass=kFALSE;
      break;
    case kTOFpid:
      if (!PassesTOFpidCut(track)) pass=kFALSE;
      break;
    case kTOFbeta:
      if (!PassesTOFbetaCut(track)) pass=kFALSE;
      break;
    case kTOFbetaSimple:
      if (!PassesTOFbetaSimpleCut(track)) pass=kFALSE;
      break;
    case kTPCbayesian:
      if (!PassesTPCbayesianCut(track)) pass=kFALSE;
      break;
    case kTOFbayesian:
      if (!PassesTOFbayesianCut(track)) pass=kFALSE;
      break;
    case kTPCNuclei:
      if (!PassesNucleiSelection(track)) pass=kFALSE;
      break;
    case kTPCTOFNsigma:
      if (!PassesTPCTOFNsigmaCut(track)) pass = kFALSE;
```

```
      break;
    default:
      printf("AliFlowTrackCuts::PassesCuts() this should never be called!\n");
      pass=kFALSE;
      break;
  }
  return pass;
}
```

In general, particle identification is not a trivial procedure, and one needs to find a balance between purity and efficiency. Which particle identification to choose depends heavily on the desired outcome of the analysis. In case of e.g. a high-precision measurement of π v_2, a method which has a very high purity but low efficiency can be chosen: π's are an abundant particle species and high precision requires high purity. On the other hand, if one does selection for kaons to reconstruct φ-mesons, loose cuts with high efficiency can be chosen, as the φ-meson is a rare probe and invariant mass requirements on the kaon pairs will take care of mis-identifications.

To get access to QA information on track selection *before* and *after* PID cuts, the QA mode of the `AliFlowTrackCuts` can be selected.

Track cuts outside of the `AliAnalysisTaskFlowEvent` class Just as the flow event cuts can be used outside of the `AliAnalysisTaskFlowEvent` class, one can use the `AliFlowTrackCuts` class in a similar way, by calling, for each track,

```
Bool_t AliFlowTrackCuts::IsSelected(TObject* obj, Int_t id)
```

or directly one of the `PassesCuts(*)` functions which `IsSelected()` calls.

VZERO

Now that the barrel tracks have been explained, let's continue to the treatment of VZERO information. The VZERO detector consists of two scintillator arrays at opposite sides of the interaction point (VZEROA and VZEROC) each containing 32 readout channels. To convert the VZERO information to `AliFlowTrackCuts`, two steps are taken:

1. A 'track' is built from a VZERO tile by taking the geometric mean of the tile as the track direction (from which η and φ can be constructed);
2. The VZERO analogue signal strength within a VZERO tile (which is proportional to charge deposition) is taken as a weight when evaluating the total **Q** vector.

As there is no straightforward way to convert VZERO multiplicity to p_t, the VZERO signal can in principle not be used as POI in the flow analysis, neither can a p_t range be selected when using the VZERO as RP selection. In addition to this, the 'raw' VZERO signal itself cannot be used directly for flow analysis but needs to be calibrated tile-by-tile. To understand how this calibration is performed in the flow package, we need to go into a little bit of detail on how to build a **Q** vector.

In general, a **Q** vector is defined as

$$\mathbf{Q} = \sum_{\text{tracks}} w_i \exp\left(in\varphi\right) \tag{3.2.4.1}$$

where w_i is a track weight, n is the harmonic, and φ is the azimuthal angle of a track. As explained, in the case of VZERO tiles, φ is derived from the position of the VZERO tile and w_i is the VZERO signal which is proportional to multiplicity. However, not all VZERO tiles are equally sensitive, and the sensitivity (can have) a run-number dependence, which results in a non-flat VZERO **Q** vector distribution. As this effect might be different run-by-run, it cannot be corrected by applying a non-uniform acceptance correction at the end of your analysis, as an analysis generally comprises running over multiple run-numbers and the non-uniform acceptance correction corrects only for non-uniformity which is equal for all runs. Hence, the VZERO non-uniformity needs to be corrected at the time of the construction of the **Q** vectors.

The functions in the flow package which are responsible for building the **Q** vectors (or sub-event **Q** vectors, the use of which will be described in subsection 4.3) are

```
// Q-vector  calculation
AliFlowVector AliFlowEventSimple::GetQ(
    Int_t n,         // harmonic
    TList *weightsList,     // weight list
    Bool_t usePhiWeights,   // use phi weights?
    Bool_t usePtWeights,  // use pt weights?
    Bool_t useEtaWeights  // use eta weights?
    )

// Q-vectors of sub-events
void AliFlowEventSimple::Get2Qsub(
  AliFlowVector* Qarray,  // array with q-vectors
```

```
    Int_t n,
    TList *weightsList,
    Bool_t usePhiWeights,
    Bool_t usePtWeights,
    Bool_t useEtaWeights
    )

// overloaded implementation of Q-vectors of sub-events for VZERO information
void AliFlowEvent::Get2Qsub(
  AliFlowVector* Qarray,
  Int_t n,
  TList *weightsList,
  Bool_t usePhiWeights,
  Bool_t usePtWeights,
  Bool_t useEtaWeights
  )
```

These functions are called by the flow analysis tasks and generally not by the user directly, but it is good to know where they can be found. The first two functions merely loop over all tracks in a flow event and fill the **Q** vector. The last function is designed for building a **Q** vector from VZERO information, applying a calibration step to the VZERO signal. To make life complicated, the calibration of the VZERO **Q** vector in LHC10h is not the same as the calibration of the VZERO **Q** vector LHC11h data. Let's start by taking a look at the LHC10h case.

LHC10h The calibration of LHC10h data is a two-step procedure.

- The first step is evaluating the **Q** vector using equation 3.2.4.1. However, the VZERO signal of each tile is *re-weighted* before it is used as a weight in equation 3.2.4.1. The re-weighting comprises
  1. Taking a `TProfile` with average multiplicity per cell (these profiles are stored in a `OADB` file for each run-number)
  2. Fitting a constant line per disc (or ring) $y = a$ (see next slide for example)
  3. Evaluating the track weight for each VZERO cell is now calculated in a second iteration as

$$\text{track weight} = \frac{\text{cell multiplicity} * a}{\text{average multiplicity in a cell}} \tag{3.2.4.2}$$

- After the **Q** vectors have been built, they are re-centered. Re-centering is basically a small adjustment of the components of the **Q** vector, changing its angle event-by-event so that on average a flat **Q** vector distribution is obtained. The steps that are taken for re-centering are the following:
  1. Retrieve the average mean and spread of the **Q** vector distribution from a database file;
  2. The corrected **Q** vectors can now be obtained by doing

$$Q_n \longrightarrow \frac{Q_n - \langle Q_n \rangle}{\sigma_{Q_n}} \tag{3.2.4.3}$$

  where brackets denote the one-run average, and σ_{Q_n} the standard deviation of Q_n in the sample

Note that the calibration is only available for $n = 2$ and $n = 3$. For higher harmonics, the flow package will use the equalized VZERO multiplicity

```
AliVEvent::GetVZEROEqMultiplicity(Int_t i);
```

to build the **Q** vectors, whether this is satisfactory for an analysis, or if non-uniform acceptance effects can be reverted by performing a correction on a run-by-run basis is up to the analyzer. The **Q** vector distributions of total **Q** vectors and sub-event vectors can always be checked via the `AliFlowCommonHists` classes (see section 2.2.1) via

```
  TH1F*     GetHistQ()              {return fHistQ; } ;
  TH1F*     GetHistAngleQ()         {return fHistAngleQ; }
  TH1F*     GetHistAngleQSub0()     {return fHistAngleQSub0; }
  TH1F*     GetHistAngleQSub1()     {return fHistAngleQSub1; }
```

LHC11h The calibration of the LHC11h VZERO information is not performed by the flow package, but by an external class, name the VZEROEPselection task, which will store the corrected **Q** vectors in the AliVEvent header, from which they are retrieved by the **AliFlowTrackCuts** class. To use this method, make sure that you run this VZEROEPselection task *before* your flow analysis tasks in an analysis train. To enable this task, add the following lines to your analysis macro

```
gROOT->LoadMacro("$ALICE_ROOT/ANALYSIS/macros/AddTaskVZEROEPSelection.C");
AddTaskVZEROEPSelection();
```

Note that for LHC11h data, calibration is performed only for the second harmonic ($n = 2$). For higher harmonics, the flow package uses equalized VZERO multiplicity to build **Q** vectors (as indicated for the LHC10h data).

After describing how and why calibration is performed, it is now time to indicate how to set up this calibration routine. Just as selecting barrel tracks, this can be done by creating an `AliFlowTrackCuts` object via a `static` access method,

```
AliFlowTrackCuts* cutsVZERO = GetStandardVZEROOnlyTrackCuts();
```

At run-time, the flow package will detector whether LHC10h or LHC11h data is used by reading the analyzed events' run-number. This can be convenient when having these cuts defined in a script which is designed to run on multiple types of input data. However, one can also call the LHC10h or LHC11h specific cuts directly via dedicated functions, which are reprinted here as the comments are important

```
AliFlowTrackCuts* AliFlowTrackCuts::GetStandardVZEROOnlyTrackCuts2010()
{
  //get standard VZERO cuts
  //DISCLAIMER: LHC10h VZERO calibration consists (by default) of two steps
  //1) re-weigting of signal
  //2) re-centering of q-vectors
  //step 2 is available only for n==2 and n==3, for the higher harmonics the user
  //is repsonsible for making sure the q-sub distributions are (sufficiently) flat
  //or a sensible NUA procedure is applied !
  AliFlowTrackCuts* cuts = new AliFlowTrackCuts("standard vzero flow cuts");
  cuts->SetParamType(AliFlowTrackCuts::kVZERO);
  cuts->SetEtaRange( -10, +10 );
  cuts->SetEtaGap(-1., 1.);
  cuts->SetPhiMin( 0 );
  cuts->SetPhiMax( TMath::TwoPi() );
  // options for the reweighting
  cuts->SetVZEROgainEqualizationPerRing(kFALSE);
  cuts->SetApplyRecentering(kTRUE);
  // to exclude a ring , do e.g.
  // cuts->SetUseVZERORing(7, kFALSE);
  // excluding a ring will break the re-centering as re-centering relies on a
  // database file which tuned to receiving info from all rings
  return cuts;
}
//-----------------------------------------------------------------------
AliFlowTrackCuts* AliFlowTrackCuts::GetStandardVZEROOnlyTrackCuts2011()
{
  //get standard VZERO cuts for 2011 data
  //in this case, the vzero segments will be weighted by
  //VZEROEqMultiplicity,
  //if recentering is enableded, the sub-q vectors
  //will be taken from the event header, so make sure to run
  //the VZERO event plane selection task before this task !
  //DISCLAIMER: recentering is only available for n==2
  //for the higher harmonics the user
  //is repsonsible for making sure the q-sub distributions are (sufficiently) flat
  //or a sensible NUA procedure is applied !
  //recentering replaces the already evaluated q-vectors, so
  //when chosen, additional settings (e.g. excluding rings)
  //have no effect. recentering is true by default
  //
  //NOTE user is responsible for running the vzero event plane
  //selection task in advance, e.g. add to your launcher macro
  //
  //  gROOT->LoadMacro("$ALICE_ROOT/ANALYSIS/macros/AddTaskVZEROEPSelection.C");
  //  AddTaskVZEROEPSelection();
  //
  AliFlowTrackCuts* cuts = new AliFlowTrackCuts("standard vzero flow cuts 2011");
  cuts->SetParamType(kVZERO);
  cuts->SetEtaRange( -10, +10 );
  cuts->SetEtaGap(-1., 1.);
  cuts->SetPhiMin( 0 );
  cuts->SetPhiMax( TMath::TwoPi() );
  cuts->SetApplyRecentering(kTRUE);
  cuts->SetVZEROgainEqualizationPerRing(kFALSE);
 return cuts;
}
```

Caveats and remarks

Using the VZERO as reference detector in a flow analysis certainly has its benefits (such as suppressing the non-flow contribution to the v_n signal) but a few remarks have to be made

Applicability to flow analysis methods As the calibration affects the information that is returned by the function

```
void AliFlowEvent::Get2Qsub()
```

only flow analysis methods which call this function (and thus use sub-events) can use the calibrated VZERO signal. Most notably, this is the scalar product method. In combination with this, one should keep in mind that the two VZERO detectors have different η coverage. For the recent `ALICE` paper on the flow of identified particles, the scalar product method with VZERO sub-events was used, where the two VZERO detectors comprised the two sub-events. For more information on this, take a look at the description of the scalar product method in subsection 4.3.

VZERO as RP source The VZERO signal should only be used as source for reference flow. Although technically there is no objection to using the VZERO signal as POI's (you will probably get output) there is no guarantee that this makes sense from a 'physics' viewpoint;

Tuning of the calibration The calibration in the LHC11h data is taken from an external class and therefore, as far as the flow package is considered, as-is (although the calibration can be disabled). The LHC10h calibration however is done within the package, and can be tuned quite a bit.

Tuning the calibration is done by functions of the `AliFlowTrackCuts` class. Some of these functions apply to both LHC10h and LHC11h data but can have slightly different effects:

```
// to either enable or disable the recentering
// (for 11h this will mean that no calibration is performed,
// for 10h it will result in only doing a re-weighting)
void SetApplyRecentering(Bool_t r)
// to enable a per-ring instead of per-disc gain equalization (=re-weighting)
// (for 11h this has no effect)
void SetVZEROgainEqualizationPerRing(Bool_t s)
// exclude vzero rings: 0 through 7 can be excluded by calling this setter multiple times
// 0 corresponds to segment ID 0 through 7, etc
// disabled vzero rings get weight 0
// with this function you can omit information from entire vzero rings
// might be useful for runs where there is a bad signal in one of the tiles
// (sometimes referred to as 'clipping')
void SetUseVZERORing(Int_t i, Bool_t u)
```

Be warned however: the databases which are read during the calibration however are tuned to the combination of re-weighting of all rings with re-centering. Changing this combination might lead to biases in the **Q** vector distribution, so: playing with the calibration settings might be interesting for e.g. evaluating systematic uncertainties, but keep an eye on the control histograms!

Track weights

When it is a-priori know that a track sample needs to be weighted in φ, η or p_t (e.g. to correct for a non-uniform acceptance bias in azimuth by using weight which are inversely proportional to the azimuthal track distribution) histograms with weight distributions can be supplied to the flow package. The weights are supplied to flow analysis tasks, which then apply these weights by passing them to the `Q` vector calculation functions which are printed in the previous subsection.

The weights have to be supplied as `TH1F` objects (or objects which can be dynamically cast to a `TH1F` encapsulated in `TList`. The histograms have to have specific names: `"phi_weights"` for φ weights, `"pt_weights"` for p_t weights and `"eta_weights"` for η weights. The binning of the histograms is not important, as long as bins are of equal width. The weights are disabled by default and have to be passed to specific flow analysis tasks (as not all tasks support weights) via

```
// set weight list
AliFlowAnalysisWith*::SetWeightsList(TList* const)
// toggle phi weights on / off
AliFlowAnalysisWith*::SetUsePhiWeights(Bool_t const)
// toggle eta weighs on / off
AliFlowAnalysisWith*::SetUseEtaWeights(Bool_t const)
// toggle pt weights on / off
AliFlowAnalysisWith*::SetUsePtWeights(Bool_t const)
```

and are applied to total `Q` vectors and sub-event `Q` vectors.

The tasks which support weights are

- AliFlowAnalysisWithNestedLoops
- AliFlowAnalysisWithScalarProduct
- AliFlowAnalysisWithQCumulants
- AliFlowAnalysisTemplate
- AliFlowAnalysisWithFittingQDistribution

- AliFlowAnalysisWithCumulants
- AliFlowAnalysisWithMixedHarmonics

For details on how the weighting is implemented (and defined) the user is referred to the specific `Q` vector evaluation functions given in the previous subsection.

AliFlowCommonConstants - The Common Constants class

All flow analysis use a common output container to store their histograms. To set the configuration for the histograms in these containers - e.g. the p_t ranges of histograms, the number of bins, etc, etc - all flow analysis methods initialize their output containers using variables from a static (global) instance of the `AliFlowCommonConstants` class. This object, which can be obtained via the a static function

```
static AliFlowCommonConstants* GetMaster();
```

can be tuned to the user's liking by requesting a pointer to it via the static access method, and using the available setter functions, e.g. the following

```
AliFlowCommonConstants* cc = AliFlowCommonConstants::GetMaster();
cc->SetNbinsPt(100);
cc->SetPtMin(0);
cc->SetPtMax(10);
```

will result in an analysis which is performed in 100 p_t bins of 0.1 GeV/c width. The full set of histogram sizes and limits that can be set is

```
//histogram sizes
 Int_t  fNbinsMult; // histogram size
 Int_t  fNbinsPt;   // histogram size
 Int_t  fNbinsPhi;  // histogram size
 Int_t  fNbinsEta;  // histogram size
 Int_t  fNbinsQ;    // histogram size
 Int_t  fNbinsMass; // histogram size

 // Histograms limits
 Double_t  fMultMin;  // histogram limit
 Double_t  fMultMax;  // histogram limit
 Double_t  fPtMin;    // histogram limit
 Double_t  fPtMax;    // histogram limit
 Double_t  fPhiMin;   // histogram limit
 Double_t  fPhiMax;   // histogram limit
 Double_t  fEtaMin;   // histogram limit
 Double_t  fEtaMax;   // histogram limit
 Double_t  fQMin;     // histogram limit
 Double_t  fQMax;     // histogram limit
 Double_t  fMassMin;  // histogram limit
 Double_t  fMassMax;  // histogram limit
 Double_t  fHistWeightvsPhiMin; // histogram limit
 Double_t  fHistWeightvsPhiMax; // histogram limit
```

via the setters

```
   void SetNbinsMult( Int_t i ) { fNbinsMult = i; }
 void SetNbinsPt( Int_t i )   { fNbinsPt = i; }
 void SetNbinsPhi( Int_t i )  { fNbinsPhi = i; }
 void SetNbinsEta( Int_t i )  { fNbinsEta = i; }
 void SetNbinsQ( Int_t i )    { fNbinsQ = i; }
 void SetNbinsMass( Int_t i ) { fNbinsMass = i; }
 void SetMultMin( Double_t i ) { fMultMin = i; }
 void SetMultMax( Double_t i ) { fMultMax = i; }
 void SetPtMin( Double_t i )   { fPtMin = i; }
 void SetPtMax( Double_t i )   { fPtMax = i; }
 void SetPhiMin( Double_t i )  { fPhiMin = i; }
 void SetPhiMax( Double_t i )  { fPhiMax = i; }
 void SetEtaMin( Double_t i )  { fEtaMin = i; }
 void SetEtaMax( Double_t i )  { fEtaMax = i; }
 void SetQMin( Double_t i )    { fQMin = i; }
 void SetQMax( Double_t i )    { fQMax = i; }
 void SetMassMin( Double_t i )    { fMassMin = i; }
 void SetMassMax( Double_t i )    { fMassMax = i; }
 void SetHistWeightvsPhiMax( Double_t d ) {fHistWeightvsPhiMax=d;}
 void SetHistWeightvsPhiMin( Double_t d ) {fHistWeightvsPhiMin=d;}
```

Note that the common constants object is `static`, meaning that, within a process (e.g. an analysis train) just *one* instance of the object is created. The histogram limits and sizes that are set via the common constants object therefore affect *all* histograms within an analysis chain.

AliFlowCommonHist and AliFlowCommonHistResults - details

Both the `AliFlowCommonHist` and `AliFlowCommonHistResults` classes do not only contain (pointers to) histograms and profiles, but also have a collection of 'getters'[e] which you can use to retrieve histograms of profiles using the `ROOT` command line in stead of the `TBrowser`, which may come in handy when one needs to read the output of the flow analysis tasks in a macro.

Using the output file that was generated in the example given in the previous sections of this chapter, reading the objects of the common histogram classes is done in the following way. First, start an `(Ali)ROOT` session, and load the prerequisite libraries,

```
gSystem->Load("libPWGflowBase");
```

Then, open the analysis file and grab the common histogram objects

```
// open the file
TFile f("AnalysisResults.root");
// get the qc analysis output directory
TDirectoryFile* dir = (TDirectoryFile*)f.Get("outputQCanalysis");
// and retrieve the output list of the analysis
TList* outputList = (TList*)dir->Get("cobjQC")
```

The `TList` that you have just obtained holds not only the common histogram objects, but can also hold additional information that has been added to the analysis output by a specific flow analysis task. To read the entire content of the `TList`, you can type

```
outputList->ls();
```

However, in this example we want to retrieve the common histogram objects. To do so, type

```
// get common histogram object from the TList
AliFlowCommonHist* commonHist = (AliFlowCommonHist*)outputList->FindObject("AliFlowCommonHistQC");
// get the results for the 2 particle cumulant from the TList
AliFlowCommonHistResults* commonHistResults2 = (AliFlowCommonHistResults*)outputList->FindObject("
    AliFlowCommonHistResults2ndOrderQC");
```

Once you have retrieved the pointers to the `AliFlowCommonHist` or `AliFlowCommonHistResults` objects, you can use the getters to retrieve a histogram. To e.g. draw the η distribution of POI's, type

```
commonHist->GetHistEtaPOI()->Draw();
```

The following getters are available in `AliFlowCommonHist`

```
Double_t GetEntriesInPtBinRP(Int_t iBin);    //gets entries from fHistPtRP
Double_t GetEntriesInPtBinPOI(Int_t iBin);   //gets entries from fHistPtPOI
Double_t GetEntriesInEtaBinRP(Int_t iBin);   //gets entries from fHistEtaRP
Double_t GetEntriesInEtaBinPOI(Int_t iBin);  //gets entries from fHistEtaPOI
Double_t GetMeanPt(Int_t iBin);              //gets the mean pt for this bin from
  fHistProMeanPtperBin
TH1F*      GetHistMultRP()           {return fHistMultRP; } ;
TH1F*      GetHistMultPOI()          {return fHistMultPOI; } ;
TH2F*      GetHistMultPOIvsRP()      {return fHistMultPOIvsRP; } ;
TH1F*      GetHistPtRP()             {return fHistPtRP; } ;
TH1F*      GetHistPtPOI()            {return fHistPtPOI; } ;
TH1F*      GetHistPtSub0()           {return fHistPtSub0; } ;
TH1F*      GetHistPtSub1()           {return fHistPtSub1; } ;
TH1F*      GetHistPhiRP()            {return fHistPhiRP; } ;
TH1F*      GetHistPhiPOI()           {return fHistPhiPOI; } ;
TH1F*      GetHistPhiSub0()          {return fHistPhiSub0; } ;
TH1F*      GetHistPhiSub1()          {return fHistPhiSub1; } ;
TH1F*      GetHistEtaRP()            {return fHistEtaRP; } ;
TH1F*      GetHistEtaPOI()           {return fHistEtaPOI;  } ;
TH1F*      GetHistEtaSub0()          {return fHistEtaSub0;  } ;
TH1F*      GetHistEtaSub1()          {return fHistEtaSub1;  } ;
TH2F*      GetHistPhiEtaRP()         {return fHistPhiEtaRP;  } ;
TH2F*      GetHistPhiEtaPOI()        {return fHistPhiEtaPOI;  } ;
TProfile* GetHistProMeanPtperBin()  {return fHistProMeanPtperBin; } ;
TH2F*      GetHistWeightvsPhi()      {return fHistWeightvsPhi; } ;
TH1F*      GetHistQ()                {return fHistQ; } ;
TH1F*      GetHistAngleQ()           {return fHistAngleQ; }
TH1F*      GetHistAngleQSub0()       {return fHistAngleQSub0; }
TH1F*      GetHistAngleQSub1()       {return fHistAngleQSub1; }
TProfile* GetHarmonic()             {return fHarmonic; } ;
TProfile* GetRefMultVsNoOfRPs()     {return fRefMultVsNoOfRPs; } ;
TH1F*      GetHistRefMult()          {return fHistRefMult; } ;
TH2F*      GetHistMassPOI()          {return fHistMassPOI; }
TList*     GetHistList()             {return fHistList;} ;
```

[e] A 'getter' in this manual will be used to describe a function of the form `Get*()` which returns a (pointer to) a member of a class and is used to interface with the class.

and in `AliFlowCommonHistResults`

```
TH1D* GetHistChi(){return fHistChi;};
TH1D* GetHistIntFlow(){return fHistIntFlow;};
TH1D* GetHistIntFlowRP(){return fHistIntFlowRP;};
TH1D* GetHistDiffFlowPtRP(){return fHistDiffFlowPtRP;};
TH1D* GetHistDiffFlowEtaRP(){return fHistDiffFlowEtaRP;};
TH1D* GetHistIntFlowPOI(){return fHistIntFlowPOI;};
TH1D* GetHistDiffFlowPtPOI(){return fHistDiffFlowPtPOI;};
TH1D* GetHistDiffFlowEtaPOI(){return fHistDiffFlowEtaPOI;};
TList* GetHistList(){return fHistList;};
```

Afterburner

To e.g. test your analysis setup, an 'afterburner' can be called which adds user-defined flow to (isotropic) events. Two afterburner techniques are implemented.

Differential v_2 The first technique injects differential v_2 into events, using the following steps: As a starting point, an isotropic distribution of tracks is used

$$\frac{dN}{d\varphi_0} = \frac{1}{2\pi}. \tag{3.2.4.4}$$

Adding a periodic azimuthal modulation, this is translated to

$$\frac{dN}{d\varphi} = \frac{1}{2\pi}\left(1 + v_2 \cos\left[2\left(\varphi - \Psi\right)\right]\right) \tag{3.2.4.5}$$

which can be re-written as

$$\frac{dN}{d\varphi} = \frac{dN}{d\varphi_0}\frac{d\varphi_0}{d\varphi} = \frac{1}{2\pi}\frac{d\varphi_0}{d\varphi} \tag{3.2.4.6}$$

so that for each track the following equation can be solved by Newton-Raphson iteration

$$\varphi = \varphi_0 - v_2 \sin\left[2\left(\varphi - \Psi\right)\right]. \tag{3.2.4.7}$$

Integrated v_n The second option is adding integrated v_n by sampling the azimuthal distribution of an event from a Fourier series

$$\frac{dN}{d\varphi} \propto 1 + \frac{1}{2}\sum_n v_n\left(n\Delta\varphi\right). \tag{3.2.4.8}$$

In the 'quick start' of this manual you have already see how you can generate flow events with a certain v_n value by generating flow events by hand. The afterburner routine can also be called from the `AliAnalysisTaskFlowEvent` via the functions

```
// setters for adding by hand flow values (afterburner)

// toggle the afterburner on / off
 void SetAfterburnerOn(Bool_t b=kTRUE) {fAfterburnerOn=b;}
// set differential v2 via a TF1
 void SetPtDifferentialV2( TF1 *gPtV2) {fDifferentialV2 = gPtV2;}
 // set integrated flow (used when the gPtV2 = NULL)
 void SetFlow( Double_t v1, Double_t v2, Double_t v3=0.0, Double_t v4=0.0, Double_t v5=0.0)
              {fV1=v1;fV2=v2;fV3=v3;fV4=v4;fV5=v5;}
```

To introduce non-flow effects to using the afterburner, tracks can be cloned. To clone, for each event, a given number n of tracks, enable the afterburner and call

```
void SetNonFlowNumberOfTrackClones(Int_t n) {fNonFlowNumberOfTrackClones=n;}
```

Effectively this will result in n tracks appearing twice in the track sample, mimicking the effects of e.g. resonance decays of track splitting on v_n.

3.2.5 Relevant pieces of code

The best way of getting familiar with the flow package is perhaps browsing the source code, but it can be difficult to find a good starting point for this. Two relevant pieces of code have been selected here which are at the heart of the flow package:

1. The AliAnalysisTaskFlowEvent::UserExec() function, which is called for each event that enters an analysis train;
2. AliFlowEvent::Fill(), which selects POI's and RP's following the track selection criteria and fills the flow event which is passed to the analysis methods. The functions are shortened and simplified and provided with additional lines of comments.

AliAnalysisTaskFlowEvent::UserExec()

This function is called for each event.

