

# The FLOW Analysis Package



**a short manual**

June 30, 2014

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

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

---

# Contents

8	<b>1 Introduction</b>	<b>1</b>
9	1.1 This manual . . . . .	1
10	1.2 Disclaimer . . . . .	1
11	<b>2 A Quick Start</b>	<b>3</b>
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	<b>3 The Program</b>	<b>7</b>
16	3.1 Overview . . . . .	7
17	3.2 Analysis in the ALICE analysis framework . . . . .	7
18	3.2.1 Input data . . . . .	7
19	3.2.2 Event selection . . . . .	8
20	3.2.3 Track cuts and the track cuts object . . . . .	10
21	3.2.4 Additional options . . . . .	12
22	3.2.5 Relevant pieces of code . . . . .	21
23	3.2.6 Some words on the ALICE analysis framework . . . . .	24
24	3.2.7 Example: $\pi^\pm v_n$ . . . . .	26
25	3.3 Flow analysis in ROOT: Using TTree's and TNTuples . . . . .	29
26	3.3.1 A custom class derived from AliFlowEventSimple . . . . .	29
27	3.3.2 A realistic example: flow package analysis on STAR data . . . . .	31
28	<b>4 Methods</b>	<b>33</b>
29	4.1 AliFlowAnalysisWithMCEventPlane . . . . .	33
30	4.1.1 Theory . . . . .	33
31	4.1.2 Implementation . . . . .	33
32	4.2 AliFlowAnalysisWithQCumulants . . . . .	33
33	4.2.1 Implementation . . . . .	33
34	4.3 AliFlowAnalysisWithScalarProduct . . . . .	35
35	4.3.1 Theory . . . . .	35
36	4.4 AliFlowAnalysisWithCumulants . . . . .	36
37	4.4.1 Theory . . . . .	36
38	4.4.2 Implementation . . . . .	36
39	4.5 AliFlowAnalysisWithMixedHarmonics . . . . .	36
40	4.5.1 Theory . . . . .	36
41	4.5.2 Implementation . . . . .	36
42	4.6 AliFlowAnalysisWithFittingQDistribution . . . . .	36
43	4.6.1 Theory . . . . .	36
44	4.6.2 Implementation . . . . .	36
45	4.7 AliFlowAnalysisWithMultiparticleCorrelations . . . . .	37
46	4.7.1 Theory . . . . .	37
47	4.7.2 Implementation . . . . .	37
48	4.8 AliFlowAnalysisWithLeeYangZeros . . . . .	37
49	4.8.1 Theory . . . . .	37
50	4.8.2 Implementation . . . . .	37
51	4.9 AliFlowAnalysisWithLYZEventPlane . . . . .	37
52	4.9.1 Theory . . . . .	37
53	4.9.2 Implementation . . . . .	37
54	4.10 Developing your own task . . . . .	37

55	<b>5 More exotic uses</b>	<b>39</b>
56	5.1 Flow analysis in the LEGO framework: re-tagging your POI and RP selections . . . . .	39
57	5.1.1 Caveats . . . . .	40
58	5.2 Flow analysis of resonances . . . . .	40
59	5.3 Non-uniform acceptance correction . . . . .	41
60	5.3.1 Caveats . . . . .	41
61	<b>6 Summary</b>	<b>43</b>
62	<b>7 Bibliography</b>	<b>45</b>
63	<b>A About this document</b>	<b>47</b>
64	A.1 Specifics and webpage . . . . .	47
65	<b>B Flow analysis ‘on-the-fly’</b>	<b>49</b>
66	B.1 Introduction . . . . .	49
67	B.2 Kickstart . . . . .	49
68	B.2.1 AliRoot users . . . . .	49
69	B.2.2 Root users . . . . .	50
70	B.3 Making your own flow events . . . . .	50
71	B.3.1 $p_T$ spectra . . . . .	50
72	B.3.2 Azimuthal distribution . . . . .	51
73	B.3.3 Nonflow . . . . .	54
74	B.3.4 Detector inefficiencies . . . . .	54
75	<b>Index</b>	<b>55</b>

---

# Chapter 1

## Introduction

The ALICE flow package<sup>a</sup> contains most known flow analysis methods. The package itself consists of two parts

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

### 1.1 This manual

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

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

### 1.2 Disclaimer

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

---

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



---

# Chapter 2

## A Quick Start

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

### 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<sup>b</sup>:

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`<sup>c</sup>.

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();
```

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

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

<sup>c</sup>In 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<sup>d</sup>

```
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 sub-directories 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);
```

<sup>d</sup>The on the fly event generator is not limited to the generation of the second harmonic  $v_2$ , but to get started, this is a nice example.

Note that `AnalysisResults.root` is the default name given to analyses in AliROOT. Many macros in AliROOT will expect a file `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<sup>e</sup>

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<sup>f</sup>

2. `AliFlowCommonHistResults` is an object designed to hold the common results of the flow analysis<sup>g</sup>. 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).

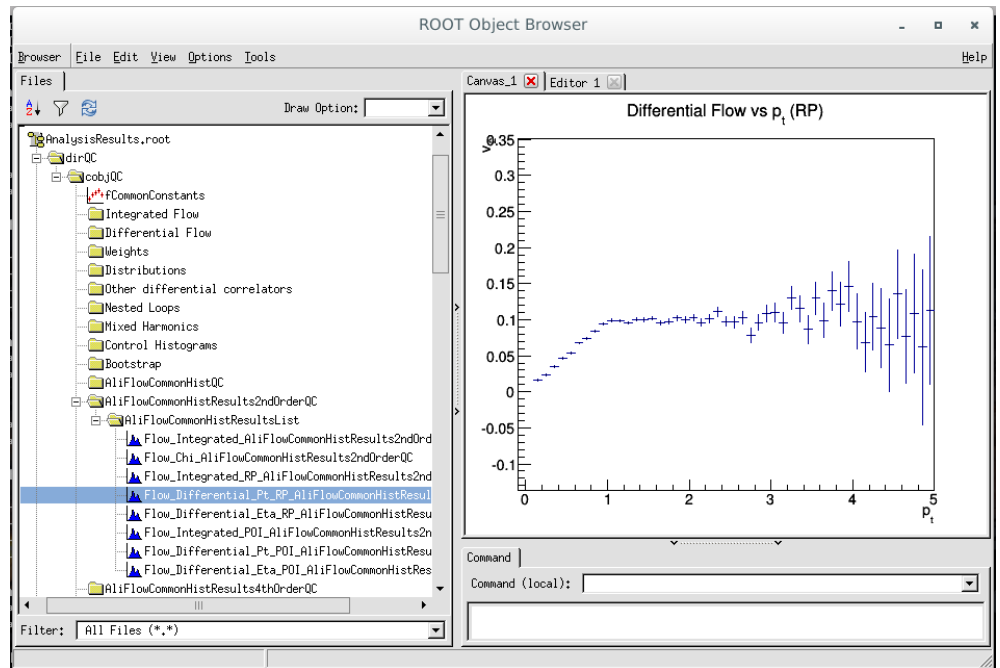
<sup>e</sup>Make sure that `libPWGflowBase.so` is loaded in your (Ali)ROOT session, otherwise these objects will be unknown.

<sup>f</sup>The headers of both output objects can be found in `$ALICE_ROOT/PWG/FLOW/Base/`.

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



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



Differential Flow vs  $p_t$  (POI)

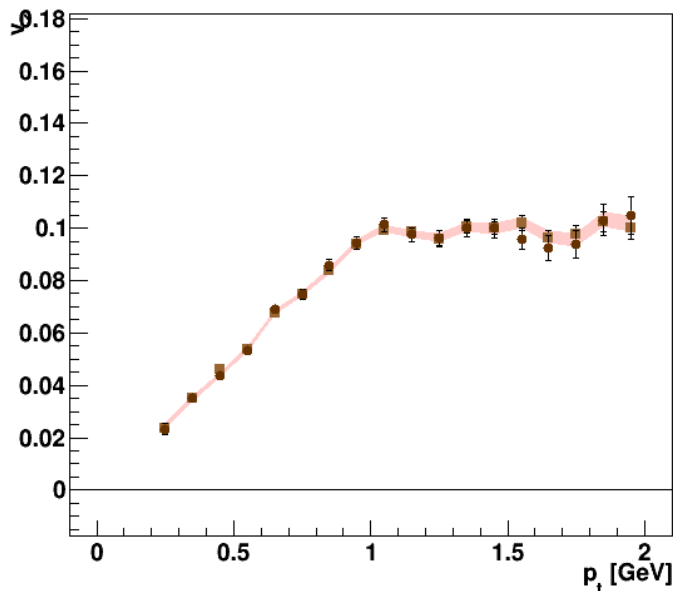


Figure 2.2: Example of inspecting the output file of the on the fly analysis with the compareFlowResults.C macro.

