

The FLOW Analysis Package



a short manual

June 23, 2014

Redmer Alexander Bertens
(`rbertens @ cern.ch`)

with excerpts from other manuals, authors of those are mentioned in text

Contents

8	1 Introduction	1
9	1.1 This manual	1
10	1.2 Disclaimer	1
11	2 A Quick Start	3
12	2.1 On the fly - getting started on a Toy MC	3
13	2.2 What is in the output file ?	5
14	2.2.1 AliFlowCommonHists - Output objects	5
15	3 The Program	7
16	3.1 Overview	7
17	3.2 Flow Event	7
18	3.2.1 Input data	7
19	3.2.2 Event selection	8
20	3.3 Track cuts and the track cuts object	10
21	3.3.1 ESD tracks as data source	11
22	3.3.2 AOD tracks as data source	12
23	3.3.3 Additional options	12
24	3.3.4 Relevant pieces of code	21
25	3.4 The flow package in the ALICE analysis framework	24
26	3.4.1 AliAnalysisTaskSE	24
27	3.4.2 Example: $\pi^\pm v_n$	26
28	4 Methods	31
29	4.1 AliFlowAnalysisWithMCEventPlane	31
30	4.1.1 Theory	31
31	4.1.2 Implementation	31
32	4.2 AliFlowAnalysisWithQCumulants	31
33	4.2.1 Implementation	31
34	4.3 AliFlowAnalysisWithScalarProduct	33
35	4.3.1 Theory	33
36	4.4 AliFlowAnalysisWithCumulants	34
37	4.4.1 Theory	34
38	4.4.2 Implementation	34
39	4.5 AliFlowAnalysisWithMixedHarmonics	34
40	4.5.1 Theory	34
41	4.5.2 Implementation	34
42	4.6 AliFlowAnalysisWithFittingQDistribution	34
43	4.6.1 Theory	34
44	4.6.2 Implementation	34
45	4.7 AliFlowAnalysisWithMultiparticleCorrelations	35
46	4.7.1 Theory	35
47	4.7.2 Implementation	35
48	4.8 AliFlowAnalysisWithLeeYangZeros	35
49	4.8.1 Theory	35
50	4.8.2 Implementation	35
51	4.9 AliFlowAnalysisWithLYZEventPlane	35
52	4.9.1 Theory	35
53	4.9.2 Implementation	35
54	4.10 Developing your own task	35

55	5 More exotic uses	37
56	5.1 Flow analysis in the LEGO framework: re-tagging your POI and RP selections	37
57	5.1.1 Caveats	38
58	5.2 Flow analysis of resonances	38
59	5.3 Non-uniform acceptance correction	39
60	5.3.1 Caveats	39
61	6 Summary	41
62	7 Bibliography	43
63	A About this document	45
64	A.1 Specifics and webpage	45
65	B Flow analysis ‘on-the-fly’	47
66	B.1 Introduction	47
67	B.2 Kickstart	47
68	B.2.1 AliRoot users	47
69	B.2.2 Root users	48
70	B.3 Making your own flow events	48
71	B.3.1 p_T spectra	48
72	B.3.2 Azimuthal distribution	49
73	B.3.3 Nonflow	52
74	B.3.4 Detector inefficiencies	52
75	Index	53

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);
- 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`.

^aThe 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:

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

In root additional libraries need to be loaded:

```
1 gSystem->Load("libGeom");
2 gSystem->Load("libVMC");
3 gSystem->Load("libXMLIO");
4 gSystem->Load("libPhysics");
5 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.

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

^aIn 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.

^bIn 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.

^cIn aliroot, this macro can be found at

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

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

4. To define the particles we are going to use as Reference Particles (RP's, particles used for the \mathbf{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:

```
1 AliFlowTrackSimpleCuts *cutsRP = new AliFlowTrackSimpleCuts();
2 AliFlowTrackSimpleCuts *cutsPOI = new AliFlowTrackSimpleCuts();
3 cutsPOI->SetPtMin(0.2);
4 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

```
1 Int_t nEvents = 1000; // generate 1000 events
2 Int_t mult = 2000; // use track multiplicity of 2000
3 Double_t v2 = .05; // 5 pct integrated flow
4 // or sample differential flow
5 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

```
1 for(Int_t i=0; i<nEvents; i++) {
2 // make an event with mult particles
3 AliFlowEventSimple* flowevent = AliFlowEventSimple(mult, AliFlowEventSimple::kGenerate);
4 // modify the tracks adding the flow value v2
5 flowevent->AddV2(diffv2);
6 // select the particles for the reference flow
7 flowevent->TagRP(cutsRP);
8 // select the particles for differential flow
9 flowevent->TagPOI(cutsPOI);
10 // do flow analysis with various methods:
11 mcep->Make(flowevent);
12 qc->Make(flowevent);
13 // delete the event from memory
14 delete flowevent;
15 }
```

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

```
1 mcep->Finish();
2 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

```
1 TFile *outputFile = new TFile("outputMCEPanalysis.root", "RECREATE");
2 mcep->WriteHistograms();
3 TFile *outputFile = new TFile("outputQCanalysis.root", "RECREATE");
4 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 subdirectories of one file, one can do the following:

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

^dThe 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 `AnalysisResults.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

```
1 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:

```
1 Bool_t      fBookOnlyBasic; // book and fill only control histos needed for all methods
2 TH1F*      fHistMultRP; // multiplicity for RP selection
3 TH1F*      fHistMultPOI; // multiplicity for POI selection
4 TH2F*      fHistMultPOIvsRP; // multiplicity for POI versus RP
5 TH1F*      fHistPtRP; // pt distribution for RP selection
6 TH1F*      fHistPtPOI; // pt distribution for POI selection
7 TH1F*      fHistPtSub0; // pt distribution for subevent 0
8 TH1F*      fHistPtSub1; // pt distribution for subevent 1
9 TH1F*      fHistPhiRP; // phi distribution for RP selection
10 TH1F*     fHistPhiPOI; // phi distribution for POI selection
11 TH1F*     fHistPhiSub0; // phi distribution for subevent 0
12 TH1F*     fHistPhiSub1; // phi distribution for subevent 1
13 TH1F*     fHistEtaRP; // eta distribution for RP selection
14 TH1F*     fHistEtaPOI; // eta distribution for POI selection
15 TH1F*     fHistEtaSub0; // eta distribution for subevent 0
16 TH1F*     fHistEtaSub1; // eta distribution for subevent 1
17 TH2F*     fHistPhiEtaRP; // eta vs phi for RP selection
18 TH2F*     fHistPhiEtaPOI; // eta vs phi for POI selection
19 TProfile* fHistProMeanPtperBin; // mean pt for each pt bin (for POI selection)
20 TH2F*     fHistWeightvsPhi; // particle weight vs particle phi
21 TH1F*     fHistQ; // Qvector distribution
22 TH1F*     fHistAngleQ; // distribution of angle of Q vector
23 TH1F*     fHistAngleQSub0; // distribution of angle of subevent 0 Q vector
24 TH1F*     fHistAngleQSub1; // distribution of angle of subevent 1 Q vector
25 TProfile* fHarmonic; // harmonic
26 TProfile* fRefMultVsNoOfRPs; // <reference multiplicity> versus # of RPs
27 TH1F*     fHistRefMult; // reference multiplicity distribution
28 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

```
1 TH1D* fHistIntFlow; // reference flow
2 TH1D* fHistChi; // resolution
3 // RP = Reference Particles:
4 TH1D* fHistIntFlowRP; // integrated flow of RPs
5 TH1D* fHistDiffFlowPtRP; // differential flow (Pt) of RPs
6 TH1D* fHistDiffFlowEtaRP; // differential flow (Eta) of RPs
7 // POI = Particles Of Interest:
8 TH1D* fHistIntFlowPOI; // integrated flow of POIs
9 TH1D* fHistDiffFlowPtPOI; // differential flow (Pt) of POIs
10 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).

^eMake sure that `libPWGflowBase.so` is loaded in your (Ali)ROOT session, otherwise these objects will be unknown.

^fThe headers of both output objects can be found in `$ALICE_ROOT/PWG/FLOW/Base/`.

^gThe 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!

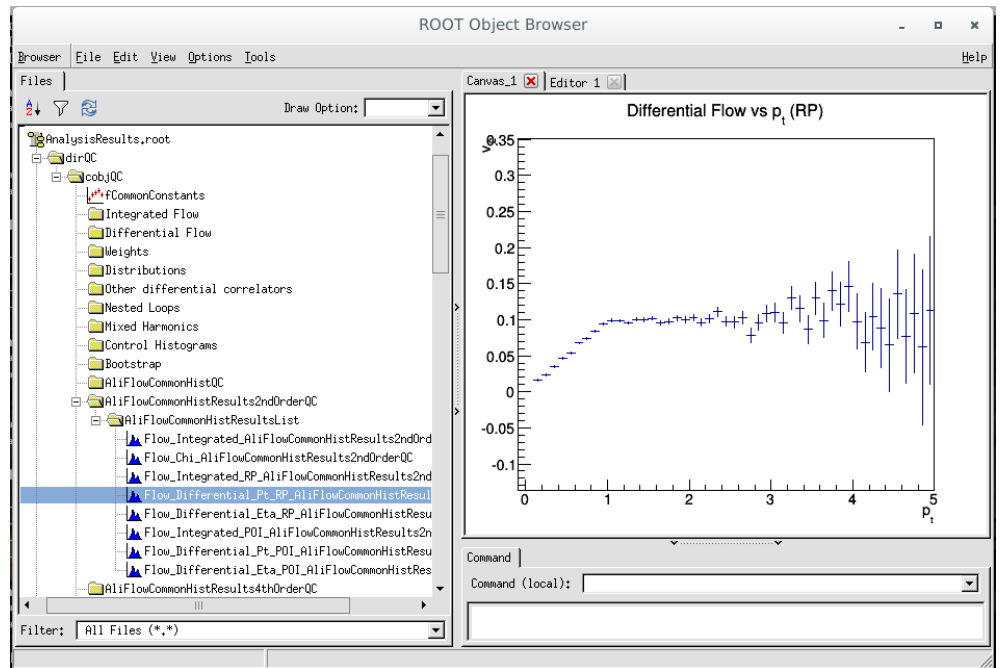


Figure 2.1: Example of output file opened in a TBrowser, results of differential v_2 analysis with second order Q-cumulant analysis are shown.