```
void AliAnalysisTaskFlowEvent::UserExec(Option_t *)
{
  // Main loop
  // Called for each event
  //delete fFlowEvent;
  AliMCEvent*  mcEvent = MCEvent();                                  // from TaskSE
  AliESDEvent* myESD = dynamic_cast<AliESDEvent*>(InputEvent()); // from TaskSE
  AliAODEvent* myAOD = dynamic_cast<AliAODEvent*>(InputEvent()); // from TaskSE

  // the rp and poi cuts will be used to fill the flow event
  // so they have to be defined here
  if (!(fCutsRP&&fCutsPOI&&fCutsEvent))
  {
    AliError("cuts not set");
    return;
  }

  //DEFAULT - automatically takes care of everything
  // the flow package will determine the datatype that you are using
  if (fAnalysisType == "AUTOMATIC")
  {
    //check event cuts
    if (InputEvent() && !fCutsEvent->IsSelected(InputEvent(),MCEvent()))
      return;

    //first attach all possible information to the cuts
    // the track cuts will make the track selection, so they
    // have to be supplied with the current event
    // the mc event is NULL unless it is retrieved by AliAnalysisTaskSE
    fCutsRP->SetEvent( InputEvent(), MCEvent() );  //attach event
    fCutsPOI->SetEvent( InputEvent(), MCEvent() );

    //then make the event
    // this function will fill the flow event with selected poi's and rp's
    // the implementation is printed below
    fFlowEvent->Fill( fCutsRP, fCutsPOI );

    // pass some event info to the flow event
    fFlowEvent->SetReferenceMultiplicity(fCutsEvent->GetReferenceMultiplicity(InputEvent(),mcEvent))
    ;
    fFlowEvent->SetCentrality(fCutsEvent->GetCentrality(InputEvent(),mcEvent));
    if (mcEvent && mcEvent->GenEventHeader()) fFlowEvent->SetMCReactionPlaneAngle(mcEvent);
  }

  // a lot of code is omitted here //

  ////////////////////////////////////////////////////////////////////////////
  ////////////////////////AFTERBURNER
  if (fAfterburnerOn)
  {
    //if reaction plane not set from elsewhere randomize it before adding flow
    if (!fFlowEvent->IsSetMCReactionPlaneAngle())
      fFlowEvent->SetMCReactionPlaneAngle(gRandom->Uniform(0.0,TMath::TwoPi()));

    if(fDifferentialV2)
      fFlowEvent->AddV2(fDifferentialV2);
    else
      fFlowEvent->AddFlow(fV1,fV2,fV3,fV4,fV5);      //add flow
    fFlowEvent->CloneTracks(fNonFlowNumberOfTrackClones); //add nonflow by cloning tracks
  }
  ////////////////////////////////////////////////////////////////////////////

  //tag subEvents
  // some flow analysis methods (such as the scalar product)
  // use sub-events. by calling this function, all tracks in the
  // flow event are tagged as belonging to either sub-event a or b
  fFlowEvent->TagSubeventsInEta(fMinA,fMaxA,fMinB,fMaxB);
```

AliFlowEvent::Fill()

This function fills the flow event with `AliFlowSimpleTracks`. One important thing to notice here, is that both POI's and RP's are stored in a common array of flow tracks, internally only referred to as POI's. What distinguishes the POI's and RP's is their *type*: RP's are stored as type 0 POI's, and POI's are stored as non-zero type POI's (where nonzero means 1, 2, 3 ...).

```
//-----------------------------------------------------------------------
void AliFlowEvent::Fill( AliFlowTrackCuts* rpCuts,
                         AliFlowTrackCuts* poiCuts )
{
  //Fills the event from a vevent: AliESDEvent,AliAODEvent,AliMCEvent
  //the input data needs to be attached to the cuts
  //we have two cases, if we're cutting the same collection of tracks
  //(same param type) then we can have tracks that are both rp and poi
  //in the other case we want to have two exclusive sets of rps and pois
  //e.g. one tracklets, the other PMD or global - USER IS RESPOSIBLE
  //FOR MAKING SURE THEY DONT OVERLAP OR ELSE THE SAME PARTICLE WILL BE
  //TAKEN TWICE

  // remove the previous event
  ClearFast();
  if (!rpCuts || !poiCuts) return;
  // check the source of rp's
  AliFlowTrackCuts::trackParameterType sourceRP = rpCuts->GetParamType();
  // and ditto for the poi's
  AliFlowTrackCuts::trackParameterType sourcePOI = poiCuts->GetParamType();

  AliFlowTrack* pTrack=NULL;

  // if the source for rp's or poi's is the VZERO detector, get the calibration
  // and set the calibration parameters
  if (sourceRP == AliFlowTrackCuts::kVZERO) {
      SetVZEROCalibrationForTrackCuts(rpCuts);
      if(!rpCuts->GetApplyRecentering()) {
          // if the user does not want to recenter, switch the flag
          fApplyRecentering = -1;
      }
      // note: this flag is used in the overloaded implementation of Get2Qsub()
      // and tells the function to use as Qsub vectors the recentered Q-vectors
      // from the VZERO oadb file or from the event header
  }
  if (sourcePOI == AliFlowTrackCuts::kVZERO) {
      // probably no-one will choose vzero tracks as poi's ...
      SetVZEROCalibrationForTrackCuts(poiCuts);
  }


  if (sourceRP==sourcePOI)
  {
    //loop over tracks
    Int_t numberOfInputObjects = rpCuts->GetNumberOfInputObjects();
    for (Int_t i=0; i<numberOfInputObjects; i++)
    {
      //get input object (particle)
      TObject* particle = rpCuts->GetInputObject(i);

      Bool_t rp = rpCuts->IsSelected(particle,i);
      Bool_t poi = poiCuts->IsSelected(particle,i);

      if (!(rp||poi)) continue;

      //make new AliFlowTrack
      if (rp)
      {
        pTrack = rpCuts->FillFlowTrack(fTrackCollection,fNumberOfTracks);
        if (!pTrack) continue;
        pTrack->Tag(0); IncrementNumberOfPOIs(0);
        if (poi) {pTrack->Tag(1); IncrementNumberOfPOIs(1);}
        if (pTrack->GetNDaughters()>0) fMothersCollection->Add(pTrack);
      }
      else if (poi)
      {
        pTrack = poiCuts->FillFlowTrack(fTrackCollection,fNumberOfTracks);
        if (!pTrack) continue;
        pTrack->Tag(1); IncrementNumberOfPOIs(1);
        if (pTrack->GetNDaughters()>0) fMothersCollection->Add(pTrack);
      }
      fNumberOfTracks++;
    }//end of while (i < numberOfTracks)
  }
  else if (sourceRP!=sourcePOI)
  {
    //here we have two different sources of particles, so we fill
    //them independently
    //POI
```

```
80      for (Int_t i=0; i<poiCuts->GetNumberOfInputObjects(); i++)
81      {
82        TObject* particle = poiCuts->GetInputObject(i);
83        Bool_t poi = poiCuts->IsSelected(particle,i);
84        if (!poi) continue;
85        pTrack = poiCuts->FillFlowTrack(fTrackCollection,fNumberOfTracks);
86        if (!pTrack) continue;
87        pTrack->Tag(1);
88        IncrementNumberOfPOIs(1);
89        fNumberOfTracks++;
90        if (pTrack->GetNDaughters()>0) fMothersCollection->Add(pTrack);
91      }
92      //RP
93      Int_t numberOfInputObjects = rpCuts->GetNumberOfInputObjects();
94      for (Int_t i=0; i<numberOfInputObjects; i++)
95        {
96        TObject* particle = rpCuts->GetInputObject(i);
97        Bool_t rp = rpCuts->IsSelected(particle,i);
98        if (!rp) continue;
99        pTrack = rpCuts->FillFlowTrack(fTrackCollection,fNumberOfTracks);
100       if (!pTrack) continue;
101       pTrack->Tag(0);
102       IncrementNumberOfPOIs(0);
103       fNumberOfTracks++;
104       if (pTrack->GetNDaughters()>0) fMothersCollection->Add(pTrack);
105     }
106   }
107 }
```

3.2.6 Some words on the ALICE analysis framework

Many of the classes which are described in the previous section deal with `ALICE` data (e.g. event and track selection). Generally, this data is analyzed in `ALICE` analysis framework. This framework is setup in the following way

1. An analysis manager `analysis manager` is created;
2. The manager is connected to a source of input data (this can be data that is stored on your local machine, but more often data comes in the form of `.xml` files which point to data on `GRID` storage elements);
3. A number of analysis tasks is initialized, configured, and added to the analysis manager (so that you construct an 'analysis train');
4. The analysis is performed, which in effect means that the manager reads an event, passes it to the analysis tasks (who analyze it sequentially), and repeats this until all events are read. In this way, an event can be analyzed by many tasks whilst reading it from file just once;
5. The analysis outputs are gathered by the manager and written to an output file.

In this case of the flow package, the most common way of using this framework is

- Creating flow events using the dedicated flow event task `AliAnalysisTaskFlowEvent`;
- Analyzing these events using the `AliROOT` interface to the generic flow analysis tasks.

AliAnalysisTaskSE

All analysis tasks that are called by the analysis manager have to be derived from a common class, the `AliAnalysisTaskSE`[f] (where the suffix 'SE' stands for 'single event'). `AliAnalysisTaskSE` has a few virtual functions which can be called in user tasks by the analysis manager at specific times. Most notably these are

UserCreateOutputObjects This function is called *before* the analysis starts;

UserExec This function is called for each event;

Terminate Called at the end of the analysis (after the last event has been processed).

So, why is this important for the flow package? As said, the analysis manager can only handle tasks that derive from `AliAnalysisTaskSE`. Therefore, all flow analysis in the flow package consist of *two* classes:

AliAnalysisTask* These can be found in the 'tasks' directory of the flow package and are derived of `AliAnalysisTaskSE`. These classes interface with `AliROOT`;

[f] This section is very brief an incomplete, but keep in mind that this is a flow package manual, and not an `AliROOT` tutorial.

AliFlowAnalysisWith* These can be found in the 'base' folder of the flow package and perform the actual flow analysis.

In chapter 2 of this manual, you have seen that, using just the `AliFlowAnalysisWith*` class, a flow analysis basically follows the path

1. `Init()`: called once to initialize the task and histograms;
2. `Make()`: called for each event, does the analysis;
3. `Finish()`: wrap up the analysis.

When doing the analysis in the analysis framework, you will not use the `AliFlowAnalysisWith*` class, but instead use the `AliAnalysisTask*` which calls the `AliFlowAnalysisWith*` class for you via the calls from `AliAnalysisTaskSE`. To be more specific:

1. `Init()` is called in `UserCreateOutputObjects()`;
2. `Make()` is called in `UserExec()`;
3. `Finish()` is called in `Terminate()`.

All of this may still seem a bit abstract at this point, but in principle you now know all you need to know about the structure of the flow package. It is recommended however that you take a look at the example in 3.2.7, to get a step-by-step explanation of how these things work in the real world.

Analysys on grid: redoFinish.C

As explained in 2 and in the previous subsection, a flow analysis is finished by a call to `Finish()`. Although the exact implementation of `Finish()` is different for each flow analysis method, the general principle method in most methods is that calculations on event-averaged values are performed to end up with a final value for an observable.

When an analysis is run in parallel on many nodes (e.g. when running on `GRID`) the output of the flow analysis tasks in `AnalysisResults.root` is typically wrong, as merging files via `ROOT's TFileMerger` will trivially sum up results in all histograms.

The `redoFinish.C`[g] macro re-evaluates all output that cannot trivially be merged and re-calls the `Finish()` method. To use `redoFinish.C`, make sure your analysis output file is called `mergedAnalysisResults.root` and simply run the macro

```
.L redoFinish.C
redoFinish();
```

`redoFinish.C` will produce a new `AnalysisResults.root` file with the corrected results by calling the `::Finish()` function on all known output structures in the `mergedAnalysisResults.root` file. Additionally `redoFinish.C` can be used to repeat the call to `::Finish()` with different settings, which might alter the outcome of the flow analysis (e.g. use a different strategy to correct for non-uniform acceptance).

The macro itself is well documented and lists several options that are available at the time of running:

```
// Macro redoFinish.C is typically used after the merging macros (mergeOutput.C or
// mergeOutputOnGrid.C) have been used to produce the merged, large statistics
// file of flow analysis. Results stored in merged file are WRONG because after
// merging the results from small statistics files are trivially summed up in all
// histograms. This is taken into account and corrected for with macro redoFinish.C.
// Another typical use of the macro redoFinish.C is to repeat the call to Finish()
// in all classes, but with different values of some settings which might modify
// the final results (Example: redo the Finish() and apply correction for detector
// effects in QC code because by default this correction is switched off).