```
278
279
280 1 new TBrowser();
```

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

## 285 Comparing flow results

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

```
290
291
292 1 .L compareFlowResults.C
293 2 compareFlowResults(TSring("")) // the empty suffix indicates on the fly events
```

<sup>h</sup>\$ALICE\_ROOT/PWGCF/FLOW/macros/compareFlowResults.C

---

## Chapter 3

# The Program

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

### 3.1 Overview

Figure 3.1 gives a simple schematic representation of the flow package. Input events (in the case of the figure this is either ESDs or AODs) pass a set of event cuts (the common cuts) and are then converted to a flow event (stored as an `AliFlowEventSimple` object). This flow event holds a collection of flow tracks (`AliFlowTrackSimple` objects) which are passed to flow analysis methods. The only steps of this flow chart which depend on ALIROOT libraries are the ones handling ALICE data types (the ESDs or AODs). The rest of the analysis chain (the `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.

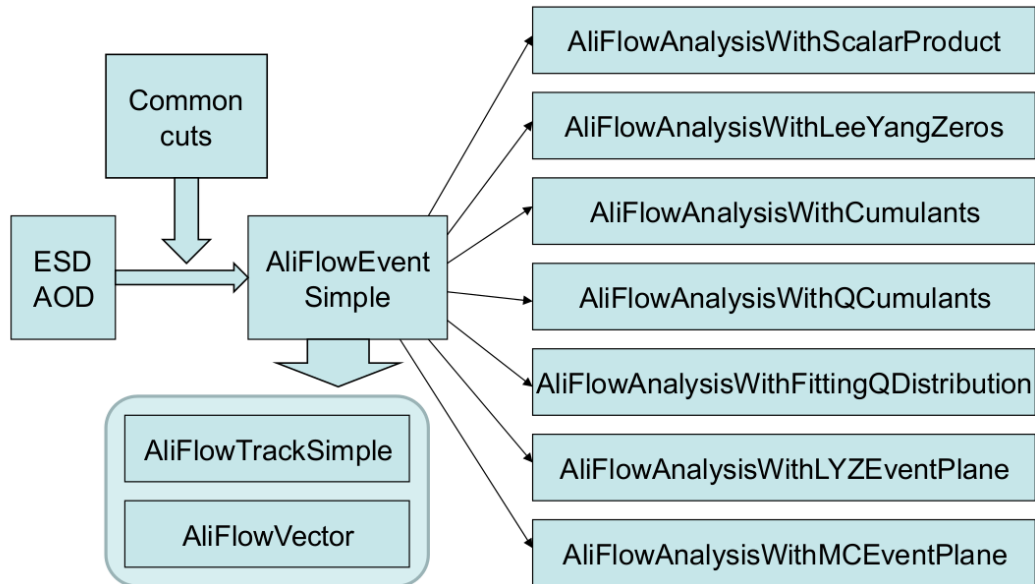
Every flow analysis in the flow package starts with the flow event. As mentioned earlier, the flow event is a simple container class which holds a collection of flow tracks, which are in turn fed to the flow analysis methods. In the next section it will be explained how the flow event can be filled with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

### 3.2 Analysis in the ALICE analysis framework

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

#### 3.2.1 Input data

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



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

335 *particles of interest* (or *POI's*), which are the particles of which you'll calculate the differential flow. The flow event and  
 336 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),  
 337 which is important to avoid auto-correlation effects which can distort the  $v_n$  measurement. The user of the flow package  
 338 however is responsible for properly setting up the analysis!

339 The flow event can be filled with input from many sources. In the second chapter of this manual, a simple method has  
 340 been shown where the flow event (the `AliFlowEventSimple` object) fills itself by generating a set of Monte Carlo tracks by  
 341 sampling kinematic variables from supplied p.d.f.'s. Using this method is a very effective tool for testing and developing  
 342 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  
 343 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  
 344 analysis, a somewhat more advanced - but not difficult - approach is necessary.

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

348 The idea behind this is the following:

- 349 1. Setup the `AliAnalysisTaskFlowEvent` task to receive input events (e.g. AODs, ESDs, MC, ...);
- 350 2. Define two sets of track selection criteria (colloquially referred to as *track cuts*), one for *POI's* and one for *RP's*;
- 351 3. Pass these two sets of track cuts to the `AliAnalysisTaskFlowEvent`;
- 352 4. The `AliAnalysisTaskFlowEvent` will convert the tracks of each input event to a set of `AliFlowSimpleTracks`.  
 353 Depending on whether or not a track passes the track selection for *POI's* or *RP's*, the `AliFlowSimpleTrack`  
 354 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  
 355 omitted from the `AliFlowSimpleTrack` collection and not added to the flow event);
- 356 5. All the `AliFlowSimpleTracks` are added to the flow event which is passed to the flow analysis methods.

### 357 3.2.2 Event selection

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

#### 362 Trigger selection

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

365 You can set a trigger by calling

```
366 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<sup>a</sup>. This function, however, is not implemented 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<sup>b</sup>.

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 SPDvertexz TRKvertexz
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,

<sup>a</sup>`$ALICE_ROOT/...`

<sup>b</sup>`$ALICE_ROOT/PWG/FLOW/Tasks/AliFlowEventCuts.cxx`

```

439
440 1 void SetNumberOfTracksMax(Int_t value) {fNumberOfTracksMax=value;fCutNumberOfTracks=kTRUE;}
441 2 void SetNumberOfTracksMin(Int_t value) {fNumberOfTracksMin=value;fCutNumberOfTracks=kTRUE;}
442 3 void SetNumberOfTracksRange(Int_t min, Int_t max) {fNumberOfTracksMin=min;fNumberOfTracksMax=max;
443   fCutNumberOfTracks=kTRUE;}
444 4 void SetRefMultMax(Int_t value) {fRefMultMax=value;fCutRefMult=kTRUE;}
445 5 void SetRefMultMin(Int_t value) {fRefMultMin=value;fCutRefMult=kTRUE;}
446 6 void SetRefMultRange(Int_t min, Int_t max) {fRefMultMin=min;fRefMultMax=max;fCutRefMult=kTRUE;}
447 7 void SetImpactParameterMax(Double_t value) {fImpactParameterMax=value;fCutImpactParameter=kTRUE;}
448 8 void SetImpactParameterMin(Double_t value) {fImpactParameterMin=value;fCutImpactParameter=kTRUE;}
449 9 void SetImpactParameterRange(Double_t min, Double_t max) {fImpactParameterMin=min;
450   fImpactParameterMax=max;fCutImpactParameter=kTRUE;}
451 10 void SetPrimaryVertexXrange(Double_t min, Double_t max)
452 11 void SetPrimaryVertexYrange(Double_t min, Double_t max)
453 12 void SetPrimaryVertexZrange(Double_t min, Double_t max)
454 13 void SetNContributorsRange(Int_t min, Int_t max=INT_MAX)
455 14 void SetMeanPtRange(Double_t min, Double_t max) {fCutMeanPt=kTRUE; fMeanPtMax=max; fMeanPtMin=min
456   };}
457 15 void SetCutSPDvertexerAnomaly(Bool_t b=kTRUE) {fCutSPDvertexerAnomaly=b;}
458 16 void SetCutZDctiming(Bool_t c=kTRUE) {fCutZDctiming=c;}
459 17 void SetCutSPDTRKVtxZ(Bool_t b=kTRUE) {fCutSPDTRKVtxZ=b;}
460 18 void SetCutTPCmultiplicityOutliers(Bool_t b=kTRUE) {fCutTPCmultiplicityOutliers=b;}
461 19 void SetCutTPCmultiplicityOutliersAOD(Bool_t b=kTRUE) {fCutTPCmultiplicityOutliersAOD=b;}
462 20 void SetRefMultMethod(refMultMethod m) {fRefMultMethod=m;}
463 21 void SetRefMultMethod(AliESDtrackCuts::MultEstTrackType m) { fRefMultMethodAliESDtrackCuts=m;
464 22 void SetRefMultCuts( AliFlowTrackCuts* cuts ) {fRefMultCuts=static_cast<AliFlowTrackCuts*>(cuts->
465   Clone());}
466 23 void SetMeanPtCuts( AliFlowTrackCuts* cuts ) {fMeanPtCuts=static_cast<AliFlowTrackCuts*>(cuts->
467   Clone());}
468 24 void SetQA(Bool_t b=kTRUE) {if (b) DefineHistograms();}
469 25 void SetCentralityPercentileMethod( refMultMethod m) {fCentralityPercentileMethod=m;}
470 26 void SetUseCentralityUnchecked(Bool_t b=kTRUE) {fUseCentralityUnchecked=b;}
471 27 void SetUsedDataset(Bool_t b=kTRUE) {fData2011=b;} // confusing name, better use different
472   interface
473 28 void SetLHC10h(Bool_t b=kTRUE) {fData2011=!b;} // TODO let cut object determine runnumber
474   and period
475 29 void SetLHC11h(Bool_t b=kTRUE) {fData2011=b;} // use this only as 'manual override'
476

```

## 477 Caveats and remarks

478 Some caveats and remarks about using the event cuts object

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

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

```

486
487 1 Bool_t AliFlowEventCuts::PassesCuts(AliVEvent *event, ALICEEvent *mcevent)
488

```

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

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

```

495
496 1 Bool_t AliFlowEventCuts::PassesCuts(AliVEvent *event, ALICEEvent *mcevent)
497

```

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

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

## 502 3.2.3 Track cuts and the track cuts object

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

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

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

## 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<sup>c</sup>. The following default track cuts sets are available:

```

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

```

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

```

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

```

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

```

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

```

The configuration falls into three categories:

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

<sup>c</sup>`$ALICE_ROOT/ANALYSIS/AliESDtrackCuts.cxx`

```

572 1  enum trackParameterType { kMC,
573 2                          kGlobal,
574 3                          kTPCstandalone,
575 4                          kSPDtracklet,
576 5                          kPMD,
577 6                          kV0,      //neutral reconstructed v0 particle
578 7                          kVZERO,  //forward VZERO detector
579 8                          kMUON,
580 9                          kKink,
581 10                         kAODFilterBit,
582 11                         kUserA,   // reserved for custom cuts
583 12                         kUserB   // reserved for custom cuts
584 13 };
585

```

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

### 589 AOD tracks as data source

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

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

```

599 1  static AliFlowTrackCuts* GetAODTrackCutsForFilterBit(UInt_t bit = 1);
600