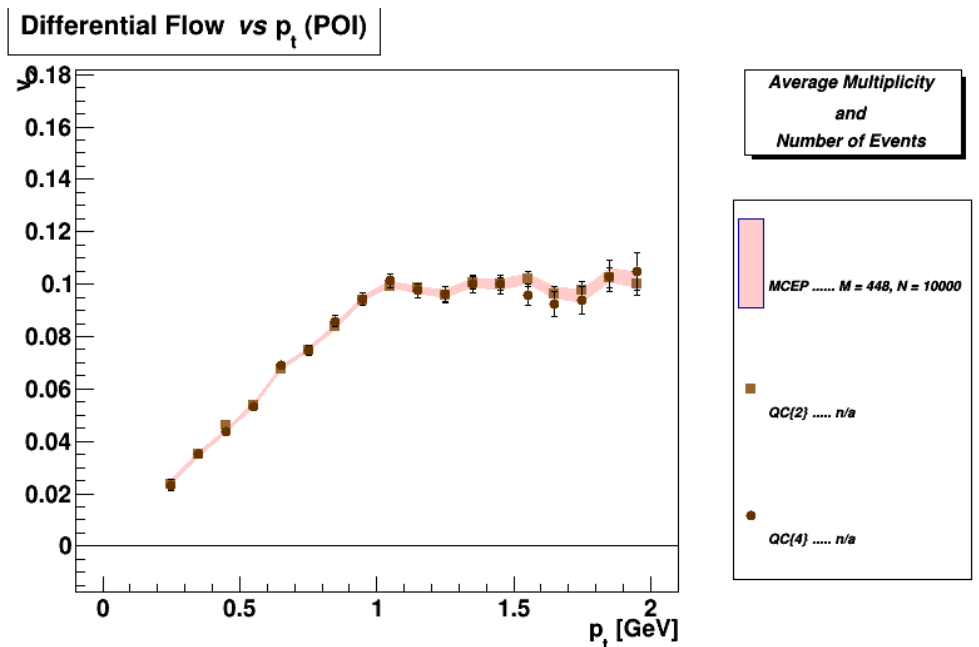


Figure 2.2: Example of inspecting the output file of the on the fly analysis with the compareFlowResults.C macro.

```
277
278 1 new TBrowser();
279
```

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

284 Comparing flow results

285 A convenient way of comparing the results of the different flow analysis strategies that have been used is invoking the macro
 286 compareFlowResults.C^h. This macro will read the analysis output file AnalysisResults.root, extract the requested
 287 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
 288 itself and its ample documentation. To run the macro on the dataset that we have just generated, simply do

```
289
290 1 .L compareFlowResults.C
291 2 compareFlowResults(TSring("")) // the empty suffix indicates on the fly events
292
```

^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 3.4.2 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 `AliFlowEventSimple` 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.

3.2 Flow Event

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.

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 \mathbf{Q} vector(s), and *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!

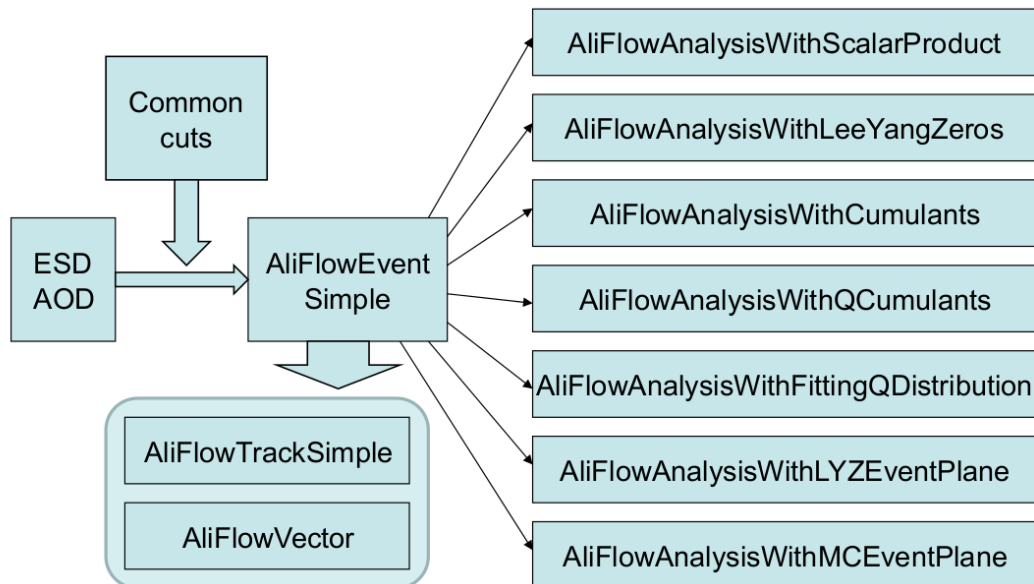


Figure 3.1: Schematic presentation of the organization of the flow package. Input, which can come from any kind of source, is converted to a generic `AliFlowEventSimple` object, which in turn is passed to the analysis methods.

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 description, can be found in the header file of the `AliVEvent` class^a. This function, however, is not implement

^a`$ALICE_ROOT/...`

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 \mathbf{Q} vector distribution. Specific triggers (e.g. EMCAL triggers) result in a \mathbf{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`:

```

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

```

The available cut parameters in the flow event cuts object are

```

1  Bool_t fCutNumberOfTracks; //cut on # of tracks
2  Int_t fNumberOfTracksMax; //limits
3  Int_t fNumberOfTracksMin; //limits
4  Bool_t fCutRefMult; //cut on refmult
5  refMultMethod fRefMultMethod; //how do we calculate refmult?
6  Bool_t fUseAliESDtrackCutsRefMult; //use AliESDtrackCuts for refmult calculation
7  AliESDtrackCuts::MultEstTrackType fRefMultMethodAliESDtrackCuts;
8  Int_t fRefMultMax; //max refmult
9  Int_t fRefMultMin; //min refmult
10 AliFlowTrackCuts* fRefMultCuts; //cuts
11 AliFlowTrackCuts* fMeanPtCuts; //mean pt cuts
12 AliFlowTrackCuts* fStandardTPCcuts; //Standard TPC cuts
13 AliFlowTrackCuts* fStandardGlobalCuts; //StandardGlobalCuts
14 Bool_t fCutPrimaryVertexX; //cut on x of prim vtx
15 Double_t fPrimaryVertexXmax; //max x prim vtx
16 Double_t fPrimaryVertexXmin; //min x prim vtx
17 Bool_t fCutPrimaryVertexY; //cut on y of prim vtx
18 Double_t fPrimaryVertexYmax; //max y prim vtx
19 Double_t fPrimaryVertexYmin; //min y prim vtx
20 Bool_t fCutPrimaryVertexZ; //cut on z of prim vtx
21 Double_t fPrimaryVertexZmax; //max z prim vtx
22 Double_t fPrimaryVertexZmin; //min z prim vtx
23 Bool_t fCutNContributors; //cut on number of contributors
24 Int_t fNContributorsMax; //maximal number of contrib
25 Int_t fNContributorsMin; //minimal number of contrib
26 Bool_t fCutMeanPt; //cut on mean pt
27 Double_t fMeanPtMax; //max mean pt
28 Double_t fMeanPtMin; //min mean pt
29 Bool_t fCutSPDvertexerAnomaly; //cut on the spd vertexer anomaly
30 Bool_t fCutSPDTRKVtxZ; //require compatibility between SPDvertex TRKvertex
31 Bool_t fCutTPCmultiplicityOutliers; //cut TPC multiplicity outliers
32 Bool_t fCutTPCmultiplicityOutliersAOD; // cut TPC outliers in 10h or 11h aod
33 Bool_t fUseCentralityUnchecked; //use the unchecked method
34 refMultMethod fCentralityPercentileMethod; //where to get the percentile from
35 Bool_t fCutZDctiming; //cut on ZDC timing
36 AliTriggerAnalysis fTrigAna; //trigger analysis object
37 Bool_t fCutImpactParameter; //cut on impact parameter (MC header)
38 Double_t fImpactParameterMin; // min impact parameter
39 Double_t fImpactParameterMax; // max impact parameter
40 TH2F *fHistTPCvsGlobalMult; //!correlation between TPCMult and GlobalMult
41 Bool_t fData2011; //2011 data is used

```

all of which are accessible via dedicated setters,

```

1  void SetNumberOfTracksMax(Int_t value) {fNumberOfTracksMax=value;fCutNumberOfTracks=kTRUE;}
2  void SetNumberOfTracksMin(Int_t value) {fNumberOfTracksMin=value;fCutNumberOfTracks=kTRUE;}

```

^b`$ALICE_ROOT/PWG/FLOW/Tasks/AliFlowEventCuts.cxx`

```

438 3 void SetNumberOfTracksRange(Int_t min, Int_t max) {fNumberOfTracksMin=min;fNumberOfTracksMax=max;
439   fCutNumberOfTracks=kTRUE;};
440 4 void SetRefMultMax(Int_t value) {fRefMultMax=value;fCutRefMult=kTRUE;};
441 5 void SetRefMultMin(Int_t value) {fRefMultMin=value;fCutRefMult=kTRUE;};
442 6 void SetRefMultRange(Int_t min, Int_t max) {fRefMultMin=min;fRefMultMax=max;fCutRefMult=kTRUE;};
443 7 void SetImpactParameterMax(Double_t value) {fImpactParameterMax=value;fCutImpactParameter=kTRUE;};
444 8 void SetImpactParameterMin(Double_t value) {fImpactParameterMin=value;fCutImpactParameter=kTRUE;};
445 9 void SetImpactParameterRange(Double_t min, Double_t max) {fImpactParameterMin=min;
446   fImpactParameterMax=max;fCutImpactParameter=kTRUE;};
447 10 void SetPrimaryVertexXrange(Double_t min, Double_t max)
448 11 void SetPrimaryVertexYrange(Double_t min, Double_t max)
449 12 void SetPrimaryVertexZrange(Double_t min, Double_t max)
450 13 void SetNContributorsRange(Int_t min, Int_t max=INT_MAX)
451 14 void SetMeanPtRange(Double_t min, Double_t max) {fCutMeanPt=kTRUE; fMeanPtMax=max; fMeanPtMin=min
452   };}
453 15 void SetCutSPDvertexerAnomaly(Bool_t b=kTRUE) {fCutSPDvertexerAnomaly=b;};
454 16 void SetCutZDCtiming(Bool_t c=kTRUE) {fCutZDCtiming=c;};
455 17 void SetCutSPDTRKVtxZ(Bool_t b=kTRUE) {fCutSPDTRKVtxZ=b;};
456 18 void SetCutTPCmultiplicityOutliers(Bool_t b=kTRUE) {fCutTPCmultiplicityOutliers=b;};
457 19 void SetCutTPCmultiplicityOutliersAOD(Bool_t b=kTRUE) {fCutTPCmultiplicityOutliersAOD=b;};
458 20 void SetRefMultMethod(refMultMethod m) {fRefMultMethod=m;};
459 21 void SetRefMultMethod(AliESDtrackCuts::MultEstTrackType m) { fRefMultMethodAliESDtrackCuts=m;
460 22 void SetRefMultCuts( AliFlowTrackCuts* cuts ) {fRefMultCuts=static_cast<AliFlowTrackCuts*>(cuts->
461   Clone());};
462 23 void SetMeanPtCuts( AliFlowTrackCuts* cuts ) {fMeanPtCuts=static_cast<AliFlowTrackCuts*>(cuts->
463   Clone());};
464 24 void SetQA(Bool_t b=kTRUE) {if (b) DefineHistograms();};
465 25 void SetCentralityPercentileMethod( refMultMethod m) {fCentralityPercentileMethod=m;};
466 26 void SetUseCentralityUnchecked(Bool_t b=kTRUE) {fUseCentralityUnchecked=b;};
467 27 void SetUsedDataset(Bool_t b=kTRUE) {fData2011=b;}; // confusing name, better use different
468   interface
469 28 void SetLHC10h(Bool_t b=kTRUE) {fData2011=!b;}; // TODO let cut object determine runnumber
470   and period
471 29 void SetLHC11h(Bool_t b=kTRUE) {fData2011=b;}; // use this only as 'manual override'
472

```

473 Caveats and remarks

474 Some caveats and remarks about using the event cuts object

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

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

```

482 1 Bool_t AliFlowEventCuts::PassesCuts(AliEvent *event, ALICEEvent *mcevent)
483

```

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

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

```

491 1 Bool_t AliFlowEventCuts::PassesCuts(AliEvent *event, ALICEEvent *mcevent)
492

```

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

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

498 3.3 Track cuts and the track cuts object

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

3.3.1 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 3.3.4), 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`

```

568 1  enum trackParameterType { kMC,
569 2                          kGlobal,
570 3                          kTPCstandalone,
571 4                          kSPDtracklet,
572 5                          kPMD,
573 6                          kV0,      //neutral reconstructed v0 particle
574 7                          kVZERO,  //forward VZERO detector
575 8                          kMUON,
576 9                          kKink,
577 10                         kAODFilterBit,
578 11                         kUserA,    // reserved for custom cuts
579 12                         kUserB    // reserved for custom cuts
580 13 };
581