// Name of the merged, large statistics file obtained with the merging macros:
TString mergedFileName = "mergedAnalysisResults.root";
// Final output file name holding correct final results for large statistics sample:
TString outputFileName = "AnalysisResults.root";

Bool_t bApplyCorrectionForNUA = kFALSE; // apply correction for non-uniform acceptance
Bool_t bApplyCorrectionForNUAVsM = kFALSE; // apply correction for non-uniform acceptance in each
    multiplicity bin independently
Bool_t bPropagateErrorAlsoFromNIT = kFALSE; // propagate error also from non-isotropic terms
Bool_t bMinimumBiasReferenceFlow = kTRUE; // store in CRH for reference flow the result obtained
    without rebinning in multiplicity (kTRUE)
Bool_t checkForCommonHistResults = kTRUE; // check explicitely if the TList AliFlowCommonHistResults
     is available in the output
```

Flow analysis output is recognized by keywords in output list names (e.g. a Q-cumulant output needs to have the letters 'QC' somewhere in the name to be recognized).

When your analysis output is in the form of a merged file, *always* run `redoFinish.C` to get your results!

[g] `$ALICE_ROOT/PWGCF/FLOW/macros/refoFinish.C`

3.2.7 Example: $\pi^{\pm}$ v_n

As an example of how to do a flow analysis using the flow package within the AliROOT analysis framework, this section will guide you through the process of measuring $\pi^{\pm}$ v_2, v_3 and v_4 step-by-step, using the Q-vector cumulant flow analysis method.

Generally, doing an analysis in the AliROOT is a 'two-file process', where one runs a run.C script in AliROOT (coloquially referred to as 'steering macro'), which sets up the analysis framework and takes care of the interface to the analysis GRID, and calls an AddTask*.C macro which in turn creates and configures instances of the relevant analysis tasks. In this example, the distinction will not be so clear, but mentioned in the text. In practice of course, you would copy these steps into macros and launch the macros from the AliROOT command line when doing analysis. We will not run this test on GRID, but assume that you have some AliAOD.root files available on your local system. Note that this example is a guideline, there are many ways leading to Rome, and many ways of setting up an analysis. Some of the variables that are set in the code examples below are actually also set by default. This may seem a little bit redundant, but it is done to make the reader aware of the fact that they exist.

A script which contains all the steps described below and should work 'out-of-the-box' can be found at $ALICE_ROOT/PWGCF/FLOW/Documentation/examples/manual/runFlowOnDataExample.C.

Preparing the session First, we need to prepare the framework and root session (these steps would go into your run.C macro). Launch AliROOT and load the necessary libraries

```
// load libraries
gSystem->Load("libCore.so");
gSystem->Load("libGeom.so");
gSystem->Load("libVMC.so");
gSystem->Load("libPhysics.so");
gSystem->Load("libTree.so");
gSystem->Load("libSTEERBase.so");
gSystem->Load("libESD.so");
gSystem->Load("libAOD.so");
gSystem->Load("libANALYSIS.so");
gSystem->Load("libANALYSISalice.so");
gSystem->Load("libEventMixing.so");
gSystem->Load("libCORRFW.so");
gSystem->Load("libPWGTools.so");
gSystem->Load("libPWGCFebye.so");
gSystem->Load("libPWGflowBase.so");
gSystem->Load("libPWGflowTasks.so");
```

Creating the manager and connecting input data Create an analysis manager and create a TChain which we will point to the data you have stored locally on your machine

```
// create the analysis manager
AliAnalysisManager* mgr = new AliAnalysisManager("MyManager");
// create a tchain which will point to an aod tree
TChain* chain = new TChain("aodTree");
// add a few files to the chain
chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/
  AOD086/0003/AliAOD.root");
chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/
  AOD086/0003/AliAOD.root");
chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/
  AOD086/0004/AliAOD.root");
chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/
  AOD086/0005/AliAOD.root");
chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/
  AOD086/0006/AliAOD.root");
chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/
  AOD086/0007/AliAOD.root");
chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/
  AOD086/0008/AliAOD.root");
chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/
  AOD086/0009/AliAOD.root");
chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/
  AOD086/0010/AliAOD.root");
// create an input handler
AliVEventHandler* inputH = new AliAODInputHandler();
// and connect it to the manager
mgr->SetInputEventHandler(inputH);
```

Great, at this point we have created an analysis manager, which will read events from a chain of AliAOD.root files.

The next step will be adding specific analyses to the analysis manager. This is usually done by calling an AddTask*.C macro, which creates instances of analysis tasks, connects input (events from the analysis manager) to these tasks, and then connects output from the task back to the analysis manager (which will take care of writing the analysis to a common output file). These next steps show what would be in your AddTask*.C macro.

The heart of our flow analysis will be the flow event. To fill a flow event from the input AOD events, we will use the `AliAnalysisTaskFlowEvent` class. The AOD input events have to be supplied by the analysis manager, so first things first, retrieve the manager to which you will connect your flow analysis tasks[h]:

```
// the manager is static, so get the existing manager via the static method
AliAnalysisManager *mgr = AliAnalysisManager::GetAnalysisManager();
if (!mgr) {
    printf("No analysis manager to connect to!\n");
    return NULL;
}

// just to see if all went well, check if the input event handler has been connected
if (!mgr->GetInputEventHandler()) {
    printf("This task requires an input event handler!\n");
    return NULL;
  }
```

Setting up the flow event task The manager and input data are present, so we can create the flow event task and do some basic configuration

```
// create instance of the class. because possible qa plots are added in a second ouptut slot,
// the flow analysis task must know if you want to save qa plots at the time of class
  construction
Bool_t doQA = kTRUE;
// craete instance of the class
AliAnalysisTaskFlowEvent* taskFE = new AliAnalysisTaskFlowEvent("FlowEventTask", "", doQA);
// add the task to the manager
mgr->AddTask(taskFE);
// set the trigger selection
taskFE->SelectCollisionCandidates(AliVEvent::kMB);
```

Note that in the last step you have set the trigger configuration. Always make sure that you run on a trigger that makes sense for your analysis. A general remark is that the non-uniform acceptance correction methods that are implemented in the flow package, assume a flat **Q** vector distribution. Specific triggers (e.g. EMCal triggers) result in a **Q** vector bias which should *not* be corrected as they invalidate that assumption[i].

In addition to the trigger selection, one might want to do some more event selection. The flow package has a common event selection class, which we will add to your flow event

```
// define the event cuts object
AliFlowEventCuts* cutsEvent = new AliFlowEventCuts("EventCuts");
// configure some event cuts, starting with centrality
cutsEvent->SetCentralityPercentileRange(20., 30.);
// method used for centrality determination
cutsEvent->SetCentralityPercentileMethod(AliFlowEventCuts::kV0);
// vertex-z cut
cutsEvent->SetPrimaryVertexZrange(-10.,10.);
// enable the qa plots
cutsEvent->SetQA(doQA);
// explicit multiplicity outlier cut
cutsEvent->SetCutTPCmultiplicityOutliersAOD(kTRUE);
cutsEvent->SetLHC10h(kTRUE);


// and, last but not least, pass these cuts to your flow event task
taskFE->SetCutsEvent(cutsEvent);
```

Track selection Now that the flow event task has been created and some basic configuration has been done, it's time to specify the POI and RP selection. This is done by defining sets of track selection criteria for both POI's and RP's: tracks in an event that pass the track selection criteria are used as POI or RP. The track selection is defined in `AliFlowTrackCuts` objects which are passed to the `AliAnalysisTaskFlowEvent` task which does the actual selection based on the passed criteria. So, let's create some track selection objects!

Starting with the RP's, for which we'll just use a uniform selection of charged tracks,

```
//create the track cuts object using a static function of AliFlowTrackCuts
AliFlowTrackCuts* cutsRP = AliFlowTrackCuts::GetAODTrackCutsForFilterBit(1, "RP cuts");
// specify the pt range
cutsRP->SetPtRange(0.2, 5.);
```

[h] In the example macro this is a not necessary as you already have a pointer to the manager in your macro. However, if you split the macro into a steering macro and AddTask macro, the AddTask macro needs to retrieve a pointer to the manager which is created in the steering macro.

[i] The actual event selection based on triggers is done in the `AliAnalysisTaskSE` class (to be specific, the trigger is checked in `AliAnalysisTaskSE::Exec()`) from which the `AliAnalysisTaskFlowEvent` is derived. The full set of available triggers can be found in the virtual event header `AliVEvent.h`.

```
// specify eta range
cutsRP->SetEtaRange(-0.8, 0.8);
// specify track type
cutsRP->SetParamType(AliFlowTrackCuts::kAODFilterBit);
// enable saving qa histograms
cutsRP->SetQA(kTRUE);
```

The particles in this example of which we want to measure the differential v_2 (the POI's) are the charged pions. To measure the v_2 of charged pions, one must of course identify tracks are pions: for this we will use the `AliFlowTrackCuts` class. First, we do the basic setup, creating the cut object and setting some kinematic variables:

```
//create the track cuts object using a static function of AliFlowTrackCuts
AliFlowTrackCuts* cutsPOI = AliFlowTrackCuts::GetAODTrackCutsForFilterBit(1, "pion selection");
// specify the pt range
cutsPOI->SetPtRange(0.2, 5.);
// specify eta range
cutsPOI->SetEtaRange(-0.8, 0.8);
// specify the track type
cutsRP->SetParamType(AliFlowTrackCuts::kAODFilterBit);
// enable saving qa histograms
cutsPOI->SetQA(kTRUE);
```

Once this is done, the particle identification routine is defined. In this example, the particle identification will be done using a Bayesian approach, combining the signals from the TPC and TOF detectors.

```
// which particle do we want to identify ?
AliPID::EParticleType particleType=AliPID::kPion;
// specify the pid method that we want to use
AliFlowTrackCuts::PIDsource sourcePID=AliFlowTrackCuts::kTOFbayesian;
// define the probability (between 0 and 1)
Double_t probability = .9;
// pass these variables to the track cut object
cutsPOI->SetPID(particleType, sourcePID, probability);
// the bayesian pid routine uses priors tuned to an average centrality
cutsPOI->SetPriors(35.);
```

Now that the track cuts for both POI's and RP's are defined, we can connect them to the flow event task,

```
// connect the RP's to the flow event task
taskFE->SetCutsRP(cutsRP);
// connect the POI's to the flow event task
taskFE->SetCutsPOI(cutsPOI);
```

Connecting input and output At this point, the event and track cuts have been set and connected to the flow event task. The next step will be connecting the flow event task to the analysis manager (so that it can receive input events) and subsequently connecting the flow event task to flow analysis tasks, so that the flow events can be analyzed by our favorite flow analysis methods.

```
// get the default name of the output file ("AnalysisResults.root")
TString file = GetCommonFileName();
// get the common input container from the analysis manager
AliAnalysisDataContainer *cinput = mgr->GetCommonInputContainer();
 // create a data container for the output of the flow event task
 // the output of the task is the AliFlowEventSimle class which will
 // be passed to the flow analysis tasks. note that we use a kExchangeContainer here,
 // which exchanges data between classes of the analysis chain, but is not
 // written to the output file
AliAnalysisDataContainer *coutputFE = mgr->CreateContainer(
    "FlowEventContainer",
    AliFlowEventSimple::Class(),
    AliAnalysisManager::kExchangeContainer);
// connect the input data to the flow event task
mgr->ConnectInput(taskFE,0,cinput);
// and connect the output to the flow event task
mgr->ConnectOutput(taskFE,1,coutputFE);
// create an additional container for the QA output of the flow event task
// the QA histograms will be stored in a sub-folder of the output file called 'QA'
TString taskFEQAname = file;
taskFEQAname += ":QA";
AliAnalysisDataContainer* coutputFEQA = mgr->CreateContainer(
    "FlowEventContainerQA",
     TList::Class(),
     AliAnalysisManager::kOutputContainer,
     taskFEQAname.Data()
     );
// and connect the qa output container to the flow event.
```

```
// this container will be written to the output file
mgr->ConnectOutput(taskFE,2,coutputFEQA);
```

Flow analysis tasks Now that the flow event task is connected to input data, the flow analysis tasks can be set up:

```
// declare necessary pointers
AliAnalysisDataContainer *coutputQC[3];
AliAnalysisTaskQCumulants *taskQC[3];

// the tasks will be created and added to the manager in a loop
for(Int_t i = 0; i < 3; i++) {
    // create the flow analysis tasks
    taskQC[i] = new AliAnalysisTaskQCumulants(Form("TaskQCumulants_n=%i", i+2));
    // set thei triggers
    taskQC[i]->SelectCollisionCandidates(AliVEvent::kMB);
    // and set the correct harmonic n
    taskQC[i]->SetHarmonic(i+2);

    // connect the task to the analysis manager
    mgr->AddTask(taskQC[i]);

    // create and connect the output containers
    TString outputQC = file;
    // create a sub-folder in the output file for each flow analysis task's output
    outputQC += Form(":QC_output_for_n=%i", i+2);
    /// create the output containers
    coutputQC[i] = mgr->CreateContainer(
        outputQC.Data(),
        TList::Class(),
        AliAnalysisManager::kOutputContainer,
        outputQC);
    // connect the output of the flow event task to the flow analysis task
    mgr->ConnectInput(taskQC[i], 0, coutputFE);
    // and connect the output of the flow analysis task to the output container
    // which will be written to the output file
    mgr->ConnectOutput(taskQC[i], 1, coutputQC[i]);
}
```

Launching the analysis With this, the `AddTask*.C` is concluded. The only thing that is left to do, is (from the `run.C` macro) see if all tasks and containers are properly connected and initialized and launch the analysis locally:

```
// check if we can initialize the manager
if(!mgr->InitAnalysis()) return;
// print the status of the manager to screen
mgr->PrintStatus();
// print to screen how the analysis is progressing
mgr->SetUseProgressBar(1, 25);
// start the analysis locally, reading the events from the tchain
mgr->StartAnalysis("local", chain);
```

3.3 Flow analysis in ROOT: Using TTree's and TNTuples

As stated at the beginning of this chapter, every flow analysis in the flow package starts by filling the flow event. The flow event base class, `AliFlowEventSimple`, is a class in `libPWGflowBase` which has no dependencies other than some `ROOT` libraries; the same is true for the implementation of the flow analysis methods. This means that when you do not need the `AliROOT` interface for e.g. track and event selection, the flow package can be used by just invoking the `libPWGflowBase.so` library in `ROOT`[j]. The steps that are necessary to use the flow package in a bare `ROOT` environment are similar to those explained in chapter 2, with the exception that instead of generating events on-the-fly, we need to fill the flow event with information from the source of data which we want to analyze. In the next two subsections we will take a look at how to do a flow analysis on generic data in just `ROOT`. To start, pseudo-code of how to setup an analysis on a `TTree` will filled with particles be given. This example can be used as a starting point for running the flow package on any kind of input data. After this, we will work through an example of reading and analyzing `STAR` data. The last subsection of this chapter will point you to a fully working starting point for doing flow analysis on `TTree`'s, which firstly converts data to a `TTree` and after this reads the stored `TTree` from file and performs flow analysis in it in `ROOT`.

3.3.1 A custom class derived from AliFlowEventSimple

In this example, an analysis on a `TTree` is performed by deriving a class from the flow event class AliFlowEventSimple, `MyFlowEvent`, which can read a specific input format (in this case a branch`TTree!branch` of a `TTree`) and fills the flow

[j] A makefile to compile the `libPWGflowBase.so` library from the command line will be added to `$ALICE_ROOT/PWGCF/FLOW/macros/` ...

event from this input. Of course you can design your task in a different way, but in this section we will stick to that example. Note that the following suggestions are all written in pseudo-code, so copy-pasting it will lead to nothing ...

Let's start with writing an an event loop. In this example the assumption is made that you have a `TTree` with events, called 'myTree', which contains a branch holding a `TClonesArray` of 'myParticle' objects, which contain kinematic information. The 'myParticle' class could look a bit like

```
class myParticle : public TObject
{
public:
        myParticle(Float_t eta, Float_t phi, Float_t pt, Int_t charge) : fEta(eta), fPhi(phi), fpT(
    pt), fCharge(charge) {  }
    ~myParticle() {}
    virtual Double_t P()                const { return fp; }
    virtual Double_t Pt()               const { return fpT; }
    virtual Double_t Phi()              const { return fPhi; }
    virtual Double_t Eta()              const { return fEta; }
    virtual Int_t Charge()              const { return fCharge; }
private:
    Float_t                             fEta;       // eta
    Float_t                             fPhi;       // phi
    Float_t                             fpT;        // pT
    Int_t                               fCharge;    // charge
    ClassDef(myParticle, 1); // example class
};
```

Note that the members of this class (p_t, η, φ, charge) are all the information that an `AliFlowTrackSimple` needs to hold.

In the event loop, we'll retrieve the track array from the `TTree` and pass it to your derived flow event class. As we have seen in earlier examples, tracks in a flow event are classified as POI's or RP's via track cuts objects. We'll initialize these classes as well.

```
  // first, define a set of simple cuts (the kinematic cuts)
  // which will define our poi and rp selection
  AliFlowTrackSimpleCuts *cutsRP = new AliFlowTrackSimpleCuts();
  AliFlowTrackSimpleCuts *cutsPOI = new AliFlowTrackSimpleCuts();
  cutsPOI->SetPtMin(0.2);
  cutsPOI->SetPtMax(2.0);
  // get number of entries from your ttree
  Int_t nEvents = myTree->GetEntries();
  // loop over all entries
  for(Int_t i = 0; i < nEvents; i++) {
      // get the track array from the ttree
      TClonesArray* particleArray = 0x0;
      // get the branch address by name
      myTree->SetBranchAddress("myParticles", &particleArray);
      // switch to the tree's i-th entry
      myTree->GetEntry(i);
      // now we do some magic: with a dedicated inherited class
      // we construct a flow event from your ttree
      AliFlowEventSimple* flowEvent = new MyFlowEvent(particleArray, cutsPOI, cutsRP);
      // and from here we know how to proceed: connect the flow event
      // to the flow analysis classes, and do the analysis
      qc->Make(flowEvent);
      // memory management
      delete flowEvent;
      }
  qc->Finish();
  }
```

So what is 'the magic'? This is filling your flow event from the `TTree`. As we have seen in the previous sections, filling means that need to select our tracks, tag them as POI's and RP's, and add them to the flow event. Our derived class, AliFlowEventSimple::MyFlowEvent will take care of this. A possible constructor for this class, which performs the 'magic', could look like the following piece of pseudo-code:

```
// class constructor of an example class which reads a ttree,
// selects poi's and rp's and fills a flow event.
// this class is derived from the flow event simple class
// and therefore can be passed to the flow analysis methods

// we'll feed to class with your custom particles,
// so this include will be necessary
#include myParticle.h

// this is the class constructor
MyFlowEvent::MyFlowEvent(
    // start with the input tracks
    TClonesArray* particleArray,
    // and pass the poi and rp cuts
```

```
    const AliStarTrackCuts* cutsRP,
    const AliStarTrackCuts* cutsPOI) :
  // derived from AliFlowEventSimple, initialized to hold a certain number of
  // tracks
  AliFlowEventSimple(particleArray->GetEntries())
{
  // the next step will be filling the flow event
  // with POI's and RP's according to our
  // POI and RP cuts

  for (Int_t i = 0; i < particleArray->GetEntries(); i++)
  {
    // get a particle from the particle array
    const myParticle* part = static_cast<myParticle*>particleArray->At(i);
    if (!myParticle) continue;

    // build flow track simple (for the flow event)
    AliFlowTrackSimple* flowtrack = new AliFlowTrackSimple();
    // copy the kinematic information from the star track
    flowtrack->SetPhi(part->Phi());
    flowtrack->SetEta(part->Eta());
    flowtrack->SetPt(part->Pt());
    flowtrack->SetCharge(part->Charge());
    // see if the track is a reference track
    if (cutsRP)
    {
      Bool_t pass = rpCuts->PassesCuts(flowtrack);
      flowtrack->TagRP(pass); //tag RPs
      if (pass) IncrementNumberOfPOIs(0);
    }
    // see if the track is a particle of interest
    if (poiCuts)
    {
      flowtrack->TagPOI(poiCuts->PassesCuts(flowtrack));
    }
    // add the track to the flow event
    AddTrack(flowtrack);
  }
}
```

That's it! Following (variations on) these steps, you'll be able to connect any type of input data to the flow package. Note that compiling the scripts in which you define these steps will be much faster than running your code in the interpreter mode of ROOT. The next subsection will show these steps in action in the for of a flow analysis on STAR data.

3.3.2 A realistic example: flow package analysis on STAR data

The following section will show you how to use non-ALICE data in a realistic example, using events from the STAR experiment at RHIC. STAR data is stored in a TTree. To use the flow package for flow analysis on this data, the information from the TTree needs to be converted into an AliFlowEventSimple. In the specific case of the STAR data, things are a bit more complicated than in the pseudo-code example given in the previous section. Event- and track-level cuts still have to be applied to the STAR data, therefore a reader class is written which reads data from file, applies track and event cuts and converts the STAR data to 'star flow events'. This reading is left to a dedicated class, AliStarEventReader, which reads a TTree and for each event creates an AliStarEvent. The AliStarEvent is a derived class which inherits from AliFlowEventSimple (similar to the MyFlowEvent class from the example in the previous subsection). To understand this process a bit better, we'll take a look at a few code snippets from the relevant classes and macros which are currently present in AliROOT. A macro which reads STAR data and performs a flow analysis can be found at $ALICE_ROOT/PWGCF/FLOW/macros/runStarFlowAnalysis.C.

```
  // connect the class which can read and understand your ttree to
  // the input data
  AliStarEventReader starReader(inputDataFiles) ;
  // loop as long as there are events
  while ( starReader.GetNextEvent() )                    // Get next event
  {
    // read a star event from the ttree
    AliStarEvent* starEvent = starReader.GetEvent();
    // see if the event meets event cuts (of course these are
    // specific for STAR analysis, whether or not your ttree would
    // need such a cut is up to you
    if ( !starEventCuts->PassesCuts(starEvent) ) continue;

    // this is where flow package comes into play.
    // at this moment, a star event has been read from a ttree,
    // and is stored as a 'AliStarEvent'
    // in the next step, we'll create an AliFlowEventSimple from
    // this star event using the AliFlowEventStar class, which is derived
```

```
    // from the AliFlowEventSimple class.
    // as input, the AliFlowEventStar class receives the star event,
    // and a set of poi and rp cuts
    AliFlowEventSimple* flowEvent = new AliFlowEventStar(starEvent,rpCuts,poiCuts);  // make a flow
    event from a star event (aka "the magic")
    // for the scalar product method, we need to tag subevents
    flowEvent->TagSubeventsInEta(minA, maxA, minB, maxB );

    qc->Make(flowEvent);
    delete flowEvent;
  }
```

The most important piece of the code snippet printed here is the routine where the `AliFlowEventSimple` is formed from the `AliStarEvent`. What happens in the `AliFlowEventStar` class is the following:

```
// class constructor
AliFlowEventStar::AliFlowEventStar( const AliStarEvent* starevent,
                                    const AliStarTrackCuts* rpCuts,
                                    const AliStarTrackCuts* poiCuts ):
  // derived from AliFlowEventSimple, initialized to hold a certain number of
  // tracks
  AliFlowEventSimple(starevent->GetNumberOfTracks())
{
  //construct the flow event from the star event information
  SetReferenceMultiplicity(starevent->GetRefMult());
  // track loop
  for (Int_t i=0; i<starevent->GetNumberOfTracks(); i++)
  {
    // get star track from the star event
    const AliStarTrack* startrack = starevent->GetTrack(i);
    if (!startrack) continue;
    // build flow track simple (for the flow event)
    AliFlowTrackSimple* flowtrack = new AliFlowTrackSimple();
    // copy the kinematic information from the star track
    flowtrack->SetPhi(startrack->GetPhi());
    flowtrack->SetEta(startrack->GetEta());
    flowtrack->SetPt(startrack->GetPt());
    flowtrack->SetCharge(startrack->GetCharge());
    // see if the track is a reference track
    if (rpCuts)
    {
      Bool_t pass = rpCuts->PassesCuts(startrack);
      flowtrack->TagRP(pass); //tag RPs
      if (pass) IncrementNumberOfPOIs(0);
    }
    // see if the track is a particle of interest
    if (poiCuts)
    {
      flowtrack->TagPOI(poiCuts->PassesCuts(startrack)); //tag POIs
    }
    // add the track to the flow event
    AddTrack(flowtrack);
  }
}
```

3.3.3 Getting started yourself

To get started with flow analysis on `TTree's` yourself, a set of example macros and classes is provided at `$ALICE_ROOT/PWGCF/FLOW/Documentation/examples/manual/ttree`. These classes and macros will guide you through creating a `TTree` with data from `ALICE` events in the analysis framework, and performing a flow analysis on them using only `ROOT`. The example is set up as follows:

- There are two macros (in macros folder)
  - run: runs (in `AliROOT`) and fills a TTree with kinematic info from `AliVEvent`
  - read: reads (in just `ROOT`) the TTree info and performs a flow analysis with the flow package
- There are two analysis classes
  - `AliAnalysisTaskTTreeFilter`, an analysis task for AliROOT which converts input events to a `TTree`
  - `AliFlowEventSimpleFromTTree` a task for ROOT, fills flow events with `TTree` input
- and lastly two helper classes which should serve as a starting point
  - `AliFlowTTreeEvent`, a simple event class

2005 – `AliFlowTTreeTrack`, a simple track class

2006 As these are helper classes designed to get the user started, they are not compiled by default. The run and read macro
2007 will them compile on-the-fly.

Chapter 4

Methods

The flow package aims at providing the user with most of the known flow analysis methods. Detailed theoretical overview of the methods can be found in the following papers, which are included in the folder `$ALICE_ROOT/PWGCF/FLOW/Documentation/otherdocs/`

- Scalar Product Method
  `EventPlaneMethod/FlowMethodsPV.pdf`
- Generating Function Cumulants
  `GFCumulants/Borghini_GFCumulants_PracticalGuide.pdf`
- Q-vector Cumulant method
  `QCumulants/QCpaperdraft.pdf`
- Lee-Yang Zero Method
  `LeeYangZeroes/Borghini_LYZ_PracticalGuide.pdf`
- Lee-Yang Zero Method
  `LeeYangZeroesEP/LYZ_RP.pdf`

The structure of this chapter is as follows: of each of the available methods a short description is given in the `theory` subsection (for more detailed information, see the papers listed above) followed by details which are specific to the implementation in the subsection `implementation`. Caveats, possible issues, etc, are listed in the `caveats` subsections.

4.1 AliFlowAnalysisWithMCEventPlane

4.1.1 Theory

From the `.cxx` of the task:

```
// Description: Maker to analyze Flow from the generated MC reaction plane.
//              This class is used to get the real value of the flow
//              to compare the other methods to when analysing simulated events.
```

This method can be used to check what v_n was generated in an on-the-fly flow study or using the `AliAnalysisTaskFlowEvent` with afterburner.

4.1.2 Implementation

There is no specific information on the implementation here, for details the reader is referred to the source code.

4.2 AliFlowAnalysisWithQCumulants

4.2.1 Implementation

A how-to of the QC method in the flow-package is written by the author of the analysis software and is available on the FLOW-PAG twiki page (https://twiki.cern.ch/twiki/bin/view/ALICE/FlowPackageHowto). *This section is copied from the twiki page (and may therefore overlap with other parts of this manual).*

To get the first feeling how the FLOW package and QC output are organized, perhaps you can just trivially execute one 'on-the-fly' example

Essentially, you have to do two things:

```
cp $ALICE_ROOT/PWGCF/FLOW/macros/runFlowAnalysisOnTheFly.C .
aliroot runFlowAnalysisOnTheFly.C
```

In the analysis on-the-fly particles are sampled from hardwired Fourier-like p.d.f, so input vn harmonics are completely under control. Please have a look at the steering macro runFlowAnalysisOnTheFly.C and corresponding class AliFlowEventSimpleMakerOnTheFly.cxx in the FLOW package, which are easily written (no fancy C++ features in my code!), and well documented.

If you have landed successfully, you will get an output AnalysisResults.root, where the results from each method are structured in directories.

To make a size of the file lighter (which matters a lot during merging!), you may want not to use all the methods. You can make your selection of the methods via:

```
Bool_t MCEP = kTRUE; // Monte Carlo Event Plane
Bool_t SP = kTRUE; // Scalar Product (a.k.a 'flow analysis with eta gaps')
Bool_t GFC = kTRUE; // Generating Function Cumulants
Bool_t QC = kTRUE; // Q-cumulants
Bool_t FQD = kTRUE; // Fitted q-distribution
Bool_t LYZ1SUM = kTRUE; // Lee-Yang Zero (sum generating function), first pass over the data
Bool_t LYZ1PROD = kTRUE; // Lee-Yang Zero (product generating function), first pass over the data
Bool_t LYZ2SUM = kFALSE; // Lee-Yang Zero (sum generating function), second pass over the data
Bool_t LYZ2PROD = kFALSE; // Lee-Yang Zero (product generating function), second pass over the data
Bool_t LYZEP = kFALSE; // Lee-Yang Zero Event Plane
Bool_t MH = kFALSE; // Mixed Harmonics (used for strong parity violation studies)
Bool_t NL = kFALSE; // Nested Loops (neeed for debugging, only for developers)
```

Next important remark, if you want to browse through AnalysisResults.root, make sure that in AliROOT prompt you have loaded the FLOW library:

```
root [0] gSystem->Load("libPWGflowBase");
```

In the AnalysisResults.root, the QC output is stored in "outputQCanalysis". Just browse there, browse in "cobjQC", and you will see the directory structure. "Integrated Flow" ⇒ contains all results needed for reference flow. Browse in, and explore the directory (in fact, TList) "Results". The names of the histos should be self-explanatory; "Differential Flow" ⇒ browse further into "Results", and you will find a bunch of things that you can explore. For instance, in the directory "Differential Q-cumulants (POI,p_T)" you will find histos holding differential QC{2} vs pt, QC{4} vs p_T, etc. On the other hand, the flow estimates themselves, namely differential vn{2} vs pt, vn{4} vs pt you can fetch from TList "Differential Flow (POI,p_T)" I hope that the names for all other things you might need are self-explanatory. You configure QC method in the steering macro via setters:

```
qc->SetHarmonic(2);
qc->SetCalculateDiffFlow(kTRUE);
qc->SetCalculate2DDiffFlow(kFALSE); // vs (pt,eta)
qc->SetApplyCorrectionForNUA(kFALSE);
qc->SetFillMultipleControlHistograms(kFALSE);
qc->SetMultiplicityWeight("combinations"); // default (other supported options are "unit" and "
    multiplicity")
qc->SetCalculateCumulantsVsM(kFALSE);
qc->SetCalculateAllCorrelationsVsM(kFALSE); // calculate all correlations in mixed harmonics "vs M"
qc->SetnBinsMult(10000);
qc->SetMinMult(0);
qc->SetMaxMult(10000);
qc->SetBookOnlyBasicCCH(kFALSE); // book only basic common control histograms
qc->SetCalculateDiffFlowVsEta(kTRUE); // if you set kFALSE only differential flow vs pt is
    calculated
qc->SetCalculateMixedHarmonics(kFALSE); // calculate all multi-partice mixed-harmonics correlators
```

You can make QC output lighter by setting

```
qc->SetBookOnlyBasicCCH(kTRUE);
```

(to book only basic control histograms, and disabling lot of 2D beasts), and

```
qc->SetCalculateDiffFlowVsEta(kFALSE);
```

(if not interested in differential flow vs eta ⇒ this will make the final output smaller) In the "cobjQC" you might also consider "AliFlowCommonHistQC" to be useful thing, which contains a lot of trivial but still important control histograms (eg multiplicity distribution of RPs, POIs, etc). I think this is the best and fastest way for you to get familiar with the FLOW package =¿ once you send the QC code over the real data, you get the output organized in the very same way. I will send you shortly an example set of macros which get be used for the analysis on Grid over the real data. Differential QC{2} and QC{4} implementation is generic. You can tag as RP and POI whatever you want, and it will give you results automatically decoupled from any autocorrelation effects. For this reason, it is important that if you have certain particles which is classified both as RP and POI, to be explicitly tagged also as RPs and POI once you are building the "flow event". The basic feature in the FLOW package is that from whichever input you start, we have to build the same

intermediate step called "flow event", with which than we feed all methods (SP, QC, etc) in the very same way. To see what "flow event" does, and what does it need as an input, you may want to consult task AliAnalysisTaskFlowEvent.cxx and classes needed there-in.

4.3 AliFlowAnalysisWithScalarProduct

4.3.1 Theory

```
////////////////////////////////////////////////////////////////////////////
// Description: Maker to analyze Flow from the Event Plane method.
//              Adaptation based on Scalar Product
// authors: Naomi van del Kolk
//          Ante Bilandzic
// mods:    Carlos Perez
////////////////////////////////////////////////////////////////////////////
```

The scalar product method

The scalar product method estimates v_n directly from $\mathbf{Q}$ vectors:

$$v_n = \frac{\langle u \cdot Q \rangle}{\sqrt{\langle Q_A \cdot Q_B \rangle}} \tag{4.3.1.1}$$

The denominator of equation 4.3.1.1 consists of two sub-event $\mathbf{Q}$ vectors, $\mathbf{Q}_A$ and $\mathbf{Q}_B$. Sub-events are built from RP's. These sub-event vectors are in the flow package defined as coming from different η ranges.

To setup the different η ranges, one can use the `AliAnalysisTaskFlowEvent` directly by calling

```
AliAnalysisTaskFlowEvent:: void SetSubeventEtaRange(Double_t minA, Double_t maxA, Double_t minB,
    Double_t maxB)
    {this->fMinA = minA; this->fMaxA = maxA; this->fMinB = minB; this->fMaxB = maxB; }
```

Sub-events can be re-tagged using the filter task, which will be described in section 5. Internally, the tagging is performed by the function

```
AliFlowEventSimple::TagSubEventsInEta(Double_t etaMinA, Double_t etaMaxA, Double_t etaMinB, Double_t
     etaMaxB);
```

which should be called when you fill your flow events 'by-hand' and want to tag sub-events.

The numerator of equation 4.3.1.1 is the correlator of the POI $\mathbf{Q}$ vector (u) and a sub-event $\mathbf{Q}$ vector which is generally referred to as the reference detector. In the flow package, this sub-event $\mathbf{Q}$ vector is called 'total q-vector'. The user of the task needs to specify what part of the RP selection (that is, which sub-events) are used as total $\mathbf{Q}$ vector. Passing this information to the scalar product task is done in the following way

```
AliAnalysisTaskScalarProduct:: void SetTotalQvector(const char *tqv) {*this->fTotalQvector = tqv;};
```

where the following options are available

```
  TString   *fTotalQvector;      // total Q-vector is: "QaQb" (means Qa+Qb), "Qa"  or "Qb"
```

In general, one has to be a bit careful with setting up sub-events. Make sure that the combination of reference detector and sub-events is mathematically sound! An example of how to deal with complex setups is given in the VZERO scalar product subsection (4.3.1).

VZERO scalar product

The VZEROA and VZEROC detectors have different η coverage w.r.t the TPC, so to evaluate v_2 from VZERO-SP, do

$$v_n = \sqrt{\frac{\langle u_i \cdot Q_A \rangle}{\sqrt{\langle Q_A \cdot Q_B \rangle}} \cdot \frac{\langle u_j \cdot Q_B \rangle}{\sqrt{\langle Q_A \cdot Q_B \rangle}}} \tag{4.3.1.2}$$

- Q_A and Q_B are the VZEROC and VZEROA RP's

What is up for debate is the following: how do we defined the POI's?

- Take u = full TPC = $u_j = u_i$, or do $u_j = \eta < 0$, $u_i = \eta > 0$?

In the elliptic flow analysis of identified particles, majority vote has yielded the following:

- u = full TPC = $u_j = u_i$

so that in the end the published points were obtained using

$$v_n = \sqrt{\frac{\langle u \cdot Q_A \rangle}{\sqrt{\langle Q_A \cdot Q_B \rangle}} \cdot \frac{\langle u \cdot Q_B \rangle}{\sqrt{\langle Q_A \cdot Q_B \rangle}}} \tag{4.3.1.3}$$

Note that this requires running *two* scalar product tasks in the flow package (one for each reference detector) the output v_2 of which was in turn multiplied point-by-point in p_t.

Extension to Event Plane method

By normalizing the **Q** vectors, the scalar product method is essentially reduced to the 'classic' event plane method. Normalization of the **Q** vectors can be set using

```
AliAnalysisTaskScalarProduct::SetBehaveAsEP()
```

4.4 AliFlowAnalysisWithCumulants

4.4.1 Theory

```
/***************************************************
 * Flow analysis with cumulants. In this class    *
 * cumulants are calculated by making use of the *
 * formalism of generating functions proposed by *
 * Ollitrault et al.                              *
 *                                                *
 *       Author: Ante Bilandzic                   *
 ***************************************************/
```

4.4.2 Implementation

There is no specific information on the implementation here, for details the reader is referred to the source code. Do not confuse this method with the often used Q-cumulant method!

4.5 AliFlowAnalysisWithMixedHarmonics

4.5.1 Theory

There is no specific information on the theory here, for details the reader is referred to the source code.

4.5.2 Implementation

There is no specific information on the implementation here, for details the reader is referred to the source code.

4.6 AliFlowAnalysisWithFittingQDistribution

4.6.1 Theory

```
/*********************************
 * estimating reference flow by *
 *   fitting q-distribution     *
 *                              *
 * author: Ante Bilandzic       *
 *                              *
 *  based on the macro written  *
 *      by Sergei Voloshin      *
 *********************************/
```

4.6.2 Implementation

There is no specific information on the implementation here, for details the reader is referred to the source code.

4.7 AliFlowAnalysisWithMultiparticleCorrelations

4.7.1 Theory

```
/***************************************************
 * In this class azimuthal correlators in mixed harmonics *
 * are implemented in terms of Q-vectors. This approach   *
 * doesn't require evaluation of nested loops. This class *
 * can be used to:                                        *
 *                                                        *
 *  a) Extract subdominant harmonics (like v1 and v4);    *
 *  b) Study flow of two-particle resonances;             *
 *  c) Study strong parity violation.                     *
 *                                                        *
 * Author: Ante Bilandzic                                 *
 ***************************************************/
```

4.7.2 Implementation

There is no specific information on the implementation here, for details the reader is referred to the source code.

4.8 AliFlowAnalysisWithLeeYangZeros

4.8.1 Theory

```
////////////////////////////////////////////////////////////////////
// Description: Maker to analyze Flow by the LeeYangZeros method
//              One needs to do two runs over the data;
//              First to calculate the integrated flow
//              and in the second to calculate the differential flow
// Author: Naomi van der Kolk
////////////////////////////////////////////////////////////////////
```

4.8.2 Implementation

There is no specific information on the implementation here, for details the reader is referred to the source code. This method requires two passes over the data. You can take a look at the on-the-fly analysis example macro to see how these two steps can be set up:

```
Bool_t LYZ1SUM = kTRUE; // Lee-Yang Zero (sum generating function), first pass over the data
Bool_t LYZ1PROD = kTRUE; // Lee-Yang Zero (product generating function), first pass over the data
Bool_t LYZ2SUM = kFALSE; // Lee-Yang Zero (sum generating function), second pass over the data
Bool_t LYZ2PROD = kFALSE; // Lee-Yang Zero (product generating function), second pass over the data
```

4.9 AliFlowAnalysisWithLYZEventPlane

4.9.1 Theory

```
// AliFlowAnalysisWithLYZEventPlane:
// Class to do flow analysis with the event plane
// from the LYZ method
```

4.9.2 Implementation

There is no specific information on the implementation here, for details the reader is referred to the source code.

4.10 Developing your own task

Of course this list of flow analysis methods could be extended. Adding a new flow analysis method means developing two classes: a 'base' class where the method is implemented and a 'tasks' class to interface with the analysis manager. As a starting point, 'templates' have been developed, which are just empty base and task classes in the flow package. You can find these at

base `$ALICE_ROOT/PWG/FLOW/Base/AliFlowAnalysisTemplate.cxx (h)`

tasks `$ALICE_ROOT/PWG/FLOW/Tasks/AliAnalysisTaskTemplate.cxx (h)`

Chapter 5

More exotic uses

This chapter deals with more 'exotic' uses of the flow package.

5.1 Flow analysis in the LEGO framework: re-tagging your POI and RP selections

To save resources, it is beneficial to construct analysis trains in which just one flow event is created which is passed to multiple analysis tasks. This can be inconvenient when the different analysis tasks require different POI and RP selections[a]. To overcome this, a filter task, `AliAnalysisTaskFilterFE`, has been developed, which can run between the `AliAnalysisTaskFlowEvent` and a specific flow analysis task, and can re-tag POI's and RP's. The re-tagging is performed by looping over all tracks in an event and checking whether or not these tracks pass a selection of simple cuts. The filter task can only re-tag existing tracks in the flow event, it cannot add new tracks to the flow event. To illustrate the functionality of the filtertask, we'll take the example of section 3.2.7 but perform the analysis using different $|\eta|$ windows for RP's.

The first step towards filtering is setting up the filtering criteria. These are defined using the `AliFlowTrackSimpleCuts` object:

```
// create the simple cuts object
AliFlowTrackSimpleCuts* filterRP = new AliFlowTrackSimpleCuts("filterRP");
// specify a rapidity interval
filterRP->SetEtaMin(-0.4);
filterRP->SetEtaMax(0.4);
```

All available filtering options in `AliFlowTrackSimpleCuts` are:

```
  //setters
  void SetPtMax(Double_t max)    {this->fPtMax = max; fCutPt=kTRUE; }
  void SetPtMin(Double_t min)    {this->fPtMin = min; fCutPt=kTRUE;  }
  void SetEtaMax(Double_t max)   {this->fEtaMax = max; fCutEta=kTRUE; }
  void SetEtaMin(Double_t min)   {this->fEtaMin = min; fCutEta=kTRUE; }
  void SetEtaGap(Double_t min, Double_t max)
        {fEtaGapMin = min, fEtaGapMax = max, fCutEtaGap = kTRUE; }
  void SetPhiMax(Double_t max)   {this->fPhiMax = max; fCutPhi=kTRUE; }
  void SetPhiMin(Double_t min)   {this->fPhiMin = min; fCutPhi=kTRUE; }
  void SetPID(Int_t pid)         {this->fPID = pid; fCutPID=kTRUE; }
  void SetCharge(Int_t c)        {this->fCharge = c; fCutCharge=kTRUE; }
  void SetMassMax(Double_t max) {this->fMassMax = max; fCutMass=kTRUE; }
  void SetMassMin(Double_t min) {this->fMassMin = min; fCutMass=kTRUE; }
```

All cuts are disabled by default.

The second step is constructing the filter class object itself:

```
// create the filter task object. note that the desired cuts have to be passed
// in the constructor, the 0x0 that is passed means that POI's will not be filtered
AliAnalysisTaskFilterFE* filterTask = AliAnalysisTaskFilterFE("filter task", filterRP, 0x0);
```

Sub-events can also be re-defined using the filter task. To do so, call

```
AliAnalysisTaskFilterFE::SetSubeventEtaRange(Double_t minA, Double_t maxA, Double_t minB, Double_t
    maxB)
    {this->fMinA = minA; this->fMaxA = maxA; this->fMinB = minB; this->fMaxB = maxB; }
```

If yo use the filter task for a flow analysis method which uses sub-events, make sure that you set the correct η ranges! Otherwise, the default values will be used, which may (or may not) be correct for your analysis.

The `UserExec()` of the filter task is as follows:

[a] A notable example of this is doing an invariant mass analysis, which will briefly be touched in the next section.

```
void AliAnalysisTaskFilterFE::UserExec(Option_t *)
{
  // Main loop
  fFlowEvent = dynamic_cast<AliFlowEventSimple*>(GetInputData(0)); // from TaskSE
  if (!fFlowEvent) return;
  if(fCutsRFP) fFlowEvent->TagRP(fCutsRFP);
  if(fCutsPOI) fFlowEvent->TagPOI(fCutsPOI);
  fFlowEvent->TagSubeventsInEta(fMinA,fMaxA,fMinB,fMaxB);
  PostData(1,fFlowEvent);
}
```

Now that the filter task has been configured, it needs to be added to the analysis chain. As stated, the task needs to be put *in between* the flow event task and the flow analysis method.

```
// get the analysis manager
AliAnalysisManager *mgr = AliAnalysisManager::GetAnalysisManager();
// add the fitler task to the manager (should be done before the
// analysis task is added!)
mgr->AddTask(filterTask);
// create a temporary container which the filter task will pass to the
// analysis task
AliAnalysisDataContainer *coutputFilter = mgr->CreateContainer(
    "FilterContainer",
    AliFlowEventSimple::Class(),
    AliAnalysisManager::kExchangeContainer);
// connect the output of the flow analysis task as input to the filter task
mgr->ConnectInput(filterTask, 0, coutputFE);
// and connect the filter container as output
mgr->ConnectOutput(filterTask, 1, coutputFilter);
// pass the filter task output to the analysis method
// (this is assuming you already have setup the analysis task as
// explained in the example in section 3.4.3
mgr->ConnectInput(taskQC[i], 0, coutputFilter);
```

5.1.1 Caveats

Note that the filter task will change the tags of the flow tracks in the flow event. *Every* analysis task that runs after the filter task in an analysis train will therefore be affected by the re-taggging that is performed by the filter task. Often it can be useful to run multiple filter tasks with different configurations in an analysis train.

5.2 Flow analysis of resonances

One notable case in which the filter task is useful, is the flow analysis of rapidly decaying particles via the invariant mass method. If a particle decays to daughter particles, e.g.

$$\Lambda \longrightarrow \pi + \mathrm{p} \tag{5.2.0.1}$$

one can do an invariant mass flow analysis, which basically comprises

1. Take all the $\pi + \mathrm{p}$ pairs in an event and plot their invariant mass
2. Extract the signal yield N^S and total yield N^T from this distribution
3. Measure v_2 of all $\pi + \mathrm{p}$ pairs

Under the assumption that signal and background flow are additive, their contributions can be disentangled by solving

$$v_2^T(m_{inv}) = v_2^S \frac{N^S}{N^S + N^B}(m_{inv}) + v_2^B(m_{inv}) \frac{N^B}{N^S + N^B}(m_{inv}) \tag{5.2.0.2}$$

for v_2^S. To do so, $v_2^T(m_{inv})$ must be measured. This can be done by measuring the v_2 of all possible $\pi + \mathrm{p}$ pairs in different invariant mass intervals. When a flow event is filled by-hand with $\pi + \mathrm{p}$ pairs, the filter task can then be in turn be used to split the flow event into invariant mass intervals and perform flow analysis on those separately, thereby extracting all necessary information. Examples of such analyses are e.g. the -meson flow analysis (`$ALICE_ROOT/PWG/FLOW/Tasks/AliAnalylsisTaskPhiFlow`) or the Λ and K^0 flow task (`$ALICE_ROOT/PWG/FLOW/Tasks/AliAnalysisTaskFlowStrange`).

5.3 Non-uniform acceptance correction

In practice a detector can have inefficiencies which result in a non-uniform acceptance which might bias the measured v_n signal. One way of compensating for this is using track weights (as explained in section 3.2.4. Another way of correcting for these effects is by adjusting the `Q` vectors based on the assumption that the underlying `Q` vector distribution itself is flat.

By default all necessary information to perform such a correction is stored when running a flow analysis task. The actual correction itself is performed when `Finish()` is called, depending whether or not the flag to perform the correction is set to `kTRUE`.

The effects of the acceptance correction can always be checked by running the `redoFinish.C` macro, by toggling the flag

```
1 Bool_t bApplyCorrectionForNUA = kFALSE; // apply correction for non-uniform acceptance
```

to either false or true.

5.3.1 Caveats

The non-uniform acceptance correction is based on the assumption that the physical `Q` vector distribution in your event sample is flat. This works for minimum bias events, but might not work for e.g. triggered events or for event samples where the detector efficiency varies event-by-event. Details pertaining to the implementation can be found in the `Finish()` methods of the various flow analysis tasks.

Chapter 6

Summary

After reading the documentation, you should have a general feeling of how the flow package is organized and be able to do a standard flow analysis. This however is just where the fun begins! Connect your classes, write a new method, add new routines $\cdots$ and publish your paper!

Chapter 7

Bibliography

[1] J. Y. Ollitrault, Phys. Rev. D **46** (1992) 229.

[2] P. Danielewicz, Nucl. Phys. A **661** (1999) 82.

[3] D. H. Rischke, Nucl. Phys. A **610** (1996) 88C.

[4] J. Y. Ollitrault, Nucl. Phys. A **638** (1998) 195.

[5] S. Voloshin and Y. Zhang, Z. Phys. C **70** (1996) 665.

[6] K. H. Ackermann *et al.* [STAR Collaboration], Phys. Rev. Lett. **86** (2001) 402

[7] C. Adler *et al.* [STAR Collaboration], Phys. Rev. Lett. **87** (2001) 182301

[8] T.D. Lee *et al.*, New Discoveries at RHIC: Case for the Strongly Interacting Quark-Gluon Plasma. Contributions from the RBRC Workshop held May 14-15, 2004. Nucl. Phys. A **750** (2005) 1-171

Appendix A

About this document

A.1 Specifics and webpage

Typeset using LaTeX, converted to HTML using `pandoc` via `pandoc -r latex -w html -S -s -m -N --toc --highlight-style tango --indented-code-classes numberLines --self-contained -o FlowPackageManual.html FlowPackageManual.tex`

Appendix B

Flow analysis 'on-the-fly'

The original 'on-the-fly' manual by Ante Bilandžić is reprinted here in this appendix

B.1 Introduction

Flow analysis 'on the fly' is a feature in the *ALICE flow package*[a] which can serve both as a demo for the potential users of the package and as an important debugging tool for the core flow code developers. Underlying idea is very simple: To simulate events of interest for flow analysis (in what follows we shall refer to such events as *flow events*) in the computers memory and than pass them 'on the fly' to the implemented methods for flow analysis. Benefits of this approach include:

1. No need to store data on disk (storing only the output files with the final results and not the simulated events themselves);
2. Enormous gain in statistics;
3. Speed (no need to open the files from disk to read the events);
4. Random generators initialized with the same and random seed (if the same seed is used simulations are reproducible) .

In Section B.2 we indicate how the user can immediately in a few simple steps start flow analysis 'on the fly' with the default settings both within AliRoot and Root. In Section B.3 we explain how the user can modify the default settings and create 'on the fly' different flow events by following the guidance of his own taste.

B.2 Kickstart

We divide the potential users of ALICE flow package into two groups, namely the users which are using AliRoot (default) and the users which are using only Root.

B.2.1 AliRoot users

To run flow analysis 'on the fly' with the default settings within AliRoot and to see the final results obtained from various implemented methods for flow analysis, the user should execute the following steps:

Step 1: Turn off the lights ...

Step 2: ... take a deep breath ...

Step 3: ... start to copy macros `runFlowAnalysisOnTheFly.C` and `compareFlowResults.C` from `AliRoot/PWG2/FLOW/macros` to your favorite directory slowly.

Step 4: Once you have copied those macros in your favorite directory simply go to that directory and type

```
aliroot runFlowAnalysisOnTheFly.C
```

Step 5: If you have a healthy AliRoot version the flow analysis 'on the fly' will start. Once it is finished in your directory you should have the following files:

[a] `http://alisoft.cern.ch/viewvc/trunk/PWG2/FLOW/?root=AliRoot` .

```
runFlowAnalysisOnTheFly.C
compareFlowResults.C
outputLYZ1PRODanalysis.root
outputQCanalysis.root
outputFQDanalysis.root
outputLYZ1SUManalysis.root
outputSPanalysis.root
outputGFCanalysis.root
outputMCEPanalysis.root
```

Each implemented method for flow analysis produced its own output file holding various output histograms. The final flow results are stored in the common histogram structure implemented in the class `AliFlowCommonHistResults`.

Step 6: To access and compare those final flow results automatically there is a dedicated macro available, so execute

```
> aliroot
root [0] .x compareFlowResults.C("")
```

Step 7: If you want to rerun and get larger statistics modify

```
Int_t nEvts=440
```

in the macro `runFlowAnalysisOnTheFly.C`.

Step 8: Have fun!

In the next section we outline the steps for the Root users.

B.2.2 Root users

To be written at Nikhef...

B.3 Making your own flow events

This section is common both for AliRoot and Roor users. In this section we outline the procedure the user should follow in order to simulate 'on the fly' the events with his own settings by making use of the available setters. Those setters are implemented in the class `AliFlowEventSimpleMakerOnTheFly` and user shall use them in the macro `runFlowAnalysisOnTheFly.C`.

B.3.1 p_T spectra

Transverse momentum of particles is sampled from the predefined Boltzmann distribution

$$\frac{dN}{dp_T} = M p_T \exp\left(-\frac{\sqrt{m^2+p_T^2}}{T}\right), \tag{B.3.1.1}$$

where M is the multiplicity of the event, T is "temperature" and m is the mass of the particle. By increasing the parameter T one is increasing the number of high p_T particles and this parameter is the same for all events. On the other hand, multiplicity M will in general vary from event to event. In the macro `runFlowAnalysisOnTheFly.C` one can modify distribution (B.3.1.1) by using setter for "temperature" T and various setters for multiplicity M.

Example: *If one wants to increase/decrease the number of high p_T particles, one should modify the line*