```

602 which is defined as

```

603 1  AliFlowTrackCuts* AliFlowTrackCuts::GetAODTrackCutsForFilterBit(UInt_t bit)
604 2  {
605 3  // object which in its default form only cuts on filterbit (for AOD analysis)
606 4  AliFlowTrackCuts* cuts = new AliFlowTrackCuts(Form("AOD filterbit %i", (int)bit));
607 5  cuts->SetMinimalTPCdedx(-999999999);
608 6  cuts->SetAODfilterBit(bit);
609 7  cuts->SetParamType(AliFlowTrackCuts::kAODFilterBit);
610 8  return cuts;
611 9  }
612
613

```

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

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

### 616 3.2.4 Additional options

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

#### 621 Identified particles

622 The `AliFlowTrackCuts` object can do particle selection for a number of particles that are defined in the `AliPID`<sup>d</sup>. To  
623 enable particle identification as a selection criterion, call the function

```

624 1  void AliFlowTrackCuts::SetPID(
625 2  AliPID::EParticleType pid,
626 3  PIDsource s=kTOFPID,
627 4  Double_t prob=0.9)
628 5  {fParticleID=pid; fPIDsource=s; fParticleProbability=prob; fCutPID=kTRUE; InitPIDcuts();
629 6  }
630

```

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

<sup>d</sup>`$ALICE_ROOT/STEER/STEERBas/AliPID.h`

```

634
635 1  enum EParticleType {
636 2      kElectron = 0,
637 3      kMuon = 1,
638 4      kPion = 2,
639 5      kKaon = 3,
640 6      kProton = 4,
641 7
642 8      kDeuteron = 5,
643 9      kTriton = 6,
644 10     kHe3 = 7,
645 11     kAlpha = 8,
646 12
647 13     kPhoton = 9,
648 14     kPi0 = 10,
649 15     kNeutron = 11,
650 16     kKaon0 = 12,
651 17     kEleCon = 13,
652 18
653 19     kUnknown = 14
654 20 };
655

```

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

```

659
660 1  enum PIDsource {
661 2      kTPCpid,          // default TPC pid (via GetTPCpid)
662 3      kTOFpid,         // default TOF pid (via GetTOFpid)
663 4      kTOFbayesian,   // TOF bayesian pid (F.Noferini)
664 5      kTOFbeta,       // asymmetric cuts of TOF beta signal
665 6      kTPCdedx,       // asymmetric cuts of TPC dedx signal
666 7      kTOFbetaSimple, //simple TOF only cut
667 8      kTPCbayesian,   //bayesian cutTPC
668 9      kTPCNuclei,    // added by Natasha for Nuclei
669 10     kTPCTOFNsigma // simple cut on combined tpc tof nsigma
670 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  $\beta$  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:

```

690
691 1  gROOT->LoadMacro("ANALYSIS/macros/AddTaskPIDResponse.C");
692 2  AddTaskPIDResponse();
693

```

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

```

695
696 1  void AliFlowTrackCuts::SetNumberOfSigmas(Float_t val);
697

```



## 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.

```

1 Bool_t AliFlowTrackCuts::PassesAODpidCut(const AliAODTrack* track )
2 {
3     if(!track->GetAODEvent()->GetTOFHeader()){
4         AliAODPid *pidObj = track->GetDetPid();
5         if (!pidObj) fESDPid.GetTOFResponse().SetTimeResolution(84.);
6         else{
7             Double_t sigmaTOFPidInAOD[10];
8             pidObj->GetTOFPidResolution(sigmaTOFPidInAOD);
9             if(sigmaTOFPidInAOD[0] > 84.){
10                fESDPid.GetTOFResponse().SetTimeResolution(sigmaTOFPidInAOD[0]); // use the electron
11                TOF PID sigma as time resolution (including the TO used)
12            }
13        }
14    }
15    //check if passes the selected pid cut for ESDs
16    Bool_t pass = kTRUE;
17    switch (fPIDsource)
18    {
19        case kTOFbeta:
20            if (!PassesTOFbetaCut(track)) pass=kFALSE;
21            break;
22        case kTOFbayesian:
23            if (!PassesTOFbayesianCut(track)) pass=kFALSE;
24            break;
25        case kTPCbayesian:
26            if (!PassesTPCbayesianCut(track)) pass=kFALSE;
27            break;
28        case kTPCTOFNsigma:
29            if (!PassesTPCTOFNsigmaCut(track)) pass = kFALSE;
30            break;
31        default:
32            return kTRUE;
33        break;
34    }
35    return pass;
36 }
37 }
38 //-----
39 Bool_t AliFlowTrackCuts::PassesESDpidCut(const AliESDtrack* track )
40 {
41    //check if passes the selected pid cut for ESDs
42    Bool_t pass = kTRUE;
43    switch (fPIDsource)
44    {
45        case kTPCpid:
46            if (!PassesTPCpidCut(track)) pass=kFALSE;
47            break;
48        case kTPCdedx:
49            if (!PassesTPCdedxCut(track)) pass=kFALSE;
50            break;
51        case kTOFpid:
52            if (!PassesTOFpidCut(track)) pass=kFALSE;
53            break;
54        case kTOFbeta:
55            if (!PassesTOFbetaCut(track)) pass=kFALSE;
56            break;
57        case kTOFbetaSimple:
58            if (!PassesTOFbetaSimpleCut(track)) pass=kFALSE;
59            break;
60        case kTPCbayesian:
61            if (!PassesTPCbayesianCut(track)) pass=kFALSE;
62            break;
63        case kTOFbayesian:
64            if (!PassesTOFbayesianCut(track)) pass=kFALSE;
65            break;
66        case kTPCNuclei:
67            if (!PassesNucleiSelection(track)) pass=kFALSE;
68            break;
69        case kTPCTOFNsigma:
70            if (!PassesTPCTOFNsigmaCut(track)) pass = kFALSE;

```

```

776     71     break;
777     72     default:
778     73     printf("AliFlowTrackCuts::PassesCuts() this should never be called!\n");
779     74     pass=kFALSE;
780     75     break;
781     76     }
782     77     return pass;
783     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  $\pi v_2$ , a method which has a very high purity but low efficiency can be chosen:  $\pi$ 's are an abundant particle species and high precision requires high purity. On the other hand, if one does selection for kaons to reconstruct  $\varphi$ -mesons, loose cuts with high efficiency can be chosen, as the  $\varphi$ -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,

```

796 1 Bool_t AliFlowTrackCuts::IsSelected(TObject* obj, Int_t id)
797

```

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  $\eta$  and  $\varphi$  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

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

where  $w_i$  is a track weight,  $n$  is the harmonic, and  $\varphi$  is the azimuthal angle of a track. As explained, in the case of VZERO tiles,  $\varphi$  is derived from the position of the VZERO tile and  $w_i$  is the VZERO signal which is proportional to multiplicity. However, not all VZERO tiles are equally sensitive, and the sensitivity (can have) a run-number dependence, which results in a non-flat VZERO  $\mathbf{Q}$  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 run-numbers and the non-uniform acceptance correction corrects only for non-uniformity which is equal for all runs. Hence, the VZERO non-uniformity needs to be corrected at the time of the construction of the  $\mathbf{Q}$  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

```

824 1 // Q-vector calculation
825 2 AliFlowVector AliFlowEventSimple::GetQ(
826 3     Int_t n,           // harmonic
827 4     TList *weightsList, // weight list
828 5     Bool_t usePhiWeights, // use phi weights?
829 6     Bool_t usePtWeights, // use pt weights?
830 7     Bool_t useEtaWeights // use eta weights?
831 8 )
832 9
833 10 // Q-vectors of sub-events
834 11 void AliFlowEventSimple::Get2Qsub(
835 12     AliFlowVector* Qarray, // array with q-vectors

```

```

837 13     Int_t n,
838 14     TList *weightsList,
839 15     Bool_t usePhiWeights,
840 16     Bool_t usePtWeights,
841 17     Bool_t useEtaWeights
842 18     )
843
844 19
844 20 // overloaded implementation of Q-vectors of sub-events for VZERO information
845 21 void AliFlowEvent::Get2Qsub(
846 22     AliFlowVector* Qarray,
847 23     Int_t n,
848 24     TList *weightsList,
849 25     Bool_t usePhiWeights,
850 26     Bool_t usePtWeights,
851 27     Bool_t useEtaWeights
852 28     )

```

854 These functions are called by the flow analysis tasks and generally not by the user directly, but it is good to know where  
855 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  
856 function is designed for building a  $\mathbf{Q}$  vector from VZERO information, applying a calibration step to the VZERO signal.  
857 To make life complicated, the calibration of the VZERO  $\mathbf{Q}$  vector in LHC10h is not the same as the calibration of the  
858 VZERO  $\mathbf{Q}$  vector LHC11h data. Let's start by taking a look at the LHC10h case.

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

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

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

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

$$Q_n \rightarrow \frac{Q_n - \langle Q_n \rangle}{\sigma_{Q_n}} \quad (3.2.4.3)$$

871 where brackets denote the one-run average, and  $\sigma_{Q_n}$  the standard deviation of  $Q_n$  in the sample

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

```

874 1 AliVEvent::GetVZEROEqMultiplicity(Int_t i);
875

```

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

```

880 1 TH1F*      GetHistQ()           {return fHistQ; } ;
881 2 TH1F*      GetHistAngleQ()      {return fHistAngleQ; }
882 3 TH1F*      GetHistAngleQSub0()  {return fHistAngleQSub0; }
883 4 TH1F*      GetHistAngleQSub1()  {return fHistAngleQSub1; }
884

```

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

```

891 1 gROOT->LoadMacro("$ALICE_ROOT/ANALYSIS/macros/AddTaskVZEROEPSelection.C");
892 2 AddTaskVZEROEPSelection();
893

```

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' run-number. This can be convenient when having these cuts defined in a script which is designed to run on multiple types of input data. However, one can also call the LHC10h or LHC11h specific cuts directly via dedicated functions, which are reprinted here as the comments are important

```
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 }
```

## 965 Caveats and remarks

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

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

```

969 1 void AliFlowEvent::Get2Qsub()
970

```

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  $\eta$  coverage. For the recent ALICE paper on the flow of identified particles, the scalar product method with VZERO sub-events was used, where the two VZERO detectors comprised the two sub-events. For more information on this, take a look at the description of the scalar product method in subsection 4.3.