```

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

585 3.3.2 AOD tracks as data source

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

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

```

595 1  static AliFlowTrackCuts* GetAODTrackCutsForFilterBit(UInt_t bit = 1);
596
597

```

598 which is defined as

```

599 1  AliFlowTrackCuts* AliFlowTrackCuts::GetAODTrackCutsForFilterBit(UInt_t bit)
600 2  {
601 3  // object which in its default form only cuts on filterbit (for AOD analysis)
602 4  AliFlowTrackCuts* cuts = new AliFlowTrackCuts(Form("AOD filterbit %i", (int)bit));
603 5  cuts->SetMinimalTPCdedx(-999999999);
604 6  cuts->SetAODfilterBit(bit);
605 7  cuts->SetParamType(AliFlowTrackCuts::kAODFilterBit);
606 8  return cuts;
607 9  }
608

```

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

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

612 3.3.3 Additional options

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

617 Identified particles

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

```

620 1  void AliFlowTrackCuts::SetPID(
621 2  AliPID::EParticleType pid,
622 3  PIDsource s=kTOFPid,
623 4  Double_t prob=0.9)
624 5  {fParticleID=pid; fPIDsource=s; fParticleProbability=prob; fCutPID=kTRUE; InitPIDcuts();
625 6  }
626

```

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

^d`$ALICE_ROOT/STEER/STEERBas/AliPID.h`

```

630
631 1   enum EParticleType {
632 2     kElectron = 0,
633 3     kMuon = 1,
634 4     kPion = 2,
635 5     kKaon = 3,
636 6     kProton = 4,
637 7
638 8     kDeuteron = 5,
639 9     kTriton = 6,
640 10    kHe3 = 7,
641 11    kAlpha = 8,
642 12
643 13    kPhoton = 9,
644 14    kPi0 = 10,
645 15    kNeutron = 11,
646 16    kKaon0 = 12,
647 17    kEleCon = 13,
648 18
649 19    kUnknown = 14
650 20 };

```

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

```

655
656 1   enum PIDsource {
657 2     kTPCpid,           // default TPC pid (via GetTPCpid)
658 3     kTOFpid,          // default TOF pid (via GetTOFpid)
659 4     kTOFbayesian,    // TOF bayesian pid (F.Noferini)
660 5     kTOFbeta,        // asymmetric cuts of TOF beta signal
661 6     kTPCdedx,         // asymmetric cuts of TPC dedx signal
662 7     kTOFbetaSimple,  // simple TOF only cut
663 8     kTPCbayesian,    // bayesian cutTPC
664 9     kTPCNuclei,      // added by Natasha for Nuclei
665 10    kTPCTOFNsigma   // simple cut on combined tpc tof nsigma
666 11 };

```

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 $n\text{-}\sigma$ 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:

```

686
687 1   gROOT->LoadMacro("ANALYSIS/macros/AddTaskPIDResponse.C");
688 2   AddTaskPIDResponse();

```

The default value for $n\text{-}\sigma$ is 3, but it can be set to a different value using

```

691
692 1   void AliFlowTrackCuts::SetNumberOfSigmas(Float_t val);
693

```


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.

```

700 1 Bool_t AliFlowTrackCuts::PassesAODpidCut(const AliAODTrack* track )
701 2 {
702 3     if(!track->GetAODEvent()->GetTOFHeader()){
703 4         AliAODPid *pidObj = track->GetDetPid();
704 5         if (!pidObj) fESDPid.GetTOFResponse().SetTimeResolution(84.);
705 6         else{
706 7             Double_t sigmaTOFPidInAOD[10];
707 8             pidObj->GetTOFPidResolution(sigmaTOFPidInAOD);
708 9             if(sigmaTOFPidInAOD[0] > 84.){
709 10                fESDPid.GetTOFResponse().SetTimeResolution(sigmaTOFPidInAOD[0]); // use the electron
710 11                TOF PID sigma as time resolution (including the TO used)
711 12            }
712 13        }
713 14    }
714 15 //check if passes the selected pid cut for ESDs
715 16 Bool_t pass = kTRUE;
716 17 switch (fPIDsource)
717 18 {
718 19     case kTOFbeta:
719 20         if (!PassesTOFbetaCut(track)) pass=kFALSE;
720 21         break;
721 22     case kTOFbayesian:
722 23         if (!PassesTOFbayesianCut(track)) pass=kFALSE;
723 24         break;
724 25     case kTPCbayesian:
725 26         if (!PassesTPCbayesianCut(track)) pass=kFALSE;
726 27         break;
727 28     case kTPCTOFNsigma:
728 29         if (!PassesTPCTOFNsigmaCut(track)) pass = kFALSE;
729 30         break;
730 31     default:
731 32         return kTRUE;
732 33         break;
733 34 }
734 35 return pass;
735 36 }
736 37 }
737 38 //-----
738 39 Bool_t AliFlowTrackCuts::PassesESDpidCut(const AliESDtrack* track )
739 40 {
740 41 //check if passes the selected pid cut for ESDs
741 42 Bool_t pass = kTRUE;
742 43 switch (fPIDsource)
743 44 {
744 45     case kTPCpid:
745 46         if (!PassesTPCpidCut(track)) pass=kFALSE;
746 47         break;
747 48     case kTPCdedx:
748 49         if (!PassesTPCdedxCut(track)) pass=kFALSE;
749 50         break;
750 51     case kTOFpid:
751 52         if (!PassesTOFpidCut(track)) pass=kFALSE;
752 53         break;
753 54     case kTOFbeta:
754 55         if (!PassesTOFbetaCut(track)) pass=kFALSE;
755 56         break;
756 57     case kTOFbetaSimple:
757 58         if (!PassesTOFbetaSimpleCut(track)) pass=kFALSE;
758 59         break;
759 60     case kTPCbayesian:
760 61         if (!PassesTPCbayesianCut(track)) pass=kFALSE;
761 62         break;
762 63     case kTOFbayesian:
763 64         if (!PassesTOFbayesianCut(track)) pass=kFALSE;
764 65         break;
765 66     case kTPCNuclei:
766 67         if (!PassesNucleiSelection(track)) pass=kFALSE;
767 68         break;
768 69     case kTPCTOFNsigma:
769 70         if (!PassesTPCTOFNsigmaCut(track)) pass = kFALSE;
770 71

```

```

772     71     break;
773     72     default:
774     73     printf("AliFlowTrackCuts::PassesCuts() this should never be called!\n");
775     74     pass=kFALSE;
776     75     break;
777     76 }
778     77     return pass;
779     78 }

```

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 πv_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,

```

792 1 Bool_t AliFlowTrackCuts::IsSelected(TObject* obj, Int_t id)
793

```

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 \mathbf{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 \mathbf{Q} vector.

In general, a \mathbf{Q} vector is defined as

$$\mathbf{Q} = \sum_{\text{tracks}} w_i \exp(in\varphi) \quad (3.3.3.1)$$

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 runnumber dependence, which results in a non-flat VZERO \mathbf{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 runnumbers 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 \mathbf{Q} vectors.

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

```

819 1 // Q-vector calculation
820 2 AliFlowVector AliFlowEventSimple::GetQ(
821 3     Int_t n,           // harmonic
822 4     TList *weightsList, // weight list
823 5     Bool_t usePhiWeights, // use phi weights?
824 6     Bool_t usePtWeights, // use pt weights?
825 7     Bool_t useEtaWeights // use eta weights?
826 8 )
827 9
828 10 // Q-vectors of sub-events
829 11 void AliFlowEventSimple::Get2Qsub(
830 12     AliFlowVector* Qarray, // array with q-vectors
831 13     Int_t n,

```

```

833 14     TList *weightsList,
834 15     Bool_t usePhiWeights,
835 16     Bool_t usePtWeights,
836 17     Bool_t useEtaWeights
837 18     )
838 19
839 20 // overloaded implementation of Q-vectors of sub-events for VZERO information
840 21 void AliFlowEvent::Get2Qsub(
841 22     AliFlowVector* Qarray,
842 23     Int_t n,
843 24     TList *weightsList,
844 25     Bool_t usePhiWeights,
845 26     Bool_t usePtWeights,
846 27     Bool_t useEtaWeights
847 28     )

```

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

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

- 855 • The first step is evaluating the \mathbf{Q} vector using equation 3.3.3.1. However, the VZERO signal of each tile is
856 *re-weighted* before it is used as a weight in equation 3.3.3.1. The re-weighting comprises
 - 857 1. Taking a TProfile with average multiplicity per cell (these profiles are stored in a OADB file for each
858 run-number)
 - 859 2. Fitting a constant line per disc (or ring) $y = a$ (see next slide for example)
 - 860 3. Evaluating the track weight for each VZERO cell is now calculated in a second iteration as

$$\text{track weight} = \frac{\text{cell multiplicity} * a}{\text{average multiplicity in a cell}} \quad (3.3.3.2)$$

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

$$Q_n \rightarrow \frac{Q_n - \langle Q_n \rangle}{\sigma_{Q_n}} \quad (3.3.3.3)$$

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

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

```

869 1 AliVEvent::GetVZEROEqMultiplicity(Int_t i);
870

```

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

```

875 1 TH1F*      GetHistQ()                {return fHistQ; } ;
876 2 TH1F*      GetHistAngleQ()           {return fHistAngleQ; }
877 3 TH1F*      GetHistAngleQSub0()       {return fHistAngleQSub0; }
878 4 TH1F*      GetHistAngleQSub1()       {return fHistAngleQSub1; }
879

```

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

```

886 1 gROOT->LoadMacro("$ALICE_ROOT/ANALYSIS/macros/AddTaskVZEROEPSelection.C");
887 2 AddTaskVZEROEPSelection();
888

```

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 \mathbf{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,

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

At run-time, the flow package will detector whether LHC10h or LHC11h data is used by reading the analyzed events' runnumber. 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

```
1 AliFlowTrackCuts* AliFlowTrackCuts::GetStandardVZEROOnlyTrackCuts2010()
2 {
3     //get standard VZERO cuts
4     //DISCLAIMER: LHC10h VZERO calibration consists (by default) of two steps
5     //1) re-weighting of signal
6     //2) re-centering of q-vectors
7     //step 2 is available only for n==2 and n==3, for the higher harmonics the user
8     //is repsonsible for making sure the q-sub distributions are (sufficiently) flat
9     //or a sensible NUA procedure is applied !
10    AliFlowTrackCuts* cuts = new AliFlowTrackCuts("standard vzero flow cuts");
11    cuts->SetParamType(AliFlowTrackCuts::kVZERO);
12    cuts->SetEtaRange( -10, +10 );
13    cuts->SetEtaGap(-1., 1.);
14    cuts->SetPhiMin( 0 );
15    cuts->SetPhiMax( TMath::TwoPi() );
16    // options for the reweighting
17    cuts->SetVZEROGainEqualizationPerRing(kFALSE);
18    cuts->SetApplyRecentering(kTRUE);
19    // to exclude a ring , do e.g.
20    // cuts->SetUseVZERORing(7, kFALSE);
21    // excluding a ring will break the re-centering as re-centering relies on a
22    // database file which tuned to receiving info from all rings
23    return cuts;
24 }
25 //-----
26 AliFlowTrackCuts* AliFlowTrackCuts::GetStandardVZEROOnlyTrackCuts2011()
27 {
28     //get standard VZERO cuts for 2011 data
29     //in this case, the vzero segments will be weighted by
30     //VZEROEqMultiplicity,
31     //if recentering is enableded, the sub-q vectors
32     //will be taken from the event header, so make sure to run
33     //the VZERO event plane selection task before this task !
34     //DISCLAIMER: recentering is only available for n==2
35     //for the higher harmonics the user
36     //is repsonsible for making sure the q-sub distributions are (sufficiently) flat
37     //or a sensible NUA procedure is applied !
38     //recentering replaces the already evaluated q-vectors, so
39     //when chosen, additional settings (e.g. excluding rings)
40     //have no effect. recentering is true by default
41     //
42     //NOTE user is responsible for running the vzero event plane
43     //selection task in advance, e.g. add to your launcher macro
44     //
45     // gROOT->LoadMacro("$ALICE_ROOT/ANALYSIS/macros/AddTaskVZEROEPSelection.C");
46     // AddTaskVZEROEPSelection();
47     //
48     AliFlowTrackCuts* cuts = new AliFlowTrackCuts("standard vzero flow cuts 2011");
49     cuts->SetParamType(kVZERO);
50     cuts->SetEtaRange( -10, +10 );
51     cuts->SetEtaGap(-1., 1.);
52     cuts->SetPhiMin( 0 );
53     cuts->SetPhiMax( TMath::TwoPi() );
54     cuts->SetApplyRecentering(kTRUE);
55     cuts->SetVZEROGainEqualizationPerRing(kFALSE);
56     return cuts;
57 }
```