```
Double_t dTemperatureOfRP = 0.44;
```

Examples of p_T spectra for two different values of T are shown in Figures B.1 and B.2.

What is shown in Figures B.1 and B.2 is only one example of the so called *common control histograms*. They are the histograms organized in the same structure and implemented in the class `AliFlowCommonHist`. In output file of each method one can access those histograms with `TBrowser`.

When it comes to multiplicity M, one has a choice to sample it event-by-event from two different distributions before plugging its value into Eq. (B.3.1.1) which than will be used to sample transverse momenta of M particles in that event.

Example: *If one wants to sample multiplicity event-by-event from Gaussian distribution with mean 500 and spread 10, one should have the following relevant settings*

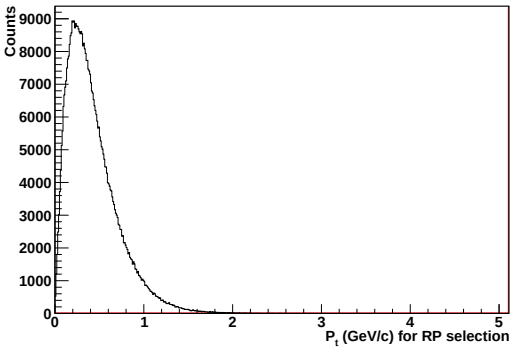

Figure B.1: $T = 0.2$ GeV/c.

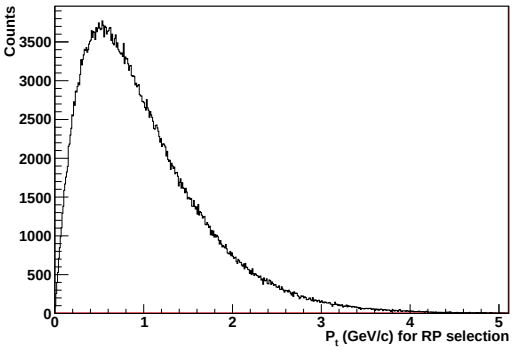

Figure B.2: $T = 0.5$ GeV/c.

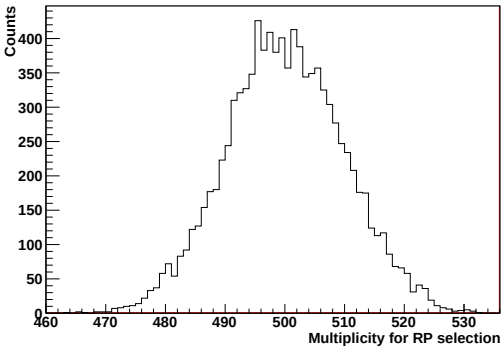

Figure B.3: Gaussian multiplicity distribution.

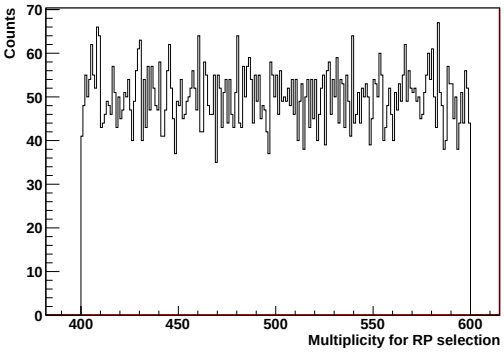

Figure B.4: Uniform multiplicity distribution.

```
Bool_t bMultDistrOfRPsIsGauss = kTRUE;
Int_t iMultiplicityOfRP = 500;
Double_t dMultiplicitySpreadOfRP = 10;
```

Example plot for multiplicity distribution with these settings is shown in Figure B.3.

Another way to sample multiplicity event-by-event is by using uniform distribution.

Example: *If one wants to sample multiplicity event-by-event from uniform distribution in the interval [400,600], one must have the following relevant settings*

```
Bool_t bMultDistrOfRPsIsGauss = kFALSE;
Int_t iMinMultOfRP = 400;
Int_t iMaxMultOfRP = 600;
```

Example plot for multiplicity distribution with these settings is shown in Figure B.4.

One can also fix multiplicity to be the same for each event.

Example: *If one wants to have the same fixed multiplicity of 500 for each event one can use the following settings:*

```
Bool_t bMultDistrOfRPsIsGauss = kTRUE;
Int_t iMultiplicityOfRP = 500;
Double_t dMultiplicitySpreadOfRP = 0;
```

These are all manipulations available at the moment with p_T spectra given in Eq. (B.3.1.1).

B.3.2 Azimuthal distribution

If the anisotropic flow exists, it will manifest itself in the anisotropic azimuthal distribution of outgoing particles measured with respect to the reaction plane:

$$E\frac{d^3N}{d^3\vec{p}} = \frac{1}{2\pi}\frac{d^2N}{p_T dp_T d\eta}\left(1 + \sum_{n=1}^{\infty} 2v_n(p_T,\eta)\cos\left(n\left(\phi - \Psi_{\mathrm{RP}}\right)\right)\right). \qquad \text{(B.3.2.1)}$$

Flow harmonics v_n quantify anisotropic flow and are in general function of transverse momentum p_T and pseudorapidity η. Orientation of reaction plane Ψ_{RP} fluctuates randomly event-by-event and cannot be measured directly. In the implementation 'on the fly' reaction plane is sampled uniformly event-by-event from the interval $[0^{\mathrm{o}}, 360^{\mathrm{o}}]$. When it comes to flow harmonics, there are two modes which we outline next.

Constant flow harmonics

In this mode all flow harmonics are treated as a constant, event-wise quantities, meaning that for a particular event azimuthal angles of all particles will be sampled from the same azimuthal distribution in which flow harmonics appear just as fixed parameters. The implemented most general azimuthal distribution for this mode reads

$$\frac{dN}{d\phi} = 1 + 2v_1\cos(\phi - \Psi_{\mathrm{RP}}) + 2v_2\cos(2(\phi - \Psi_{\mathrm{RP}})) + 2v_4\cos(4(\phi - \Psi_{\mathrm{RP}}))\,. \tag{B.3.2.2}$$

In the macro `runFlowAnalysisOnTheFly.C` one can use the dedicated setters and have handle on the flow harmonics v_1, v_2 and v_4. The most important harmonic is v_2, the so called *elliptic flow*, so we start with it first.

Example: *If one wants to sample particle azimuthal angles from azimuthal distribution parameterized only with constant elliptic flow of 5%, namely*

$$\frac{dN}{d\phi} = 1 + 2\cdot 0.05\cdot\cos(2(\phi - \Psi_{\mathrm{RP}}))\,, \tag{B.3.2.3}$$

then one should use the following settings

```
Bool_t bConstantHarmonics = kTRUE;
Bool_t bV2DistrOfRPsIsGauss = kTRUE;
Double_t dV2RP = 0.05;
Double_t dV2SpreadRP = 0.0;
Double_t dV1RP = 0.0;
Double_t dV1SpreadRP = 0.0;
Double_t dV4RP = 0.0;
Double_t dV4SpreadRP = 0.0;
```

In this mode the flow coefficients are constant for all particles within particular event, but still the flow coefficients can fluctuate event-by-event.

Example: *If one wants to sample particle azimuthal angles from azimuthal distribution parameterized only with elliptic flow which fluctuates event-by-event according to Gaussian distribution with mean 5% and spread 1%, than one should use the following settings*

```
Bool_t bConstantHarmonics = kTRUE;
Bool_t bV2DistrOfRPsIsGauss = kTRUE;
Double_t dV2RP = 0.05;
Double_t dV2SpreadRP = 0.01;
Double_t dV1RP = 0.0;
Double_t dV1SpreadRP = 0.0;
Double_t dV4RP = 0.0;
Double_t dV4SpreadRP = 0.0;
```

On can also study uniform flow fluctuations.

Example: *If one wants to sample particle azimuthal angles from azimuthal distribution parameterized only with elliptic flow which fluctuates event-by-event according to uniform distribution in interval [4%,6%], than one should use the following settings*

```
Bool_t bConstantHarmonics = kTRUE;
Bool_t bV2DistrOfRPsIsGauss = kFALSE;
Double_t dMinV2RP = 0.04;
Double_t dMinV2RP = 0.06;
Double_t dV1RP = 0.0;
Double_t dV1SpreadRP = 0.0;
Double_t dV4RP = 0.0;
Double_t dV4SpreadRP = 0.0;
```

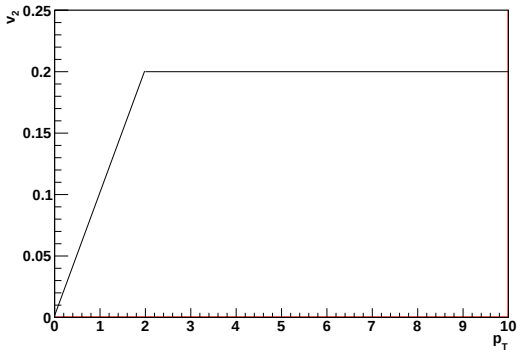

Figure B.5: p_T dependent elliptic flow.

It is of course possible to simulate simultanously nonvanishing v_1, v_2 and v_4.

Example: *If one wants to sample particle azimuthal angles from azimuthal distribution parameterized by harmonics* $v_1 = 2\%$, $v_2 = 5\%$ *and* $v_4 = 1\%$, *namely*

$$\begin{aligned} \frac{dN}{d\phi} &= 1 + 2 \cdot 0.02 \cdot \cos(\phi - \Psi_{\mathrm{RP}}) + 2 \cdot 0.05 \cdot \cos(2(\phi - \Psi_{\mathrm{RP}})) \\ &+ 2 \cdot 0.01 \cdot \cos(4(\phi - \Psi_{\mathrm{RP}})) \end{aligned} \tag{B.3.2.4}$$

then one should use the following settings

```
Bool_t bConstantHarmonics = kTRUE;
Bool_t bV2DistrOfRPsIsGauss = kTRUE;
Double_t dV2RP = 0.05;
Double_t dV2SpreadRP = 0.0;
Double_t dV1RP = 0.02;
Double_t dV1SpreadRP = 0.0;
Double_t dV4RP = 0.01;
Double_t dV4SpreadRP = 0.0;
```

In the next section we outline the procedure for simulating flow events with p_T dependent flow harmonics.

p_T dependent flow harmonics

In this mode the functional dependence of flow harmonics on transverse momentum is treated as an event-wise quantity, while within the particular event the flow harmonics will change from particle to particle depending on its transverse momentum. The implemented azimuthal distribution for this case reads

$$\frac{dN}{d\phi} = 1 + 2v_2(p_T)\cos(2(\phi - \Psi_{\mathrm{RP}}))\,, \tag{B.3.2.5}$$

and the functional dependence $v_2(p_T)$ is implemented as follows:

$$v_2(p_T) = \begin{cases} v_{\max}(p_T/p_{\mathrm{cutoff}}) & p_T < p_{\mathrm{cutoff}}\,, \\ v_{\max} & p_T \geq p_{\mathrm{cutoff}}\,. \end{cases} \tag{B.3.2.6}$$

In the macro `runFlowAnalysisOnTheFly.C` one can have the handle on the parameters $v_{\max}$ and p_{cutoff}.

Example: *If one wants to set* $v_{\max} = 0.2$ *and* $p_{\mathrm{cutoff}} = 2$ *GeV/c, than one should use the following settings:*

```
Bool_t bConstantHarmonics = kFALSE;
Double_t dV2RPMax = 0.20;
Double_t dPtCutOff = 2.0;
```

Example plot is given in Figure B.5.

(Remark: Add further explanation here.)

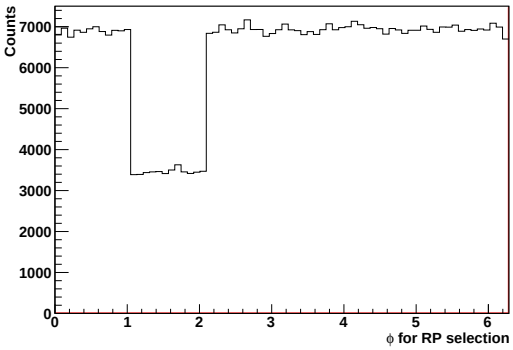

Figure B.6: Non-uniform acceptance.

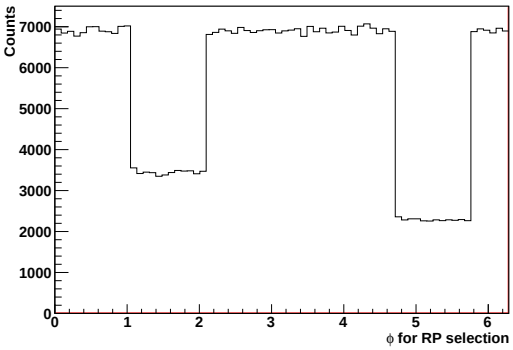

Figure B.7: Non-uniform acceptance.

B.3.3 Nonflow

One can simply simulate strong 2-particle nonflow correlations by taking each particle twice.

Example: *If one wants to simulate strong 2-particle nonflow correlations one should simply set*

```
Int_t iLoops = 2;
```

B.3.4 Detector inefficiencies

In reality we never deal with a detector with uniform azimuthal coverage, hence a need for a thorough studies of the systematic bias originating from the non-uniform acceptance.

Example: *One wants to simulate a detector whose acceptance is uniform except for the sector which spans the azimuthal interval* $[60^{\circ}, 120^{\circ}]$*. In this sector there are some issues, so only half of the particles are reconstructed. To simulate this acceptance one should use the following settings:*

```
Bool_t uniformAcceptance = kFALSE;
Double_t phimin1 = 60;
Double_t phimax1 = 120;
Double_t p1 = 1/2.;
Double_t phimin2 = 0.0;
Double_t phimax2 = 0.0;
Double_t p2 = 0.0;
```

The resulting azimuthal profile is shown in Figure (B.6).

One can also simulate two problematic sectors.

Example: *One wants to simulate a detector whose acceptance is uniform except for the two sectors which span azimuth* $[60^{\circ}, 120^{\circ}]$ *and* $[270^{\circ}, 330^{\circ}]$*, respectively. In the first sector only 1/2 of the particles are reconstructed and only 1/3 of the particles are reconstructed in the second. To simulate this acceptance one should use the following settings:*

```
Bool_t uniformAcceptance = kFALSE;
Double_t phimin1 = 60;
Double_t phimax1 = 120;
Double_t p1 = 1/2.;
Double_t phimin2 = 270.0;
Double_t phimax2 = 330.0;
Double_t p2 = 1/3.;
```

The resulting azimuthal profile is shown in Figure (B.7).

Index