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

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

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

```

985 1 // to either enable or disable the recentering
986 2 // (for 11h this will mean that no calibration is performed,
987 3 // for 10h it will result in only doing a re-weighting)
988 4 void SetApplyRecentering(Bool_t r)
989 5 // to enable a per-ring instead of per-disc gain equalization (=re-weighting)
990 6 // (for 11h this has no effect)
991 7 void SetVZEROGainEqualizationPerRing(Bool_t s)
992 8 // exclude vzero rings: 0 through 7 can be excluded by calling this setter multiple times
993 9 // 0 corresponds to segment ID 0 through 7, etc
994 10 // disabled vzero rings get weight 0
995 11 // with this function you can omit information from entire vzero rings
996 12 // might be useful for runs where there is a bad signal in one of the tiles
997 13 // (sometimes referred to as 'clipping')
998 14 void SetUseVZERORing(Int_t i, Bool_t u)
999
1000

```

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  $\varphi$ ,  $\eta$  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  $\varphi$  weights, "pt\_weights" for  $p_t$  weights and "eta\_weights" for  $\eta$  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

```

1014 1 // set weight list
1015 2 AliFlowAnalysisWith*::SetWeightsList(TList* const)
1016 3 // toggle phi weights on / off
1017 4 AliFlowAnalysisWith*::SetUsePhiWeights(Bool_t const)
1018 5 // toggle eta weights on / off
1019 6 AliFlowAnalysisWith*::SetUseEtaWeights(Bool_t const)
1020 7 // toggle pt weights on / off
1021 8 AliFlowAnalysisWith*::SetUsePtWeights(Bool_t const)
1022
1023

```

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’<sup>e</sup> 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  $\eta$  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; } ;
```

<sup>e</sup>A ‘getter’ in this manual will be used to describe a function of the form `Get*()` which returns a (pointer to) a member of a class and is used to interface with the class.

and in `AliFlowCommonHistResults`

```

1180
1181 1 TH1D* GetHistChi() {return fHistChi;};
1182 2 TH1D* GetHistIntFlow() {return fHistIntFlow;};
1183 3 TH1D* GetHistIntFlowRP() {return fHistIntFlowRP;};
1184 4 TH1D* GetHistDiffFlowPtRP() {return fHistDiffFlowPtRP;};
1185 5 TH1D* GetHistDiffFlowEtaRP() {return fHistDiffFlowEtaRP;};
1186 6 TH1D* GetHistIntFlowPOI() {return fHistIntFlowPOI;};
1187 7 TH1D* GetHistDiffFlowPtPOI() {return fHistDiffFlowPtPOI;};
1188 8 TH1D* GetHistDiffFlowEtaPOI() {return fHistDiffFlowEtaPOI;};
1189 9 TList* GetHistList() {return fHistList;};
1190
1191

```

## 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

$$\frac{dN}{d\varphi_0} = \frac{1}{2\pi}. \quad (3.2.4.4)$$

Adding a periodic azimuthal modulation, this is translated to

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

which can be re-written as

$$\frac{dN}{d\varphi} = \frac{dN}{d\varphi_0} \frac{d\varphi_0}{d\varphi} = \frac{1}{2\pi} \frac{d\varphi_0}{d\varphi} \quad (3.2.4.6)$$

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

$$\varphi = \varphi_0 - v_2 \sin[2(\varphi - \Psi)]. \quad (3.2.4.7)$$

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

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

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

```

1205 1 // setters for adding by hand flow values (afterburner)
1206 2
1207 3 // toggle the afterburner on / off
1208 4 void SetAfterburnerOn(Bool_t b=kTRUE) {fAfterburnerOn=b;}
1209 5 // set differential v2 via a TF1
1210 6 void SetPtDifferentialV2( TF1 *gPtV2) {fDifferentialV2 = gPtV2;}
1211 7 // set integrated flow (used when the gPtV2 = NULL)
1212 8 void SetFlow( Double_t v1, Double_t v2, Double_t v3=0.0, Double_t v4=0.0, Double_t v5=0.0)
1213 9 {fV1=v1; fV2=v2; fV3=v3; fV4=v4; fV5=v5;}
1214

```

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

```

1218 1 void SetNonFlowNumberOfTrackClones(Int_t n) {fNonFlowNumberOfTrackClones=n;}
1219

```

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

## 3.2.5 Relevant pieces of code

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

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



**AliAnalysisTaskFlowEvent::UserExec()**

This function is called for each event.

```

1231 void AliAnalysisTaskFlowEvent::UserExec(Option_t *)
1232 {
1233 // Main loop
1234 // Called for each event
1235 //delete fFlowEvent;
1236 AliMCEvent* mcEvent = MCEvent(); // from TaskSE
1237 AliESDEvent* myESD = dynamic_cast<AliESDEvent*>(InputEvent()); // from TaskSE
1238 AliAODEvent* myAOD = dynamic_cast<AliAODEvent*>(InputEvent()); // from TaskSE
1239
1240 // the rp and poi cuts will be used to fill the flow event
1241 // so they have to be defined here
1242 if (!(fCutsRP&&fCutsPOI&&fCutsEvent))
1243 {
1244 AliError("cuts not set");
1245 return;
1246 }
1247
1248 //DEFAULT - automatically takes care of everything
1249 // the flow package will determine the datatype that you are using
1250 if (fAnalysisType == "AUTOMATIC")
1251 {
1252 //check event cuts
1253 if (InputEvent() && !fCutsEvent->IsSelected(InputEvent(),MCEvent()))
1254 return;
1255
1256 //first attach all possible information to the cuts
1257 // the track cuts will make the track selection, so they
1258 // have to be supplied with the current event
1259 // the mc event is NULL unless it is retrieved by AliAnalysisTaskSE
1260 fCutsRP->SetEvent( InputEvent(), MCEvent() ); //attach event
1261 fCutsPOI->SetEvent( InputEvent(), MCEvent() );
1262
1263 //then make the event
1264 // this function will fill the flow event with selected poi's and rp's
1265 // the implementation is printed below
1266 fFlowEvent->Fill( fCutsRP, fCutsPOI );
1267
1268 // pass some event info to the flow event
1269 fFlowEvent->SetReferenceMultiplicity(fCutsEvent->GetReferenceMultiplicity(InputEvent(),mcEvent));
1270 ;
1271 fFlowEvent->SetCentrality(fCutsEvent->GetCentrality(InputEvent(),mcEvent));
1272 if (mcEvent && mcEvent->GenEventHeader()) fFlowEvent->SetMCReactionPlaneAngle(mcEvent);
1273 }
1274
1275 // a lot of code is omitted here //
1276
1277 ////////////////////////////////////////
1278 ////////////////////////////////////////AFTERBURNER
1279 if (fAfterburnerOn)
1280 {
1281 //if reaction plane not set from elsewhere randomize it before adding flow
1282 if (!fFlowEvent->IsSetMCReactionPlaneAngle())
1283 fFlowEvent->SetMCReactionPlaneAngle(gRandom->Uniform(0.0, TMath::TwoPi()));
1284
1285 if(fDifferentialV2)
1286 fFlowEvent->AddV2(fDifferentialV2);
1287 else
1288 fFlowEvent->AddFlow(fV1,fV2,fV3,fV4,fV5); //add flow
1289 fFlowEvent->CloneTracks(fNonFlowNumberOfTrackClones); //add nonflow by cloning tracks
1290 }
1291 ////////////////////////////////////////
1292
1293 //tag subEvents
1294 // some flow analysis methods (such as the scalar product)
1295 // use sub-events. by calling this function, all tracks in the
1296 // flow event are tagged as belonging to either sub-event a or b
1297 fFlowEvent->TagSubeventsInEta(fMinA,fMaxA,fMinB,fMaxB);
1300

```

**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 ...).