960 Caveats and remarks

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

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

```

964 1 void AliFlowEvent::Get2Qsub()
965

```

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 subevents 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:

```

980 1 // to either enable or disable the recentering
981 2 // (for 11h this will mean that no calibration is performed,
982 3 // for 10h it will result in only doing a re-weighting)
984 4 void SetApplyRecentering(Bool_t r)
985 5 // to enable a per-ring instead of per-disc gain equalization (=re-weighting)
986 6 // (for 11h this has no effect)
987 7 void SetVZEROGainEqualizationPerRing(Bool_t s)
988 8 // exclude vzero rings: 0 through 7 can be excluded by calling this setter multiple times
989 9 // 0 corresponds to segment ID 0 through 7, etc
990 10 // disabled vzero rings get weight 0
991 11 // with this function you can omit information from entire vzero rings
992 12 // might be useful for runs where there is a bad signal in one of the tiles
993 13 // (sometimes referred to as 'clipping')
994 14 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 \mathbf{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 known 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 \mathbf{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

```

1009 1 // set weight list
1010 2 AliFlowAnalysisWith*::SetWeightsList(TList* const)
1011 3 // toggle phi weights on / off
1012 4 AliFlowAnalysisWith*::SetUsePhiWeights(Bool_t const)
1013 5 // toggle eta weights on / off
1014 6 AliFlowAnalysisWith*::SetUseEtaWeights(Bool_t const)
1015 7 // toggle pt weights on / off
1016 8 AliFlowAnalysisWith*::SetUsePtWeights(Bool_t const)
1017

```

and are applied to total \mathbf{Q} vectors and sub-event \mathbf{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

```
1 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

```
1 AliFlowCommonConstants* cc = AliFlowCommonConstants::GetMaster();
2 cc->SetNbinsPt(100);
3 cc->SetPtMin(0);
4 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

```
1 //histogram sizes
2 Int_t fNbinsMult; // histogram size
3 Int_t fNbinsPt; // histogram size
4 Int_t fNbinsPhi; // histogram size
5 Int_t fNbinsEta; // histogram size
6 Int_t fNbinsQ; // histogram size
7 Int_t fNbinsMass; // histogram size
8
9 // Histograms limits
10 Double_t fMultMin; // histogram limit
11 Double_t fMultMax; // histogram limit
12 Double_t fPtMin; // histogram limit
13 Double_t fPtMax; // histogram limit
14 Double_t fPhiMin; // histogram limit
15 Double_t fPhiMax; // histogram limit
16 Double_t fEtaMin; // histogram limit
17 Double_t fEtaMax; // histogram limit
18 Double_t fQMin; // histogram limit
19 Double_t fQMax; // histogram limit
20 Double_t fMassMin; // histogram limit
21 Double_t fMassMax; // histogram limit
22 Double_t fHistWeightvsPhiMin; // histogram limit
23 Double_t fHistWeightvsPhiMax; // histogram limit
```

via the setters

```
1 void SetNbinsMult( Int_t i ) { fNbinsMult = i; }
2 void SetNbinsPt( Int_t i ) { fNbinsPt = i; }
3 void SetNbinsPhi( Int_t i ) { fNbinsPhi = i; }
4 void SetNbinsEta( Int_t i ) { fNbinsEta = i; }
5 void SetNbinsQ( Int_t i ) { fNbinsQ = i; }
6 void SetNbinsMass( Int_t i ) { fNbinsMass = i; }
7 void SetMultMin( Double_t i ) { fMultMin = i; }
8 void SetMultMax( Double_t i ) { fMultMax = i; }
9 void SetPtMin( Double_t i ) { fPtMin = i; }
10 void SetPtMax( Double_t i ) { fPtMax = i; }
11 void SetPhiMin( Double_t i ) { fPhiMin = i; }
12 void SetPhiMax( Double_t i ) { fPhiMax = i; }
13 void SetEtaMin( Double_t i ) { fEtaMin = i; }
14 void SetEtaMax( Double_t i ) { fEtaMax = i; }
15 void SetQMin( Double_t i ) { fQMin = i; }
16 void SetQMax( Double_t i ) { fQMax = i; }
17 void SetMassMin( Double_t i ) { fMassMin = i; }
18 void SetMassMax( Double_t i ) { fMassMax = i; }
19 void SetHistWeightvsPhiMax( Double_t d ) {fHistWeightvsPhiMax=d;}
20 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,

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

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

```
1 // open the file
2 TFile f("AnalysisResults.root");
3 // get the qc analysis output directory
4 TDirectoryFile* dir = (TDirectoryFile*)f.Get("outputQCanalysis");
5 // and retrieve the output list of the analysis
6 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

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

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

```
1 // get common histogram object from the TList
2 AliFlowCommonHist* commonHist = (AliFlowCommonHist*)outputList->FindObject("AliFlowCommonHistQC");
3 // get the results for the 2 particle cumulant from the TList
4 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

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

The following getters are available in `AliFlowCommonHist`

```
1 Double_t GetEntriesInPtBinRP(Int_t iBin); //gets entries from fHistPtRP
2 Double_t GetEntriesInPtBinPOI(Int_t iBin); //gets entries from fHistPtPOI
3 Double_t GetEntriesInEtaBinRP(Int_t iBin); //gets entries from fHistEtaRP
4 Double_t GetEntriesInEtaBinPOI(Int_t iBin); //gets entries from fHistEtaPOI
5 Double_t GetMeanPt(Int_t iBin); //gets the mean pt for this bin from
   fHistProMeanPtperBin
6 TH1F* GetHistMultRP() {return fHistMultRP; } ;
7 TH1F* GetHistMultPOI() {return fHistMultPOI; } ;
8 TH2F* GetHistMultPOIvsRP() {return fHistMultPOIvsRP; } ;
9 TH1F* GetHistPtRP() {return fHistPtRP; } ;
10 TH1F* GetHistPtPOI() {return fHistPtPOI; } ;
11 TH1F* GetHistPtSub0() {return fHistPtSub0; } ;
12 TH1F* GetHistPtSub1() {return fHistPtSub1; } ;
13 TH1F* GetHistPhiRP() {return fHistPhiRP; } ;
14 TH1F* GetHistPhiPOI() {return fHistPhiPOI; } ;
15 TH1F* GetHistPhiSub0() {return fHistPhiSub0; } ;
16 TH1F* GetHistPhiSub1() {return fHistPhiSub1; } ;
17 TH1F* GetHistEtaRP() {return fHistEtaRP; } ;
18 TH1F* GetHistEtaPOI() {return fHistEtaPOI; } ;
19 TH1F* GetHistEtaSub0() {return fHistEtaSub0; } ;
20 TH1F* GetHistEtaSub1() {return fHistEtaSub1; } ;
21 TH2F* GetHistPhiEtaRP() {return fHistPhiEtaRP; } ;
22 TH2F* GetHistPhiEtaPOI() {return fHistPhiEtaPOI; } ;
23 TProfile* GetHistProMeanPtperBin() {return fHistProMeanPtperBin; } ;
24 TH2F* GetHistWeightvsPhi() {return fHistWeightvsPhi; } ;
25 TH1F* GetHistQ() {return fHistQ; } ;
26 TH1F* GetHistAngleQ() {return fHistAngleQ; } ;
27 TH1F* GetHistAngleQSub0() {return fHistAngleQSub0; } ;
28 TH1F* GetHistAngleQSub1() {return fHistAngleQSub1; } ;
29 TProfile* GetHarmonic() {return fHarmonic; } ;
30 TProfile* GetRefMultVsNoOfRPs() {return fRefMultVsNoOfRPs; } ;
31 TH1F* GetHistRefMult() {return fHistRefMult; } ;
32 TH2F* GetHistMassPOI() {return fHistMassPOI; } ;
33 TList* GetHistList() {return fHistList; } ;
```

^eA ‘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

```

1176 1 TH1D* GetHistChi() {return fHistChi;};
1177 2 TH1D* GetHistIntFlow() {return fHistIntFlow;};
1178 3 TH1D* GetHistIntFlowRP() {return fHistIntFlowRP;};
1179 4 TH1D* GetHistDiffFlowPtRP() {return fHistDiffFlowPtRP;};
1180 5 TH1D* GetHistDiffFlowEtaRP() {return fHistDiffFlowEtaRP;};
1181 6 TH1D* GetHistIntFlowPOI() {return fHistIntFlowPOI;};
1182 7 TH1D* GetHistDiffFlowPtPOI() {return fHistDiffFlowPtPOI;};
1183 8 TH1D* GetHistDiffFlowEtaPOI() {return fHistDiffFlowEtaPOI;};
1184 9 TList* GetHistList() {return fHistList;};
1185

```

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

$$\frac{dN}{d\varphi_0} = \frac{1}{2\pi}. \quad (3.3.3.4)$$

Adding a periodic azimuthal modulation, this is translated to

$$\frac{dN}{d\varphi} = \frac{1}{2\pi} (1 + v_2 \cos[2(\varphi - \Psi)]) \quad (3.3.3.5)$$

which can be re-written as

$$\frac{dN}{d\varphi} = \frac{dN}{d\varphi_0} \frac{d\varphi_0}{d\varphi} = \frac{1}{2\pi} \frac{d\varphi_0}{d\varphi} \quad (3.3.3.6)$$

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

$$\varphi = \varphi_0 - v_2 \sin[2(\varphi - \Psi)]. \quad (3.3.3.7)$$

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

$$\frac{dN}{d\varphi} \propto 1 + \frac{1}{2} \sum_n v_n (n\Delta\varphi). \quad (3.3.3.8)$$

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

```

1200 1 // setters for adding by hand flow values (afterburner)
1201 2
1202 3 // toggle the afterburner on / off
1203 4 void SetAfterburnerOn(Bool_t b=kTRUE) {fAfterburnerOn=b;}
1204 5 // set differential v2 via a TF1
1205 6 void SetPtDifferentialV2( TF1 *gPtV2) {fDifferentialV2 = gPtV2;}
1206 7 // set integrated flow (used when the gPtV2 = NULL)
1207 8 void SetFlow( Double_t v1, Double_t v2, Double_t v3=0.0, Double_t v4=0.0, Double_t v5=0.0)
1208 9 {fV1=v1; fV2=v2; fV3=v3; fV4=v4; fV5=v5;}
1209

```

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

```

1213 1 void SetNonFlowNumberOfTrackClones(Int_t n) {fNonFlowNumberOfTrackClones=n;}
1214

```

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 .