```

1307 1 //-----
1308 2 void AliFlowEvent::Fill( AliFlowTrackCuts* rpCuts,
1309 3                          AliFlowTrackCuts* poiCuts )
1310 3
1311 4 {
1312 5     //Fills the event from a vevent: AliESDEvent, AliAODEvent, AliMCEvent
1313 6     //the input data needs to be attached to the cuts
1314 7     //we have two cases, if we're cutting the same collection of tracks
1315 8     //(same param type) then we can have tracks that are both rp and poi
1316 9     //in the other case we want to have two exclusive sets of rps and pois
1317 10    //e.g. one tracklets, the other PMD or global - USER IS RESPONSIBLE
1318 11    //FOR MAKING SURE THEY DONT OVERLAP OR ELSE THE SAME PARTICLE WILL BE
1319 12    //TAKEN TWICE
1320 13
1321 14    // remove the previous event
1322 15    ClearFast();
1323 16    if (!rpCuts || !poiCuts) return;
1324 17    // check the source of rp's
1325 18    AliFlowTrackCuts::trackParameterType sourceRP = rpCuts->GetParamType();
1326 19    // and ditto for the poi's
1327 20    AliFlowTrackCuts::trackParameterType sourcePOI = poiCuts->GetParamType();
1328 21
1329 22    AliFlowTrack* pTrack=NULL;
1330 23
1331 24    // if the source for rp's or poi's is the VZERO detector, get the calibration
1332 25    // and set the calibration parameters
1333 26    if (sourceRP == AliFlowTrackCuts::kVZERO) {
1334 27        SetVZEROCalibrationForTrackCuts(rpCuts);
1335 28        if(!rpCuts->GetApplyRecentering()) {
1336 29            // if the user does not want to recenter, switch the flag
1337 30            fApplyRecentering = -1;
1338 31        }
1339 32        // note: this flag is used in the overloaded implementation of Get2Qsub()
1340 33        // and tells the function to use as Qsub vectors the recentered Q-vectors
1341 34        // from the VZERO oadb file or from the event header
1342 35    }
1343 36    if (sourcePOI == AliFlowTrackCuts::kVZERO) {
1344 37        // probably no-one will choose vzero tracks as poi's ...
1345 38        SetVZEROCalibrationForTrackCuts(poiCuts);
1346 39    }
1347 40
1348 41
1349 42    if (sourceRP==sourcePOI)
1350 43    {
1351 44        //loop over tracks
1352 45        Int_t numberOfInputObjects = rpCuts->GetNumberOfInputObjects();
1353 46        for (Int_t i=0; i<numberOfInputObjects; i++)
1354 47        {
1355 48            //get input object (particle)
1356 49            TObject* particle = rpCuts->GetInputObject(i);
1357 50
1358 51            Bool_t rp = rpCuts->IsSelected(particle,i);
1359 52            Bool_t poi = poiCuts->IsSelected(particle,i);
1360 53
1361 54            if (!(rp||poi)) continue;
1362 55
1363 56            //make new AliFlowTrack
1364 57            if (rp)
1365 58            {
1366 59                pTrack = rpCuts->FillFlowTrack(fTrackCollection, fNumberOfTracks);
1367 60                if (!pTrack) continue;
1368 61                pTrack->Tag(0); IncrementNumberOfPOIs(0);
1369 62                if (poi) {pTrack->Tag(1); IncrementNumberOfPOIs(1);}
1370 63                if (pTrack->GetNDaughters()>0) fMothersCollection->Add(pTrack);
1371 64            }
1372 65            else if (poi)
1373 66            {
1374 67                pTrack = poiCuts->FillFlowTrack(fTrackCollection, fNumberOfTracks);
1375 68                if (!pTrack) continue;
1376 69                pTrack->Tag(1); IncrementNumberOfPOIs(1);
1377 70                if (pTrack->GetNDaughters()>0) fMothersCollection->Add(pTrack);
1378 71            }
1379 72            fNumberOfTracks++;
1380 73        } //end of while (i < numberOfTracks)
1381 74    }
1382 75    else if (sourceRP!=sourcePOI)
1383 76    {
1384 77        //here we have two different sources of particles, so we fill
1385 78        //them independently
1386 79        //POI

```

```

1387 80   for (Int_t i=0; i<poiCuts->GetNumberOfInputObjects(); i++)
1388 81   {
1389 82       TObject* particle = poiCuts->GetInputObject(i);
1390 83       Bool_t poi = poiCuts->IsSelected(particle,i);
1391 84       if (!poi) continue;
1392 85       pTrack = poiCuts->FillFlowTrack(fTrackCollection ,fNumberOfTracks);
1393 86       if (!pTrack) continue;
1394 87       pTrack->Tag(1);
1395 88       IncrementNumberOfPOIs(1);
1396 89       fNumberOfTracks++;
1397 90       if (pTrack->GetNDaughters()>0) fMothersCollection->Add(pTrack);
1398 91   }
1399 92   //RP
1400 93   Int_t numberOfInputObjects = rpCuts->GetNumberOfInputObjects();
1401 94   for (Int_t i=0; i<numberOfInputObjects; i++)
1402 95   {
1403 96       TObject* particle = rpCuts->GetInputObject(i);
1404 97       Bool_t rp = rpCuts->IsSelected(particle,i);
1405 98       if (!rp) continue;
1406 99       pTrack = rpCuts->FillFlowTrack(fTrackCollection ,fNumberOfTracks);
1407 100      if (!pTrack) continue;
1408 101      pTrack->Tag(0);
1409 102      IncrementNumberOfPOIs(0);
1410 103      fNumberOfTracks++;
1411 104      if (pTrack->GetNDaughters()>0) fMothersCollection->Add(pTrack);
1412 105  }
1413 106  }
1414 107  }

```

### 3.2.6 Some words on the ALICE analysis framework

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

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

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

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

#### AliAnalysisTaskSE

All analysis tasks that are called by the analysis manager have to be derived from a common class, the `AliAnalysisTaskSEf` (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;

<sup>f</sup>This section is very brief an incomplete, but keep in mind that this is a flow package manual, and not an AliROOT tutorial.

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

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

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

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

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

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

### Analysys on grid: redoFinish.C

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

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

The **redoFinish.C**<sup>§</sup> 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!

<sup>§</sup>`$ALICE_ROOT/PWCGF/FLOW/macros/redoFinish.C`

### 3.2.7 Example: $\pi^\pm v_n$

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

Generally, doing an analysis in the AliROOT is a ‘two-file process’, where one runs a run.C script in AliROOT (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/AOD086/0003/AliAOD.root");
7 chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/AOD086/0003/AliAOD.root");
8 chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/AOD086/0004/AliAOD.root");
9 chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/AOD086/0005/AliAOD.root");
10 chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/AOD086/0006/AliAOD.root");
11 chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/AOD086/0007/AliAOD.root");
12 chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/AOD086/0008/AliAOD.root");
13 chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/AOD086/0009/AliAOD.root");
14 chain->Add("/home/rbertens/Documents/CERN/ALICE_DATA/data/2010/LHC10h/000139510/ESDs/pass2/AOD086/0010/AliAOD.root");
15 // create an input handler
16 AliVEventHandler* inputH = new AliAODInputHandler();
17 // and connect it to the manager
18 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<sup>h</sup>:

```

1576 // the manager is static, so get the existing manager via the static method
1577 AliAnalysisManager *mgr = AliAnalysisManager::GetAnalysisManager();
1578 if (!mgr) {
1579     printf("No analysis manager to connect to!\n");
1580     return NULL;
1581 }
1582 // just to see if all went well, check if the input event handler has been connected
1583 if (!mgr->GetInputEventHandler()) {
1584     printf("This task requires an input event handler!\n");
1585     return NULL;
1586 }

```

**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

```

1593 // create instance of the class. because possible qa plots are added in a second ouptut slot,
1594 // the flow analysis task must know if you want to save qa plots at the time of class
1595 // construction
1596 Bool_t doQA = kTRUE;
1597 // craete instance of the class
1598 AliAnalysisTaskFlowEvent* taskFE = new AliAnalysisTaskFlowEvent("FlowEventTask", "", doQA);
1599 // add the task to the manager
1600 mgr->AddTask(taskFE);
1601 // set the trigger selection
1602 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 assumption<sup>i</sup>.

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

```

1613 // define the event cuts object
1614 AliFlowEventCuts* cutsEvent = new AliFlowEventCuts("EventCuts");
1615 // configure some event cuts, starting with centrality
1616 cutsEvent->SetCentralityPercentileRange(20., 30.);
1617 // method used for centrality determination
1618 cutsEvent->SetCentralityPercentileMethod(AliFlowEventCuts::kV0);
1619 // vertex-z cut
1620 cutsEvent->SetPrimaryVertexZrange(-10.,10.);
1621 // enable the qa plots
1622 cutsEvent->SetQA(doQA);
1623 // explicit multiplicity outlier cut
1624 cutsEvent->SetCutTPCMultiplicityOutliersAOD(kTRUE);
1625 cutsEvent->SetLHC10h(kTRUE);
1626 // and, last but not least, pass these cuts to your flow event task
1627 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,

```

1632 //create the track cuts object using a static function of AliFlowTrackCuts
1633 AliFlowTrackCuts* cutsRP = AliFlowTrackCuts::GetAODTrackCutsForFilterBit(1, "RP cuts");
1634 // specify the pt range
1635 cutsRP->SetPtRange(0.2, 5.);

```

<sup>h</sup>In 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.

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

```

1643 5 // specify eta range
1644 6 cutsRP->SetEtaRange(-0.8, 0.8);
1645 7 // specify track type
1646 8 cutsRP->SetParamType(AliFlowTrackCuts::kAODFilterBit);
1647 9 // enable saving qa histograms
1648 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:

```

1653 1 //create the track cuts object using a static function of AliFlowTrackCuts
1654 2 AliFlowTrackCuts* cutsPOI = AliFlowTrackCuts::GetAODTrackCutsForFilterBit(1, "pion selection");
1655 3 // specify the pt range
1656 4 cutsPOI->SetPtRange(0.2, 5.);
1657 5 // specify eta range
1658 6 cutsPOI->SetEtaRange(-0.8, 0.8);
1659 7 // specify the track type
1660 8 cutsRP->SetParamType(AliFlowTrackCuts::kAODFilterBit);
1661 9 // enable saving qa histograms
1662 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.

```

1667 1 // which particle do we want to identify ?
1668 2 AliPID::EParticleType particleType=AliPID::kPion;
1669 3 // specify the pid method that we want to use
1670 4 AliFlowTrackCuts::PIDsource sourcePID=AliFlowTrackCuts::kTOFbayesian;
1671 5 // define the probability (between 0 and 1)
1672 6 Double_t probability = .9;
1673 7 // pass these variables to the track cut object
1674 8 cutsPOI->SetPID(particleType, sourcePID, probability);
1675 9 // the bayesian pid routine uses priors tuned to an average centrality
1676 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,

```

1680 1 // connect the RP's to the flow event task
1681 2 taskFE->SetCutsRP(cutsRP);
1682 3 // connect the POI's to the flow event task
1683 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.