3.3.4 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.

```

1 void AliAnalysisTaskFlowEvent::UserExec(Option_t *)
2 {
3     // Main loop
4     // Called for each event
5     //delete fFlowEvent;
6     AliMCEvent* mcEvent = MCEvent(); // from TaskSE
7     AliESDEvent* myESD = dynamic_cast<AliESDEvent*>(InputEvent()); // from TaskSE
8     AliAODEvent* myAOD = dynamic_cast<AliAODEvent*>(InputEvent()); // from TaskSE
9
10    // the rp and poi cuts will be used to fill the flow event
11    // so they have to be defined here
12    if (!(fCutsRP&&fCutsPOI&&fCutsEvent))
13    {
14        AliError("cuts not set");
15        return;
16    }
17
18    //DEFAULT - automatically takes care of everything
19    // the flow package will determine the datatype that you are using
20    if (fAnalysisType == "AUTOMATIC")
21    {
22        //check event cuts
23        if (InputEvent() && !fCutsEvent->IsSelected(InputEvent(),MCEvent()))
24            return;
25
26        //first attach all possible information to the cuts
27        // the track cuts will make the track selection, so they
28        // have to be supplied with the current event
29        // the mc event is NULL unless it is retrieved by AliAnalysisTaskSE
30        fCutsRP->SetEvent( InputEvent(), MCEvent() ); //attach event
31        fCutsPOI->SetEvent( InputEvent(), MCEvent() );
32
33        //then make the event
34        // this function will fill the flow event with selected poi's and rp's
35        // the implementation is printed below
36        fFlowEvent->Fill( fCutsRP, fCutsPOI );
37
38        // pass some event info to the flow event
39        fFlowEvent->SetReferenceMultiplicity(fCutsEvent->GetReferenceMultiplicity(InputEvent(),mcEvent));
40        ;
41        fFlowEvent->SetCentrality(fCutsEvent->GetCentrality(InputEvent(),mcEvent));
42        if (mcEvent && mcEvent->GenEventHeader()) fFlowEvent->SetMCReactionPlaneAngle(mcEvent);
43    }
44
45    // a lot of code is omitted here //
46
47    ////////////////////////////////////////
48    ////////////////////////////////////////AFTERBURNER
49    if (fAfterburnerOn)
50    {
51        //if reaction plane not set from elsewhere randomize it before adding flow
52        if (!fFlowEvent->IsSetMCReactionPlaneAngle())
53            fFlowEvent->SetMCReactionPlaneAngle(gRandom->Uniform(0.0, TMath::TwoPi()));
54
55        if(fDifferentialV2)
56            fFlowEvent->AddV2(fDifferentialV2);
57        else
58            fFlowEvent->AddFlow(fV1,fV2,fV3,fV4,fV5); //add flow
59            fFlowEvent->CloneTracks(fNonFlowNumberOfTrackClones); //add nonflow by cloning tracks
60    }
61
62    ////////////////////////////////////////
63
64    //tag subEvents
65    // some flow analysis methods (such as the scalar product)
66    // use sub-events. by calling this function, all tracks in the
67    // flow event are tagged as belonging to either sub-event a or b
68    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 ...).

```

1302 1 //-----
1303 2 void AliFlowEvent::Fill( AliFlowTrackCuts* rpCuts,
1304 3                          AliFlowTrackCuts* poiCuts )
1305 3
1306 4 {
1307 5     //Fills the event from a vevent: AliESDEvent, AliAODEvent, AliMCEvent
1308 6     //the input data needs to be attached to the cuts
1309 7     //we have two cases, if we're cutting the same collection of tracks
1310 8     //(same param type) then we can have tracks that are both rp and poi
1311 9     //in the other case we want to have two exclusive sets of rps and pois
1312 10    //e.g. one tracklets, the other PMD or global - USER IS RESPONSIBLE
1313 11    //FOR MAKING SURE THEY DONT OVERLAP OR ELSE THE SAME PARTICLE WILL BE
1314 12    //TAKEN TWICE
1315 13
1316 14    // remove the previous event
1317 15    ClearFast();
1318 16    if (!rpCuts || !poiCuts) return;
1319 17    // check the source of rp's
1320 18    AliFlowTrackCuts::trackParameterType sourceRP = rpCuts->GetParamType();
1321 19    // and ditto for the poi's
1322 20    AliFlowTrackCuts::trackParameterType sourcePOI = poiCuts->GetParamType();
1323 21
1324 22    AliFlowTrack* pTrack=NULL;
1325 23
1326 24    // if the source for rp's or poi's is the VZERO detector, get the calibration
1327 25    // and set the calibration parameters
1328 26    if (sourceRP == AliFlowTrackCuts::kVZERO) {
1329 27        SetVZEROCalibrationForTrackCuts(rpCuts);
1330 28        if(!rpCuts->GetApplyRecentering()) {
1331 29            // if the user does not want to recenter, switch the flag
1332 30            fApplyRecentering = -1;
1333 31        }
1334 32        // note: this flag is used in the overloaded implementation of Get2Qsub()
1335 33        // and tells the function to use as Qsub vectors the recentered Q-vectors
1336 34        // from the VZERO oadb file or from the event header
1337 35    }
1338 36    if (sourcePOI == AliFlowTrackCuts::kVZERO) {
1339 37        // probably no-one will choose vzero tracks as poi's ...
1340 38        SetVZEROCalibrationForTrackCuts(poiCuts);
1341 39    }
1342 40
1343 41
1344 42    if (sourceRP==sourcePOI)
1345 43    {
1346 44        //loop over tracks
1347 45        Int_t numberOfInputObjects = rpCuts->GetNumberOfInputObjects();
1348 46        for (Int_t i=0; i<numberOfInputObjects; i++)
1349 47        {
1350 48            //get input object (particle)
1351 49            TObject* particle = rpCuts->GetInputObject(i);
1352 50
1353 51            Bool_t rp = rpCuts->IsSelected(particle,i);
1354 52            Bool_t poi = poiCuts->IsSelected(particle,i);
1355 53
1356 54            if (!(rp||poi)) continue;
1357 55
1358 56            //make new AliFlowTrack
1359 57            if (rp)
1360 58            {
1361 59                pTrack = rpCuts->FillFlowTrack(fTrackCollection, fNumberOfTracks);
1362 60                if (!pTrack) continue;
1363 61                pTrack->Tag(0); IncrementNumberOfPOIs(0);
1364 62                if (poi) {pTrack->Tag(1); IncrementNumberOfPOIs(1);}
1365 63                if (pTrack->GetNDaughters()>0) fMothersCollection->Add(pTrack);
1366 64            }
1367 65            else if (poi)
1368 66            {
1369 67                pTrack = poiCuts->FillFlowTrack(fTrackCollection, fNumberOfTracks);
1370 68                if (!pTrack) continue;
1371 69                pTrack->Tag(1); IncrementNumberOfPOIs(1);
1372 70                if (pTrack->GetNDaughters()>0) fMothersCollection->Add(pTrack);
1373 71            }
1374 72            fNumberOfTracks++;
1375 73        } //end of while (i < numberOfTracks)
1376 74    }
1377 75    else if (sourceRP!=sourcePOI)
1378 76    {
1379 77        //here we have two different sources of particles, so we fill
1380 78        //them independently
1381 79        //POI

```

```

1382 80   for (Int_t i=0; i<poiCuts->GetNumberOfInputObjects(); i++)
1383 81   {
1384 82       TObject* particle = poiCuts->GetInputObject(i);
1385 83       Bool_t poi = poiCuts->IsSelected(particle,i);
1386 84       if (!poi) continue;
1387 85       pTrack = poiCuts->FillFlowTrack(fTrackCollection,fNumberOfTracks);
1388 86       if (!pTrack) continue;
1389 87       pTrack->Tag(1);
1390 88       IncrementNumberOfPOIs(1);
1391 89       fNumberOfTracks++;
1392 90       if (pTrack->GetNDaughters()>0) fMothersCollection->Add(pTrack);
1393 91   }
1394 92   //RP
1395 93   Int_t numberOfInputObjects = rpCuts->GetNumberOfInputObjects();
1396 94   for (Int_t i=0; i<numberOfInputObjects; i++)
1397 95   {
1398 96       TObject* particle = rpCuts->GetInputObject(i);
1399 97       Bool_t rp = rpCuts->IsSelected(particle,i);
1400 98       if (!rp) continue;
1401 99       pTrack = rpCuts->FillFlowTrack(fTrackCollection,fNumberOfTracks);
1402 100      if (!pTrack) continue;
1403 101      pTrack->Tag(0);
1404 102      IncrementNumberOfPOIs(0);
1405 103      fNumberOfTracks++;
1406 104      if (pTrack->GetNDaughters()>0) fMothersCollection->Add(pTrack);
1407 105  }
1408 106  }
1409 107  }

```

3.4 The flow package in 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.

3.4.1 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;

^fThis 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 3.4.2, 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**[§] 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

```
1 .L redoFinish.C
2 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:

```
1 // Macro redoFinish.C is typically used after the merging macros (mergeOutput.C or
2 // mergeOutputOnGrid.C) have been used to produce the merged, large statistics
3 // file of flow analysis. Results stored in merged file are WRONG because after
4 // merging the results from small statistics files are trivially summed up in all
5 // histograms. This is taken into account and corrected for with macro redoFinish.C.
6 // Another typical use of the macro redoFinish.C is to repeat the call to Finish()
7 // in all classes, but with different values of some settings which might modify
8 // the final results (Example: redo the Finish() and apply correction for detector
9 // effects in QC code because by default this correction is switched off).
10
11 // Name of the merged, large statistics file obtained with the merging macros:
12 TString mergedFileName = "mergedAnalysisResults.root";
13 // Final output file name holding correct final results for large statistics sample:
14 TString outputFileName = "AnalysisResults.root";
15
16 Bool_t bApplyCorrectionForNUA = kFALSE; // apply correction for non-uniform acceptance
17 Bool_t bApplyCorrectionForNUAVsM = kFALSE; // apply correction for non-uniform acceptance in each
18 // multiplicity bin independently
19 Bool_t bPropagateErrorAlsoFromNIT = kFALSE; // propagate error also from non-isotropic terms
20 Bool_t bMinimumBiasReferenceFlow = kTRUE; // store in CRH for reference flow the result obtained
21 // without rebinning in multiplicity (kTRUE)
22 Bool_t checkForCommonHistResults = kTRUE; // check explicitly if the TList AliFlowCommonHistResults
23 // 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!

[§]`$ALICE_ROOT/PWCGF/FLOW/macros/redoFinish.C`

3.4.2 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 (colloquially 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

```

1 // load libraries
2 gSystem->Load("libCore.so");
3 gSystem->Load("libGeom.so");
4 gSystem->Load("libVMC.so");
5 gSystem->Load("libPhysics.so");
6 gSystem->Load("libTree.so");
7 gSystem->Load("libSTEERBase.so");
8 gSystem->Load("libESD.so");
9 gSystem->Load("libAOD.so");
10 gSystem->Load("libANALYSIS.so");
11 gSystem->Load("libANALYSISalice.so");
12 gSystem->Load("libEventMixing.so");
13 gSystem->Load("libCORRFW.so");
14 gSystem->Load("libPWGTools.so");
15 gSystem->Load("libPWGCFebye.so");
16 gSystem->Load("libPWGflowBase.so");
17 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

```

1 // create the analysis manager
2 AliAnalysisManager* mgr = new AliAnalysisManager("MyManager");
3 // create a tchain which will point to an aod tree
4 TChain* chain = new TChain("aodTree");
5 // add a few files to the chain
6 chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/
7 AOD086/0003/AliAOD.root");
8 chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/
9 AOD086/0004/AliAOD.root");
10 chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/
11 AOD086/0005/AliAOD.root");
12 chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/
13 AOD086/0006/AliAOD.root");
14 chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/
15 AOD086/0007/AliAOD.root");
16 chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/
17 AOD086/0008/AliAOD.root");
18 chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/
19 AOD086/0009/AliAOD.root");
20 chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/
21 AOD086/0010/AliAOD.root");
22 // create an input handler
23 AliVEventHandler* inputH = new AliAODInputHandler();
24 // and connect it to the manager
25 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:

```

1 // the manager is static, so get the existing manager via the static method
2 AliAnalysisManager *mgr = AliAnalysisManager::GetAnalysisManager();
3 if (!mgr) {
4     printf("No analysis manager to connect to!\n");
5     return NULL;
6 }
7
8 // just to see if all went well, check if the input event handler has been connected
9 if (!mgr->GetInputEventHandler()) {
10    printf("This task requires an input event handler!\n");
11    return NULL;
12 }

```

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

```

1 // create instance of the class. because possible qa plots are added in a second output slot,
2 // the flow analysis task must know if you want to save qa plots at the time of class
3 // construction
4 Bool_t doQA = kTRUE;
5 // create instance of the class
6 AliAnalysisTaskFlowEvent* taskFE = new AliAnalysisTaskFlowEvent("FlowEventTask", "", doQA);
7 // add the task to the manager
8 mgr->AddTask(taskFE);
9 // enable QA plots (will give you more insight into what you're doing)
10 taskFE->SetQAOn(kTRUE);
11 // set the trigger selection
12 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 \mathbf{Q} vector distribution. Specific triggers (e.g. EMCAL triggers) result in a \mathbf{Q} vector bias which should *not* be corrected as they invalidate that assumptionⁱ.

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

```

1 // define the event cuts object
2 AliFlowEventCuts* cutsEvent = new AliFlowEventCuts("EventCuts");
3 // configure some event cuts, starting with centrality
4 cutsEvent->SetCentralityPercentileRange(20., 30.);
5 // method used for centrality determination
6 cutsEvent->SetCentralityPercentileMethod(AliFlowEventCuts::kV0);
7 // vertex-z cut
8 cutsEvent->SetPrimaryVertexZrange(-10., 10.);
9 // enable the qa plots
10 cutsEvent->SetQA(doQA);
11 // explicit multiplicity outlier cut
12 cutsEvent->SetCutTPCMultiplicityOutliersAOD(kTRUE);
13 cutsEvent->SetLHC10h(kTRUE);
14
15
16 // and, last but not least, pass these cuts to your flow event task
17 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,

```

1 // create the track cuts object using a static function of AliFlowTrackCuts
2 AliFlowTrackCuts* cutsRP = AliFlowTrackCuts::GetAODTrackCutsForFilterBit(1, "RP cuts");

```

^hIn the example macro this is 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.

ⁱThe 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`.

```

1638 3 // specify the pt range
1639 4 cutsRP->SetPtRange(0.2, 5.);
1640 5 // specify eta range
1641 6 cutsRP->SetEtaRange(-0.8, 0.8);
1642 7 // specify track type
1643 8 cutsRP->SetParamType(AliFlowTrackCuts::kAODFilterBit);
1644 9 // enable saving qa histograms
1645 10 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:

```

1650 1 //create the track cuts object using a static function of AliFlowTrackCuts
1651 2 AliFlowTrackCuts* cutsPOI = AliFlowTrackCuts::GetAODTrackCutsForFilterBit(1, "pion selection");
1652 3 // specify the pt range
1653 4 cutsPOI->SetPtRange(0.2, 5.);
1654 5 // specify eta range
1655 6 cutsPOI->SetEtaRange(-0.8, 0.8);
1656 7 // specify the track type
1657 8 cutsRP->SetParamType(AliFlowTrackCuts::kAODFilterBit);
1658 9 // enable saving qa histograms
1659 10 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.

```

1664 1 // which particle do we want to identify ?
1665 2 AliPID::EParticleType particleType=AliPID::kPion;
1666 3 // specify the pid method that we want to use
1667 4 AliFlowTrackCuts::PIDsource sourcePID=AliFlowTrackCuts::kTOFbayesian;
1668 5 // define the probability (between 0 and 1)
1669 6 Double_t probability = .9;
1670 7 // pass these variables to the track cut object
1671 8 cutsPOI->SetPID(particleType, sourcePID, probability);
1672 9 // the bayesian pid routine uses priors tuned to an average centrality
1673 10 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,

```

1676 1 // connect the RP's to the flow event task
1677 2 taskFE->SetCutsRP(cutsRP);
1678 3 // connect the POI's to the flow event task
1679 4 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.

```

1687 1 // get the default name of the output file ("AnalysisResults.root")
1688 2 TString file = GetCommonFileName();
1689 3 // get the common input container from the analysis manager
1690 4 AliAnalysisDataContainer *cinput = mgr->GetCommonInputContainer();
1691 5 // create a data container for the output of the flow event task
1692 6 // the output of the task is the AliFlowEventSimple class which will
1693 7 // be passed to the flow analysis tasks. note that we use a kExchangeContainer here,
1694 8 // which exchanges data between classes of the analysis chain, but is not
1695 9 // written to the output file
1696 10 AliAnalysisDataContainer *coutputFE = mgr->CreateContainer(
1697 11     "FlowEventContainer",
1698 12     AliFlowEventSimple::Class(),
1699 13     AliAnalysisManager::kExchangeContainer);
1700 14 // connect the input data to the flow event task
1701 15 mgr->ConnectInput(taskFE,0, cinput);
1702 16 // and connect the output to the flow event task
1703 17 mgr->ConnectOutput(taskFE,1, coutputFE);
1704 18 // create an additional container for the QA output of the flow event task
1705 19 // the QA histograms will be stored in a sub-folder of the output file called 'QA'
1706 20 TString taskFEQName = file;
1707 21 taskFEQName += ":QA";
1708 22 AliAnalysisDataContainer* coutputFEQA = mgr->CreateContainer(
1709 23     "FlowEventContainerQA",
1710 24     TList::Class(),
1711 25     AliAnalysisManager::kOutputContainer,
1712 26     taskFEQName.Data());

```

```

1714         );
1715 // and connect the qa output container to the flow event.
1716 // this container will be written to the output file
1717 mgr->ConnectOutput(taskFE,2,coutputFEQA);

```

1719 **Flow analysis tasks** Now that the flow event task is connected to input data, the flow analysis tasks can be set up:

```

1720 1 // declare necessary pointers
1721 2 AliAnalysisDataContainer *coutputQC[3];
1722 3 AliAnalysisTaskQCumulants *taskQC[3];
1723 4
1724 5 // the tasks will be created and added to the manager in a loop
1725 6 for(Int_t i = 0; i < 3; i++) {
1726 7     // create the flow analysis tasks
1727 8     taskQC[i] = new AliAnalysisTaskQCumulants(Form("TaskQCumulants_n=%i", i+2));
1728 9     // set their triggers
1729 10    taskQC[i]->SelectCollisionCandidates(AliEvent::kMB);
1730 11    // and set the correct harmonic n
1731 12    taskQC[i]->SetHarmonic(i+2);
1732 13
1733 14    // connect the task to the analysis manager
1734 15    mgr->AddTask(taskQC[i]);
1735 16
1736 17    // create and connect the output containers
1737 18    TString outputQC = file;
1738 19    // create a sub-folder in the output file for each flow analysis task's output
1739 20    outputQC += Form("QC_output_for_n=%i", i+2);
1740 21    // create the output containers
1741 22    coutputQC[i] = mgr->CreateContainer(
1742 23        outputQC.Data(),
1743 24        TList::Class(),
1744 25        AliAnalysisManager::kOutputContainer,
1745 26        outputQC);
1746 27    // connect the output of the flow event task to the flow analysis task
1747 28    mgr->ConnectInput(taskQC[i], 0, coutputFE);
1748 29    // and connect the output of the flow analysis task to the output container
1749 30    // which will be written to the output file
1750 31    mgr->ConnectOutput(taskQC[i], 1, coutputQC[i]);
1751 32 }
1752
1753

```

1754 **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:

```

1756 1 // check if we can initialize the manager
1757 2 if(!mgr->InitAnalysis()) return;
1758 3 // print the status of the manager to screen
1759 4 mgr->PrintStatus();
1760 5 // print to screen how the analysis is progressing
1761 6 mgr->SetUseProgressBar(1, 25);
1762 7 // start the analysis locally, reading the events from the tchain
1763 8 mgr->StartAnalysis("local", chain);
1764

```

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:

```
1 // Description: Maker to analyze Flow from the generated MC reaction plane.  
2 //           This class is used to get the real value of the flow  
3 //           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:

```

1804 1 cp $ALICE_ROOT/PWGCF/FLOW/macros/runFlowAnalysisOnTheFly.C
1805 2 aliroot runFlowAnalysisOnTheFly.C
1809

```

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:

```

1816 1 Bool_t MCEP = kTRUE; // Monte Carlo Event Plane
1817 2 Bool_t SP = kTRUE; // Scalar Product (a.k.a 'flow analysis with eta gaps')
1818 3 Bool_t GFC = kTRUE; // Generating Function Cumulants
1819 4 Bool_t QC = kTRUE; // Q-cumulants
1820 5 Bool_t FQD = kTRUE; // Fitted q-distribution
1821 6 Bool_t LYZ1SUM = kTRUE; // Lee-Yang Zero (sum generating function), first pass over the data
1822 7 Bool_t LYZ1PROD = kTRUE; // Lee-Yang Zero (product generating function), first pass over the data
1823 8 Bool_t LYZ2SUM = kFALSE; // Lee-Yang Zero (sum generating function), second pass over the data
1824 9 Bool_t LYZ2PROD = kFALSE; // Lee-Yang Zero (product generating function), second pass over the data
1825 10 Bool_t LYZEP = kFALSE; // Lee-Yang Zero Event Plane
1826 11 Bool_t MH = kFALSE; // Mixed Harmonics (used for strong parity violation studies)
1827 12 Bool_t NL = kFALSE; // Nested Loops (need for debugging, only for developers)
1828

```

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

```

1830 1 root [0] gSystem->Load("libPWGflowBase");
1831
1832
1833

```

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" \Rightarrow 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" \Rightarrow 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:

```

1843 1 qc->SetHarmonic(2);
1844 2 qc->SetCalculateDiffFlow(kTRUE);
1845 3 qc->SetCalculate2DDiffFlow(kFALSE); // vs (pt,eta)
1846 4 qc->SetApplyCorrectionForNUA(kFALSE);
1847 5 qc->SetFillMultipleControlHistograms(kFALSE);
1848 6 qc->SetMultiplicityWeight("combinations"); // default (other supported options are "unit" and "
1849 multiplicity")
1850 7 qc->SetCalculateCumulantsVsM(kFALSE);
1851 8 qc->SetCalculateAllCorrelationsVsM(kFALSE); // calculate all correlations in mixed harmonics "vs M"
1852 9 qc->SetnBinsMult(10000);
1853 10 qc->SetMinMult(0);
1854 11 qc->SetMaxMult(10000);
1855 12 qc->SetBookOnlyBasicCCH(kFALSE); // book only basic common control histograms
1856 13 qc->SetCalculateDiffFlowVsEta(kTRUE); // if you set kFALSE only differential flow vs pt is
1857 calculated
1858 14 qc->SetCalculateMixedHarmonics(kFALSE); // calculate all multi-partice mixed-harmonics correlators
1859

```

You can make QC output lighter by setting

```

1861 1 qc->SetBookOnlyBasicCCH(kTRUE);
1862
1863

```

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

```

1864 1 qc->SetCalculateDiffFlowVsEta(kFALSE);
1865

```

(if not interested in differential flow vs eta \Rightarrow 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

```

1 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
2 // Description: Maker to analyze Flow from the Event Plane method.
3 //              Adaptation based on Scalar Product
4 // authors: Naomi van del Kolk
5 //              Ante Bilandzic
6 // mods: Carlos Perez
7 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

The scalar product method

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

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

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

```

1 AliAnalysisTaskFlowEvent::void SetSubeventEtaRange(Double_t minA, Double_t maxA, Double_t minB,
2           Double_t maxB)
3           {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

```

1 AliFlowEventSimple::TagSubEventsInEta(Double_t etaMinA, Double_t etaMaxA, Double_t etaMinB, Double_t
2           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

```

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

```

where the following options are available

```

1 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

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

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

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

- $u = \text{full TPC} = u_j = u_i$

so that in the end the published points were obtained using

$$v_n = \sqrt{\frac{\langle u \cdot Q_A \rangle}{\sqrt{\langle Q_A \cdot Q_B \rangle}} \cdot \frac{\langle u \cdot Q_B \rangle}{\sqrt{\langle Q_A \cdot Q_B \rangle}}} \quad (4.3.1.3)$$

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 \mathbf{Q} vectors, the scalar product method is essentially reduced to the ‘classic’ event plane method. Normalization of the \mathbf{Q} vectors can be set using

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

4.4 AliFlowAnalysisWithCumulants

4.4.1 Theory

```
1 /*****
2  * Flow analysis with cumulants. In this class *
3  * cumulants are calculated by making use of the *
4  * formalism of generating functions proposed by *
5  * Ollitrault et al. *
6  * *
7  * Author: Ante Bilandzic *
8  *****/
```

4.4.2 Implementation

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

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

```
1 /*****
2  * estimating reference flow by *
3  * fitting q-distribution *
4  * *
5  * author: Ante Bilandzic *
6  * *
7  * based on the macro written *
8  * by Sergei Voloshin *
9  *****/
```

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

```

1  /*****
2  * In this class azimuthal correlators in mixed harmonics *
3  * are implemented in terms of Q-vectors. This approach *
4  * doesn't require evaluation of nested loops. This class *
5  * can be used to: *
6  * *
7  * a) Extract subdominant harmonics (like v1 and v4); *
8  * b) Study flow of two-particle resonances; *
9  * c) Study strong parity violation. *
10 * *
11 * Author: Ante Bilandzic *
12 *****/

```

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

```

1  //////////////////////////////////////
2  // Description: Maker to analyze Flow by the LeeYangZeros method
3  //              One needs to do two runs over the data;
4  //              First to calculate the integrated flow
5  //              and in the second to calculate the differential flow
6  // Author: Naomi van der Kolk
7  //////////////////////////////////////

```

4.8.2 Implementation

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

4.9 AliFlowAnalysisWithLYZEventPlane

4.9.1 Theory

```

1  // AliFlowAnalysisWithLYZEventPlane:
2  // Class to do flow analysis with the event plane
3  // 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 filter task, we’ll take the example of section 3.4.2 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:

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

All available filtering options in `AliFlowTrackSimpleCuts` are:

```
1 //setters
2 void SetPtMax(Double_t max) {this->fPtMax = max; fCutPt=kTRUE; }
3 void SetPtMin(Double_t min) {this->fPtMin = min; fCutPt=kTRUE; }
4 void SetEtaMax(Double_t max) {this->fEtaMax = max; fCutEta=kTRUE; }
5 void SetEtaMin(Double_t min) {this->fEtaMin = min; fCutEta=kTRUE; }
6 void SetEtaGap(Double_t min, Double_t max)
7     {fEtaGapMin = min, fEtaGapMax = max, fCutEtaGap = kTRUE; }
8 void SetPhiMax(Double_t max) {this->fPhiMax = max; fCutPhi=kTRUE; }
9 void SetPhiMin(Double_t min) {this->fPhiMin = min; fCutPhi=kTRUE; }
10 void SetPID(Int_t pid) {this->fPID = pid; fCutPID=kTRUE; }
11 void SetCharge(Int_t c) {this->fCharge = c; fCutCharge=kTRUE; }
12 void SetMassMax(Double_t max) {this->fMassMax = max; fCutMass=kTRUE; }
13 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:

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

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

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

If you 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:

^aA notable example of this is doing an invariant mass analysis, which will briefly be touched in the next section.


```

2074 1 void AliAnalysisTaskFilterFE::UserExec(Option_t *)
2075 2 {
2076 3 // Main loop
2077 4 fFlowEvent = dynamic_cast<AliFlowEventSimple*>(GetInputData(0)); // from TaskSE
2078 5 if (!fFlowEvent) return;
2079 6 if(fCutsRFP) fFlowEvent->TagRP(fCutsRFP);
2080 7 if(fCutsPOI) fFlowEvent->TagPOI(fCutsPOI);
2081 8 fFlowEvent->TagSubeventsInEta(fMinA, fMaxA, fMinB, fMaxB);
2082 9 PostData(1, fFlowEvent);
2083 10 }
2084
2085

```

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.

```

2088 1 // get the analysis manager
2089 2 AliAnalysisManager *mgr = AliAnalysisManager::GetAnalysisManager();
2090 3 // add the filter task to the manager (should be done before the
2091 4 // analysis task is added!)
2092 5 mgr->AddTask(filterTask);
2093 6 // create a temporary container which the filter task will pass to the
2094 7 // analysis task
2095 8 AliAnalysisDataContainer *coutputFilter = mgr->CreateContainer(
2096 9 "FilterContainer",
2097 10 AliFlowEventSimple::Class(),
2098 11 AliAnalysisManager::kExchangeContainer);
2099 12 // connect the output of the flow analysis task as input to the filter task
2100 13 mgr->ConnectInput(filterTask, 0, coutputFilter);
2101 14 // and connect the filter container as output
2102 15 mgr->ConnectOutput(filterTask, 1, coutputFilter);
2103 16 // pass the filter task output to the analysis method
2104 17 // (this is assuming you already have setup the analysis task as
2105 18 // explained in the example in section 3.4.3
2106 19 mgr->ConnectInput(taskQC[i], 0, coutputFilter);
2107

```

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-tagging 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.

$$\Lambda \longrightarrow \pi + p \quad (5.2.0.1)$$

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

1. Take all the $\pi + p$ 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 $\pi + p$ pairs

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

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

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 $\pi + p$ pairs in different invariant mass intervals. When a flow event is filled by-hand with $\pi + p$ 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 ρ -meson flow analysis (`$ALICE_ROOT/PWG/FLOW/Tasks/AliAnalysisTaskPhiFlow`) or the Λ and 0 flow task (`$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 3.3.3. 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.

2146 Chapter 6

2147 Summary

2148 After reading the documentation, you should have a general feeling of how the flow package is organized and be able to
2149 do a standard flow analysis. This however is just where the fun begins! Connect your classes, write a new method, add
2150 new routines ... 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

2162 Appendix A

2163 About this document

2164 A.1 Specifics and webpage

2165 Typeset using L^AT_EX, converted to HTML using pandoc via `pandoc -r latex -w html -S -s -m -N --toc`
2166 `--highlight-style tango --indented-code-classes numberLines --self-contained -o outfile.html`
2167 `FLOW.tex`

Appendix B

Flow analysis ‘on-the-fly’

The original ‘on-the-fly’ manual by Ante Bilandis 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

```
alroot 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> .

```

2198         runFlowAnalysisOnTheFly.C
                compareFlowResults.C
                outputLYZ1PRODanalysis.root
                outputQCanalysis.root
                outputFQDanalysis.root
                outputLYZ1SUManalysis.root
                outputSPanalysis.root
                outputGFCanalysis.root
                outputMCEPanalysis.root

```

2199 Each implemented method for flow analysis produced its own output file holding various output histograms. The final
 2200 flow results are stored in the common histogram structure implemented in the class `AliFlowCommonHistResults`.

2201 **Step 6:** To access and compare those final flow results automatically there is a dedicated macro available, so execute

```

2202         > aliroot
                root [0] .x compareFlowResults.C("")

```

2203 **Step 7:** If you want to rerun and get larger statistics modify

```

2204         Int_t nEvts=440

```

2205 in the macro `runFlowAnalysisOnTheFly.C`.

2206 **Step 8:** Have fun!

2207 In the next section we outline the steps for the Root users.

2208 B.2.2 Root users

2209 To be written at Nikhef...

2210 B.3 Making your own flow events

2211 This section is common both for AliRoot and Roor users. In this section we outline the procedure the user should
 2212 follow in order to simulate ‘on the fly’ the events with his own settings by making use of the available setters.
 2213 Those setters are implemented in the class `AliFlowEventSimpleMakerOnTheFly` and user shall use them in the macro
 2214 `runFlowAnalysisOnTheFly.C`.

2215 B.3.1 p_T spectra

2216 Transverse momentum of particles is sampled from the predefined Boltzmann distribution

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

2217 where M is the multiplicity of the event, T is “temperature” and m is the mass of the particle. By increasing the parameter
 2218 T one is increasing the number of high p_T particles and this parameter is the same for all events. On the other hand,
 2219 multiplicity M will in general vary from event to event. In the macro `runFlowAnalysisOnTheFly.C` one can modify
 2220 distribution (B.3.1.1) by using setter for “temperature” T and various setters for multiplicity M .

2221 **Example:** *If one wants to increase/decrease the number of high p_T particles, one should modify the line*

```

2222         Double_t dTemperatureOfRP = 0.44;

```

2223 *Examples of p_T spectra for two different values of T are shown in Figures B.1 and B.2.*

2224 What is shown in Figures B.1 and B.2 is only one example of the so called *common control histograms*. They are the
 2225 histograms organized in the same structure and implemented in the class `AliFlowCommonHist`. In output file of each
 2226 method one can access those histograms with `TBrowser`.

2227 When it comes to multiplicity M , one has a choice to sample it event-by-event from two different distributions before
 2228 plugging its value into Eq. (B.3.1.1) which than will be used to sample transverse momenta of M particles in that event.

2229 **Example:** *If one wants to sample multiplicity event-by-event from Gaussian distribution with mean 500 and spread 10,*
 2230 *one should have the following relevant settings*

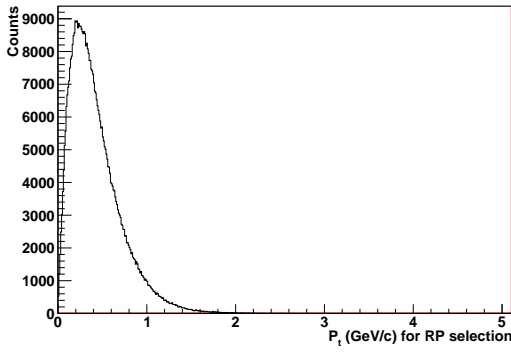
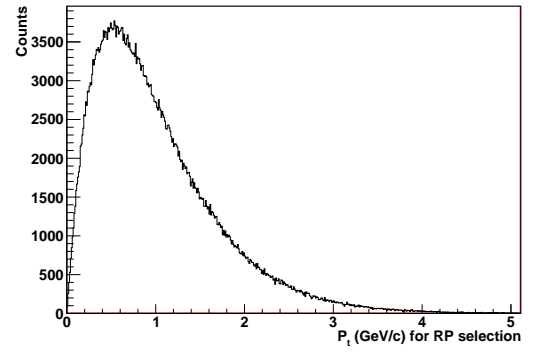
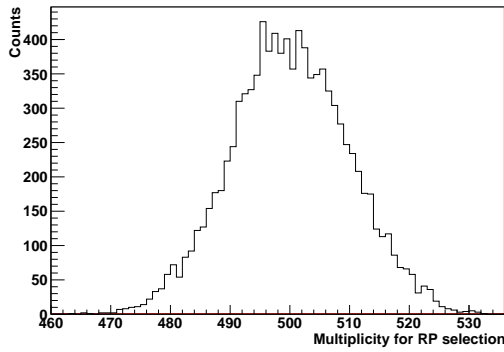
Figure B.1: $T = 0.2$ GeV/c.Figure B.2: $T = 0.5$ GeV/c.

Figure B.3: Gaussian multiplicity distribution.

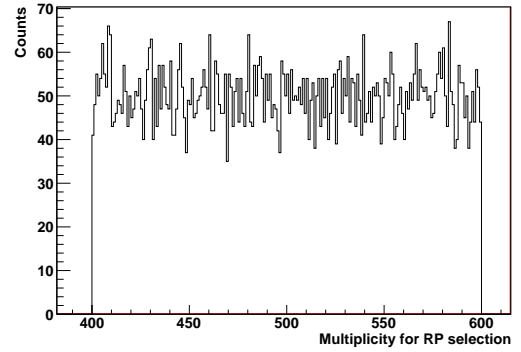


Figure B.4: Uniform multiplicity distribution.