```

1690 1 // get the default name of the output file ("AnalysisResults.root")
1691 2 TString file = GetCommonFileName();
1692 3 // get the common input container from the analysis manager
1693 4 AliAnalysisDataContainer *cinput = mgr->GetCommonInputContainer();
1694 5 // create a data container for the output of the flow event task
1695 6 // the output of the task is the AliFlowEventSimple class which will
1696 7 // be passed to the flow analysis tasks. note that we use a kExchangeContainer here,
1697 8 // which exchanges data between classes of the analysis chain, but is not
1698 9 // written to the output file
1699 10 AliAnalysisDataContainer *coutputFE = mgr->CreateContainer(
1700 11     "FlowEventContainer",
1701 12     AliFlowEventSimple::Class(),
1702 13     AliAnalysisManager::kExchangeContainer);
1703 14 // connect the input data to the flow event task
1704 15 mgr->ConnectInput(taskFE,0,cinput);
1705 16 // and connect the output to the flow event task
1706 17 mgr->ConnectOutput(taskFE,1,coutputFE);
1707 18 // create an additional container for the QA output of the flow event task
1708 19 // the QA histograms will be stored in a sub-folder of the output file called 'QA'
1709 20 TString taskFEQName = file;
1710 21 taskFEQName += ":QA";
1711 22 AliAnalysisDataContainer* coutputFEQA = mgr->CreateContainer(
1712 23     "FlowEventContainerQA",
1713 24     TList::Class(),
1714 25     AliAnalysisManager::kOutputContainer,
1715 26     taskFEQName.Data()
1716 27 );
1717 28 // and connect the qa output container to the flow event.

```

```

1719 // this container will be written to the output file
1720 mgr->ConnectOutput(taskFE, 2, coutputFEQA);

```

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

```

1723 1 // declare necessary pointers
1724 2 AliAnalysisDataContainer *coutputQC [3];
1725 3 AliAnalysisTaskQCCumulants *taskQC [3];
1726 4
1727 5 // the tasks will be created and added to the manager in a loop
1728 6 for(Int_t i = 0; i < 3; i++) {
1729 7     // create the flow analysis tasks
1730 8     taskQC[i] = new AliAnalysisTaskQCCumulants(Form("TaskQCCumulants_n=%i", i+2));
1731 9     // set their triggers
1732 10    taskQC[i]->SelectCollisionCandidates(AliveEvent::kMB);
1733 11    // and set the correct harmonic n
1734 12    taskQC[i]->SetHarmonic(i+2);
1735 13
1736 14    // connect the task to the analysis manager
1737 15    mgr->AddTask(taskQC[i]);
1738 16
1739 17    // create and connect the output containers
1740 18    TString outputQC = file;
1741 19    // create a sub-folder in the output file for each flow analysis task's output
1742 20    outputQC += Form(":QC_output_for_n=%i", i+2);
1743 21    // create the output containers
1744 22    coutputQC[i] = mgr->CreateContainer(
1745 23        outputQC.Data(),
1746 24        TList::Class(),
1747 25        AliAnalysisManager::kOutputContainer,
1748 26        outputQC);
1749 27    // connect the output of the flow event task to the flow analysis task
1750 28    mgr->ConnectInput(taskQC[i], 0, coutputFE);
1751 29    // and connect the output of the flow analysis task to the output container
1752 30    // which will be written to the output file
1753 31    mgr->ConnectOutput(taskQC[i], 1, coutputQC[i]);
1754 32 }

```

**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:

```

1759 1 // check if we can initialize the manager
1760 2 if(!mgr->InitAnalysis()) return;
1761 3 // print the status of the manager to screen
1762 4 mgr->PrintStatus();
1763 5 // print to screen how the analysis is progressing
1764 6 mgr->SetUseProgressBar(1, 25);
1765 7 // start the analysis locally, reading the events from the tchain
1766 8 mgr->StartAnalysis("local", chain);

```

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

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

### 3.3.1 A custom class derived from AliFlowEventSimple

In this example, an analysis on a `TTree` is performed by deriving a class from the flow event class `AliFlowEventSimple`, `MyFlowEvent`, which can read a specific input format (in this case a `branchTTree!branch` of a `TTree`) and fills the flow event from this input. Of course you can design your task in a different way, but in this section we will stick to that example. Note that the following suggestions are all written in pseudo-code, so copy-pasting it will lead to nothing ...

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



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

```

1785
1788 1 class myParticle : public TObject
1789 2 {
1790 3 public:
1791 4     myParticle(Float_t eta, Float_t phi, Float_t pt, Int_t charge) : fEta(eta), fPhi(phi), fpT(
1792     pt), fCharge(charge) { }
1793 5     ~myParticle() {}
1794 6     virtual Double_t P()           const { return fp; }
1795 7     virtual Double_t Pt()          const { return fpT; }
1796 8     virtual Double_t Phi()         const { return fPhi; }
1797 9     virtual Double_t Eta()         const { return fEta; }
1798 10    virtual Int_t Charge()          const { return fCharge; }
1799 11 private:
1800 12    Float_t      fEta;           // eta
1801 13    Float_t      fPhi;           // phi
1802 14    Float_t      fpT;           // pT
1803 15    Int_t        fCharge;        // charge
1804 16    ClassDef(myParticle, 1); // example class
1805 17 };
1806
1807

```

Note that the members of this class ( $p_t$ ,  $\eta$ ,  $\varphi$ , charge) are all the information that an AliFlowTrackSimple needs to hold.

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

```

1812
1813 1 // first, define a set of simple cuts (the kinematic cuts)
1814 2 // which will define our poi and rp selection
1815 3 AliFlowTrackSimpleCuts *cutsRP = new AliFlowTrackSimpleCuts();
1816 4 AliFlowTrackSimpleCuts *cutsPOI = new AliFlowTrackSimpleCuts();
1817 5 cutsPOI->SetPtMin(0.2);
1818 6 cutsPOI->SetPtMax(2.0);
1819 7 // get number of entries from your tree
1820 8 Int_t nEvents = myTree->GetEntries();
1821 9 // loop over all entries
1822 10 for(Int_t i = 0; i < nEvents; i++) {
1823 11     // get the track array from the tree
1824 12     TClonesArray* particleArray = 0x0;
1825 13     // get the branch address by name
1826 14     myTree->SetBranchAddress("myParticles", &particleArray);
1827 15     // switch to the tree's i-th entry
1828 16     myTree->GetEntry(i);
1829 17     // now we do some magic: with a dedicated inherited class
1830 18     // we construct a flow event from your tree
1831 19     AliFlowEventSimple* flowEvent = new MyFlowEvent(particleArray, cutsPOI, cutsRP);
1832 20     // and from here we know how to proceed: connect the flow event
1833 21     // to the flow analysis classes, and do the analysis
1834 22     qc->Make(flowEvent);
1835 23     // memory management
1836 24     delete flowEvent;
1837 25 }
1838 26 qc->Finish();
1839 27 }
1840

```

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

```

1845
1846 1 // class constructor of an example class which reads a tree,
1847 2 // selects poi's and rp's and fills a flow event.
1848 3 // this class is derived from the flow event simple class
1849 4 // and therefore can be passed to the flow analysis methods
1850 5
1851 6 // we'll feed to class with your custom particles,
1852 7 // so this include will be necessary
1853 8 #include myParticle.h
1854 9
1855 10 // this is the class constructor
1856 11 MyFlowEvent::MyFlowEvent(
1857 12     // start with the input tracks
1858 13     TClonesArray* particleArray,
1859 14     // and pass the poi and rp cuts
1860 15     const AliStarTrackCuts* cutsRP,
1861 16     const AliStarTrackCuts* cutsPOI) :
1862 17     // derived from AliFlowEventSimple, initialized to hold a certain number of

```

```

1863 18 // tracks
1864 19 AliFlowEventSimple(particleArray->GetEntries())
1865 20 {
1866 21 // the next step will be filling the flow event
1867 22 // with POI's and RP's according to our
1868 23 // POI and RP cuts
1869 24
1870 25 for (Int_t i = 0; i < particleArray->GetEntries(); i++)
1871 26 {
1872 27 // get a particle from the particle array
1873 28 const myParticle* part = static_cast<myParticle*>particleArray->At(i);
1874 29 if (!myParticle) continue;
1875 30
1876 31 // build flow track simple (for the flow event)
1877 32 AliFlowTrackSimple* flowtrack = new AliFlowTrackSimple();
1878 33 // copy the kinematic information from the star track
1879 34 flowtrack->SetPhi(part->Phi());
1880 35 flowtrack->SetEta(part->Eta());
1881 36 flowtrack->SetPt(part->Pt());
1882 37 flowtrack->SetCharge(part->Charge());
1883 38 // see if the track is a reference track
1884 39 if (cutsRP)
1885 40 {
1886 41 Bool_t pass = rpCuts->PassesCuts(flowtrack);
1887 42 flowtrack->TagRP(pass); //tag RPs
1888 43 if (pass) IncrementNumberOfPOIs(0);
1889 44 }
1890 45 // see if the track is a particle of interest
1891 46 if (poiCuts)
1892 47 {
1893 48 flowtrack->TagPOI(poiCuts->PassesCuts(flowtrack));
1894 49 }
1895 50 // add the track to the flow event
1896 51 AddTrack(flowtrack);
1897 52 }
1898 53 }

```

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

### 3.3.2 A realistic example: flow package analysis on STAR data

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

```

1915 1 // connect the class which can read and understand your tree to
1916 2 // the input data
1917 3 AliStarEventReader starReader(inputDataFiles) ;
1918 4 // loop as long as there are events
1919 5 while ( starReader.GetNextEvent() ) // Get next event
1920 6 {
1921 7 // read a star event from the tree
1922 8 AliStarEvent* starEvent = starReader.GetEvent();
1923 9 // see if the event meets event cuts (of course these are
1924 10 // specific for STAR analysis, whether or not your tree would
1925 11 // need such a cut is up to you
1926 12 if ( !starEventCuts->PassesCuts(starEvent) ) continue;
1927 13
1928 14 // this is where flow package comes into play.
1929 15 // at this moment, a star event has been read from a tree,
1930 16 // and is stored as a 'AliStarEvent'
1931 17 // in the next step, we'll create an AliFlowEventSimple from
1932 18 // this star event using the AliFlowEventStar class, which is derived
1933 19 // from the AliFlowEventSimple class.
1934 20 // as input, the AliFlowEventStar class receives the star event,
1935 21 // and a set of poi and rp cuts

```

```

1937 22     AliFlowEventSimple* flowEvent = new AliFlowEventStar(starEvent, rpCuts, poiCuts); // make a flow
1938     event from a star event (aka "the magic")
1939 23     // for the scalar product method, we need to tag subevents
1940 24     flowEvent->TagSubeventsInEta(minA, maxA, minB, maxB );
1941 25
1942 26     qc->Make(flowEvent);
1943 27     delete flowEvent;
1944 28 }

```

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

```

1948 1 // class constructor
1949 2 AliFlowEventStar::AliFlowEventStar( const AliStarEvent* starevent,
1950                                     const AliStarTrackCuts* rpCuts,
1951                                     const AliStarTrackCuts* poiCuts ):
1952 3
1953 4 // derived from AliFlowEventSimple, initialized to hold a certain number of
1954 5 // tracks
1955 6 AliFlowEventSimple(starevent->GetNumberOfTracks())
1956 7 {
1957 8 //construct the flow event from the star event information
1958 9 SetReferenceMultiplicity(starevent->GetRefMult());
1959 10 // track loop
1960 11 for (Int_t i=0; i<starevent->GetNumberOfTracks(); i++)
1961 12 {
1962 13 // get star track from the star event
1963 14 const AliStarTrack* startrack = starevent->GetTrack(i);
1964 15 if (!startrack) continue;
1965 16 // build flow track simple (for the flow event)
1966 17 AliFlowTrackSimple* flowtrack = new AliFlowTrackSimple();
1967 18 // copy the kinematic information from the star track
1968 19 flowtrack->SetPhi(startrack->GetPhi());
1969 20 flowtrack->SetEta(startrack->GetEta());
1970 21 flowtrack->SetPt(startrack->GetPt());
1971 22 flowtrack->SetCharge(startrack->GetCharge());
1972 23 // see if the track is a reference track
1973 24 if (rpCuts)
1974 25 {
1975 26     Bool_t pass = rpCuts->PassesCuts(startrack);
1976 27     flowtrack->TagRP(pass); //tag RPs
1977 28     if (pass) IncrementNumberOfPOIs(0);
1978 29 }
1979 30 // see if the track is a particle of interest
1980 31 if (poiCuts)
1981 32 {
1982 33     flowtrack->TagPOI(poiCuts->PassesCuts(startrack)); //tag POIs
1983 34 }
1984 35 // add the track to the flow event
1985 36 AddTrack(flowtrack);
1986 37 }
1987 38 }
1988 39 }

```

---

# 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:

```

2027 1 cp $ALICE_ROOT/PWGCF/FLOW/macros/runFlowAnalysisOnTheFly.C .
2028 2 aliroot runFlowAnalysisOnTheFly.C
2030

```

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:

```

2039 1 Bool_t MCEP = kTRUE; // Monte Carlo Event Plane
2040 2 Bool_t SP = kTRUE; // Scalar Product (a.k.a 'flow analysis with eta gaps')
2041 3 Bool_t GFC = kTRUE; // Generating Function Cumulants
2042 4 Bool_t QC = kTRUE; // Q-cumulants
2043 5 Bool_t FQD = kTRUE; // Fitted q-distribution
2044 6 Bool_t LYZ1SUM = kTRUE; // Lee-Yang Zero (sum generating function), first pass over the data
2045 7 Bool_t LYZ1PROD = kTRUE; // Lee-Yang Zero (product generating function), first pass over the data
2046 8 Bool_t LYZ2SUM = kFALSE; // Lee-Yang Zero (sum generating function), second pass over the data
2047 9 Bool_t LYZ2PROD = kFALSE; // Lee-Yang Zero (product generating function), second pass over the data
2048 10 Bool_t LYZEP = kFALSE; // Lee-Yang Zero Event Plane
2049 11 Bool_t MH = kFALSE; // Mixed Harmonics (used for strong parity violation studies)
2050 12 Bool_t NL = kFALSE; // Nested Loops (need for debugging, only for developers)
2051

```

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

```

2055 1 root [0] gSystem->Load("libPWGflowBase");
2056

```

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:

```

2066 1 qc->SetHarmonic(2);
2067 2 qc->SetCalculateDiffFlow(kTRUE);
2068 3 qc->SetCalculate2DDiffFlow(kFALSE); // vs (pt,eta)
2069 4 qc->SetApplyCorrectionForNUA(kFALSE);
2070 5 qc->SetFillMultipleControlHistograms(kFALSE);
2071 6 qc->SetMultiplicityWeight("combinations"); // default (other supported options are "unit" and "
2072    multiplicity")
2073 7 qc->SetCalculateCumulantsVsM(kFALSE);
2074 8 qc->SetCalculateAllCorrelationsVsM(kFALSE); // calculate all correlations in mixed harmonics "vs M"
2075 9 qc->SetnBinsMult(10000);
2076 10 qc->SetMinMult(0);
2077 11 qc->SetMaxMult(10000);
2078 12 qc->SetBookOnlyBasicCCH(kFALSE); // book only basic common control histograms
2079 13 qc->SetCalculateDiffFlowVsEta(kTRUE); // if you set kFALSE only differential flow vs pt is
2080    calculated
2081 14 qc->SetCalculateMixedHarmonics(kFALSE); // calculate all multi-partice mixed-harmonics correlators
2082

```

You can make QC output lighter by setting

```

2084 1 qc->SetBookOnlyBasicCCH(kTRUE);
2085
2086
2087

```

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

```

2089 1 qc->SetCalculateDiffFlowVsEta(kFALSE);
2090

```

(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  $\eta$  ranges.

To setup the different  $\eta$  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  $\eta$  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 *
2166 3  * cumulants are calculated by making use of the *
2167 4  * formalism of generating functions proposed by *
2168 5  * Ollitrault et al. *
2169 6  * *
2170 7  * Author: Ante Bilandzic *
2171 8  *****/
```

### 4.4.2 Implementation

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

## 4.5 AliFlowAnalysisWithMixedHarmonics

### 4.5.1 Theory

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

### 4.5.2 Implementation

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

## 4.6 AliFlowAnalysisWithFittingQDistribution

### 4.6.1 Theory

```
1 /*****
2184 2  * estimating reference flow by *
2185 3  * fitting q-distribution *
2186 4  * *
2187 5  * author: Ante Bilandzic *
2188 6  * *
2189 7  * based on the macro written *
2190 8  * by Sergei Voloshin *
2191 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 *
2201 3  * are implemented in terms of Q-vectors. This approach *
2202 4  * doesn't require evaluation of nested loops. This class *
2203 5  * can be used to: *
2204 6  * *
2205 7  * a) Extract subdominant harmonics (like v1 and v4); *
2206 8  * b) Study flow of two-particle resonances; *
2207 9  * c) Study strong parity violation. *
2208 10 * *
2209 11 * Author: Ante Bilandzic *
2210 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

```

2217 1  //////////////////////////////////////
2218 2  // Description: Maker to analyze Flow by the LeeYangZeros method
2219 3  //
2220 4  //         One needs to do two runs over the data;
2221 5  //         First to calculate the integrated flow
2222 6  //         and in the second to calculate the differential flow
2223 7  // Author: Naomi van der Kolk
2224 8  //////////////////////////////////////
2225

```

### 4.8.2 Implementation

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

```

2230 1 Bool_t LYZ1SUM = kTRUE; // Lee-Yang Zero (sum generating function), first pass over the data
2231 2 Bool_t LYZ1PROD = kTRUE; // Lee-Yang Zero (product generating function), first pass over the data
2232 3 Bool_t LYZ2SUM = kFALSE; // Lee-Yang Zero (sum generating function), second pass over the data
2233 4 Bool_t LYZ2PROD = kFALSE; // Lee-Yang Zero (product generating function), second pass over the data
2234
2235

```

## 4.9 AliFlowAnalysisWithLYZEventPlane

### 4.9.1 Theory

```

2238 1 // AliFlowAnalysisWithLYZEventPlane:
2239 2 // Class to do flow analysis with the event plane
2240 3 // from the LYZ method
2241
2242

```

### 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

```

2250 base $ALICE_ROOT/PWG/FLOW/Base/AliFlowAnalysisTemplate.cxx (h)
2251 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<sup>a</sup>. 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 3.2.7 but perform the analysis using different  $|\eta|$  windows for RP’s.

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

```
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
2     maxB)
3     {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  $\eta$  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:

<sup>a</sup>A notable example of this is doing an invariant mass analysis, which will briefly be touched in the next section.

```

2306 1 void AliAnalysisTaskFilterFE::UserExec(Option_t *)
2307 2 {
2308 3 // Main loop
2309 4 fFlowEvent = dynamic_cast<AliFlowEventSimple*>(GetInputData(0)); // from TaskSE
2310 5 if (!fFlowEvent) return;
2311 6 if(fCutsRFP) fFlowEvent->TagRP(fCutsRFP);
2312 7 if(fCutsPOI) fFlowEvent->TagPOI(fCutsPOI);
2313 8 fFlowEvent->TagSubeventsInEta(fMinA, fMaxA, fMinB, fMaxB);
2314 9 PostData(1, fFlowEvent);
2315 10 }
2316
2317

```

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.