```

2231 Bool_t bMultDistrOfRPsIsGauss = kTRUE;
      Int_t iMultiplicityOfRP = 500;
      Double_t dMultiplicitySpreadOfRP = 10;

```

2232 *Example plot for multiplicity distribution with these settings is shown in Figure B.3.*

2233 Another way to sample multiplicity event-by-event is by using uniform distribution.

2234 **Example:** *If one wants to sample multiplicity event-by-event from uniform distribution in the interval [400,600], one*
 2235 *must have the following relevant settings*

```

2236 Bool_t bMultDistrOfRPsIsGauss = kFALSE;
      Int_t iMinMultOfRP = 400;
      Int_t iMaxMultOfRP = 600;

```

2237 *Example plot for multiplicity distribution with these settings is shown in Figure B.4.*

2238 One can also fix multiplicity to be the same for each event.

2239 **Example:** *If one wants to have the same fixed multiplicity of 500 for each event one can use the following settings:*

```

2240 Bool_t bMultDistrOfRPsIsGauss = kTRUE;
      Int_t iMultiplicityOfRP = 500;
      Double_t dMultiplicitySpreadOfRP = 0;

```

2241 These are all manipulations available at the moment with p_T spectra given in Eq. (B.3.1.1).

2242 B.3.2 Azimuthal distribution

2243 If the anisotropic flow exists, it will manifest itself in the anisotropic azimuthal distribution of outgoing particles measured
 2244 with respect to the reaction plane:

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

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 $[0^\circ, 360^\circ]$. 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

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

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*

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

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;

```

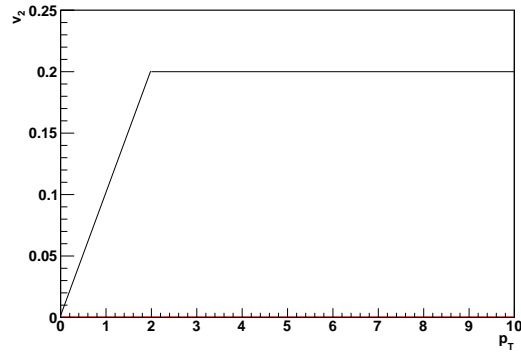


Figure B.5: p_T dependent elliptic flow.

2270 It is of course possible to simulate simultaneously nonvanishing v_1 , v_2 and v_4 .

2271 **Example:** If one wants to sample particle azimuthal angles from azimuthal distribution parameterized by harmonics
 2272 $v_1 = 2\%$, $v_2 = 5\%$ and $v_4 = 1\%$, namely

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

2273 then one should use the following settings

```
2274 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;
```

2275 In the next section we outline the procedure for simulating flow events with p_T dependent flow harmonics.

2276 p_T dependent flow harmonics

2277 In this mode the functional dependence of flow harmonics on transverse momentum is treated as an event-wise quantity,
 2278 while within the particular event the flow harmonics will change from particle to particle depending on its transverse
 2279 momentum. The implemented azimuthal distribution for this case reads

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

2280 and the functional dependence $v_2(p_T)$ is implemented as follows:

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

2281 In the macro `runFlowAnalysisOnTheFly.C` one can have the handle on the parameters v_{max} and p_{cutoff} .

2282 **Example:** If one wants to set $v_{\text{max}} = 0.2$ and $p_{\text{cutoff}} = 2 \text{ GeV}/c$, than one should use the following settings:

```
2283 Bool_t bConstantHarmonics = kFALSE;
Double_t dV2RPMax = 0.20;
Double_t dPtCutOff = 2.0;
```

2284 Example plot is given in Figure B.5.

2285 (Remark: Add further explanation here.)

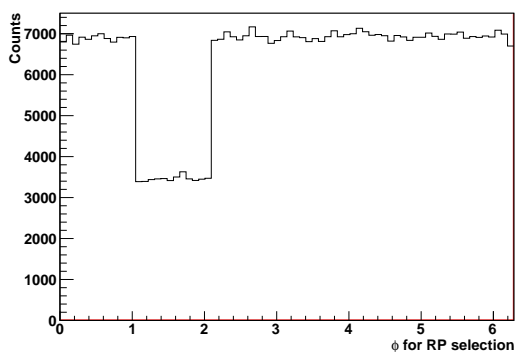


Figure B.6: Non-uniform acceptance.

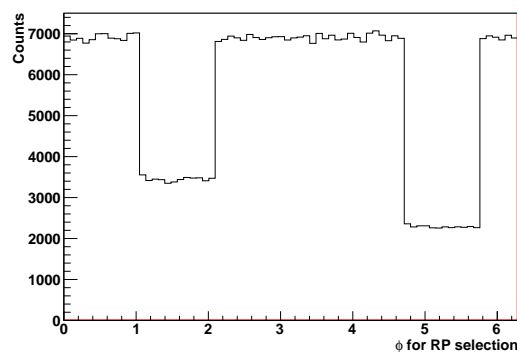


Figure B.7: Non-uniform acceptance.

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

- 2304 AddTask macro, 26
- 2305 afterburner, 21
- 2306 AliAnalysisManager, 26
- 2307 AliAnalysisTaskFilterFE, 37
- 2308 AliAnalysisTaskFlowEvent, 8
- 2309 AliAnalysisTaskFlowEvent::UserExec(), 21
- 2310 ALICE flow package, *see* flow package
- 2311 AliFlowAnalysisWithCumulants, 34
- 2312 AliFlowAnalysisWithFittingQDistribution, 34
- 2313 AliFlowAnalysisWithLeeYangZeros, 35
- 2314 AliFlowAnalysisWithLYZEventPlane, 35
- 2315 AliFlowAnalysisWithMCEventPlane, 31
- 2316 AliFlowAnalysisWithMixedHarmonics, 34
- 2317 AliFlowAnalysisWithMultiparticleCorrelations, 35
- 2318 AliFlowAnalysisWithQCumulants, 31
- 2319 AliFlowAnalysisWithScalarProduct, 33
- 2320 AliFlowCommonConstants, 19
- 2321 AliFlowCommonHist, 5
 - 2322 details, 20
- 2323 AliFlowCommonHistResults, 5
 - 2324 details, 20
- 2325 AliFlowEvent, 7
- 2326 AliFlowEvent::Fill(), 22
- 2327 AliFlowEventCuts, 9
- 2328 AliFlowEventSimple, 7
- 2329 AliFlowTrackCuts, 7, 11
- 2330 AliFlowTrackSimple, 7
- 2331 AliFlowTrackSimpleCuts, 37
- 2332 AliROOT, 1
- 2333 AliVEventHandler, 26
 - 2334 AliAODInputHandler, 26
- 2335 analysis framework, 26
- 2336 analysis manager, 26
- 2337 analysis train, 24
- 2338 AnalysisResults.root, 5

- 2339 compareFlowResults, 6
- 2340 connecting containers, 28

- 2341 event selection, 8
 - 2342 caveats, 10
 - 2343 data types, 10
 - 2344 event cuts, 9
 - 2345 parameters, 9
 - 2346 setters, 9
 - 2347 trigger selection, 8
- 2348 example, 26
 - 2349 AliAnalysisTaskFlowEvent, 27
 - 2350 connecting containers, 28
 - 2351 event selection, 27
 - 2352 launch analysis, 29
 - 2353 track selection, 27
 - 2354 trigger selection, 27

- 2355 ExchangeContainer, 28

- 2356 filterbit, 12
- 2357 Finish(), 25
- 2358 flow analysis method, 7
- 2359 flow analysis methods, 31
- 2360 flow event, 7
- 2361 flow package, 1
- 2362 flow track, 7
- 2363 flowchart, 7

- 2364 GRID, 25

- 2365 initialize methods, 4
- 2366 input data, 7
- 2367 InputContainer, 28

- 2368 LEGO framework, 37
- 2369 libPWGflowBase, 7
- 2370 libPWGflowTasks, 7
- 2371 libraries, AliROOT, 3
- 2372 libraries, ROOT, 3

- 2373 mergedAnalysisResults, 25
- 2374 methods, 31
 - 2375 AliFlowAnalysisWithCumulants, 34
 - 2376 AliFlowAnalysisWithFittingQDistribution, 34
 - 2377 AliFlowAnalysisWithLeeYangZeros, 35
 - 2378 AliFlowAnalysisWithLYZEventPlane, 31, 35
 - 2379 AliFlowAnalysisWithMixedHarmonics, 34
 - 2380 AliFlowAnalysisWithMultiparticleCorrelations, 35
 - 2381 AliFlowAnalysisWithQCumulants, 31
 - 2382 AliFlowAnalysisWithScalarProduct, 33
- 2383 Monte Carlo input, 7

- 2384 non uniform acceptance, 39
- 2385 NUA, 39

- 2386 On the fly, 3
- 2387 output file, 5
- 2388 OutputContainer, 28

- 2389 particle identification, 12
 - 2390 caveats, 14
 - 2391 methods, 13
- 2392 particles of interest, 4
- 2393 POI, *see* particles of interest

- 2394 Q-cumulant, 7

- 2395 redoFinish.C, 25
- 2396 reference particles, 4
- 2397 RP, *see* reference particles
- 2398 run.C, 26

- 2399 [runFlowOnTheFlyExample.C](#), [3](#)

- 2400 [scalar product](#), [7](#)
- 2401 [STAR input](#), [7](#)
- 2402 [steering macro](#), [26](#)

- 2403 [TBrowser](#), [5](#)
- 2404 [TChain](#), [26](#)
- 2405 [Terminate](#), [24](#)
- 2406 [TFileMerger](#), [25](#)
- 2407 [TNamed](#), [6](#)
- 2408 [track cut object](#), [simple](#), [4](#)
- 2409 [track selection](#), [10](#)
 - 2410 [AOD filterbit](#), [12](#)
 - 2411 [AOD tracks](#), [12](#)
 - 2412 [ESD tracks](#), [11](#)
 - 2413 [parameter type](#), [11](#)
 - 2414 [particle identification](#), [12](#)
 - 2415 [VZERO](#), [15](#)
- 2416 [track weights](#), [18](#)

- 2417 [UserCreateOutputObjects](#), [24](#)
- 2418 [UserExec](#), [24](#)

- 2419 [VZERO](#), [11](#), [15](#)
 - 2420 [calibration](#), [15](#)
 - 2421 [LHC10h](#), [16](#)
 - 2422 [LHC11h](#), [16](#)
 - 2423 [caveats](#), [17](#)

- 2424 [xml](#), [24](#)