```

2320 1 // get the analysis manager
2321 2 AliAnalysisManager *mgr = AliAnalysisManager::GetAnalysisManager();
2322 3 // add the filter task to the manager (should be done before the
2323 4 // analysis task is added!)
2324 5 mgr->AddTask(filterTask);
2325 6 // create a temporary container which the filter task will pass to the
2326 7 // analysis task
2327 8 AliAnalysisDataContainer *coutputFilter = mgr->CreateContainer(
2328 9 "FilterContainer",
2329 10 AliFlowEventSimple::Class(),
2330 11 AliAnalysisManager::kExchangeContainer);
2331 12 // connect the output of the flow analysis task as input to the filter task
2332 13 mgr->ConnectInput(filterTask, 0, coutputFilter);
2333 14 // and connect the filter container as output
2334 15 mgr->ConnectOutput(filterTask, 1, coutputFilter);
2335 16 // pass the filter task output to the analysis method
2336 17 // (this is assuming you already have setup the analysis task as
2337 18 // explained in the example in section 3.4.3
2338 19 mgr->ConnectInput(taskQC[i], 0, coutputFilter);
2339
2340

```

### 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  $\eta$ -meson flow analysis (`$ALICE_ROOT/PWG/FLOW/Tasks/AliAnalysisTaskPhiFlow`) or the  $\Lambda$  and  $K^0$  flow task (`$ALICE_ROOT/PWG/FLOW/Tasks/AliAnalysisTaskFlowStrange`).

## 2359 5.3 Non-uniform acceptance correction

2360 In practice a detector can have inefficiencies which result in a non-uniform acceptance which might bias the measured  $v_n$   
2361 signal. One way of compensating for this is using track weights (as explained in section 3.2.4. Another way of correcting  
2362 for these effects is by adjusting the Q vectors based on the assumption that the underlying Q vector distribution itself is  
2363 flat.

2364 By default all necessary information to perform such a correction is stored when running a flow analysis task. The  
2365 actual correction itself is performed when `Finish()` is called, depending whether or not the flag to perform the correction  
2366 is set to `kTRUE`.

2367 The effects of the acceptance correction can always be checked by running the `redoFinish.C` macro, by toggling the  
2368 flag

```
2369 1 Bool_t bApplyCorrectionForNUA = kFALSE; // apply correction for non-uniform acceptance  
2370
```

2372 to either false or true.

### 2373 5.3.1 Caveats

2374 The non-uniform acceptance correction is based on the assumption that the physical Q vector distribution in your event  
2375 sample is flat. This works for minimum bias events, but might not work for e.g. triggered events or for event samples  
2376 where the detector efficiency varies event-by-event. Details pertaining to the implementation can be found in the `Finish()`  
2377 methods of the various flow analysis tasks.



---

## 2378 Chapter 6

## 2379 Summary

2380 After reading the documentation, you should have a general feeling of how the flow package is organized and be able to  
2381 do a standard flow analysis. This however is just where the fun begins! Connect your classes, write a new method, add  
2382 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





---

## 2394 Appendix A

# 2395 About this document

### 2396 A.1 Specifics and webpage

2397 Typeset using L<sup>A</sup>T<sub>E</sub>X, converted to HTML using pandoc via `pandoc -r latex -w html -S -s -m`  
2398 `-N --toc --highlight-style tango --indented-code-classes numberLines --self-contained -o`  
2399 `FlowPackageManual.html FlowPackageManual.tex`



---

## Appendix B

# Flow analysis ‘on-the-fly’

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

### B.1 Introduction

Flow analysis ‘on the fly’ is a feature in the *ALICE flow package*<sup>a</sup> 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:

---

<sup>a</sup><http://alisoft.cern.ch/viewvc/trunk/PWG2/FLOW/?root=AliRoot> .

```

2430     runFlowAnalysisOnTheFly.C
        compareFlowResults.C
        outputLYZ1PRODanalysis.root
        outputQCanalysis.root
        outputFQDanalysis.root
        outputLYZ1SUManalysis.root
        outputSPanalysis.root
        outputGFCanalysis.root
        outputMCEPanalysis.root

```

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

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

```

2434     > aliroot
        root [0] .x compareFlowResults.C("")

```

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

```

2436     Int_t nEvts=440

```

in the macro `runFlowAnalysisOnTheFly.C`.

**Step 8:** Have fun!

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

## 2440 B.2.2 Root users

2441 To be written at Nikhef...

## 2442 B.3 Making your own flow events

2443 This section is common both for AliRoot and Roor users. In this section we outline the procedure the user should  
 2444 follow in order to simulate ‘on the fly’ the events with his own settings by making use of the available setters.  
 2445 Those setters are implemented in the class `AliFlowEventSimpleMakerOnTheFly` and user shall use them in the macro  
 2446 `runFlowAnalysisOnTheFly.C`.

### 2447 B.3.1 $p_T$ spectra

2448 Transverse momentum of particles is sampled from the predefined Boltzmann distribution

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

2449 where  $M$  is the multiplicity of the event,  $T$  is “temperature” and  $m$  is the mass of the particle. By increasing the parameter  
 2450  $T$  one is increasing the number of high  $p_T$  particles and this parameter is the same for all events. On the other hand,  
 2451 multiplicity  $M$  will in general vary from event to event. In the macro `runFlowAnalysisOnTheFly.C` one can modify  
 2452 distribution (B.3.1.1) by using setter for “temperature”  $T$  and various setters for multiplicity  $M$ .

2453 **Example:** *If one wants to increase/decrease the number of high  $p_T$  particles, one should modify the line*

```

2454     Double_t dTemperatureOfRP = 0.44;

```

2455 *Examples of  $p_T$  spectra for two different values of  $T$  are shown in Figures B.1 and B.2.*

2456 What is shown in Figures B.1 and B.2 is only one example of the so called *common control histograms*. They are the  
 2457 histograms organized in the same structure and implemented in the class `AliFlowCommonHist`. In output file of each  
 2458 method one can access those histograms with `TBrowser`.

2459 When it comes to multiplicity  $M$ , one has a choice to sample it event-by-event from two different distributions before  
 2460 plugging its value into Eq. (B.3.1.1) which than will be used to sample transverse momenta of  $M$  particles in that event.

2461 **Example:** *If one wants to sample multiplicity event-by-event from Gaussian distribution with mean 500 and spread 10,*  
 2462 *one should have the following relevant settings*

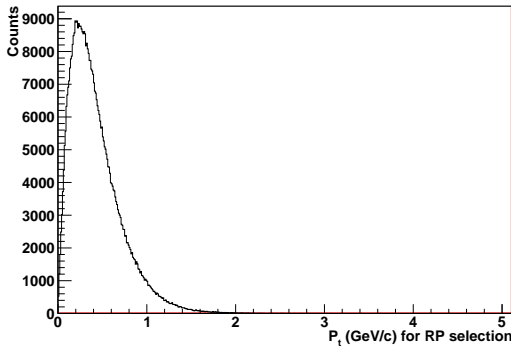
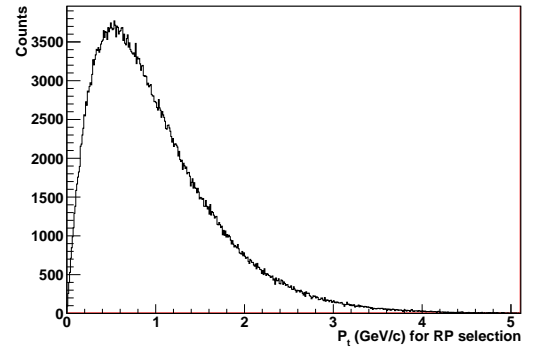
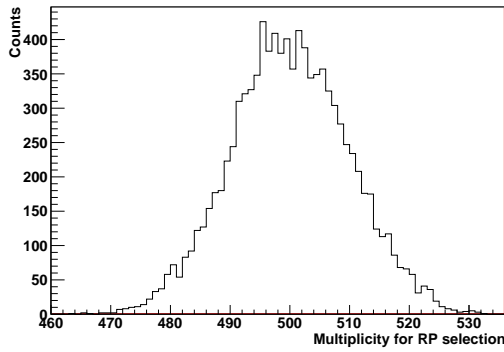
Figure B.1:  $T = 0.2$  GeV/c.Figure B.2:  $T = 0.5$  GeV/c.

Figure B.3: Gaussian multiplicity distribution.

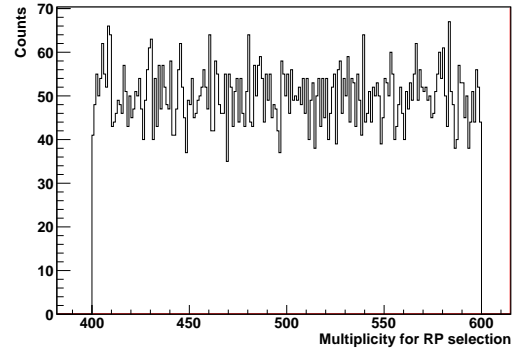


Figure B.4: Uniform multiplicity distribution.

```

2463 Bool_t bMultDistrOfRPsIsGauss = kTRUE;
      Int_t iMultiplicityOfRP = 500;
      Double_t dMultiplicitySpreadOfRP = 10;

```

2464 *Example plot for multiplicity distribution with these settings is shown in Figure B.3.*

2465 Another way to sample multiplicity event-by-event is by using uniform distribution.

2466 **Example:** *If one wants to sample multiplicity event-by-event from uniform distribution in the interval [400,600], one*  
 2467 *must have the following relevant settings*

```

2468 Bool_t bMultDistrOfRPsIsGauss = kFALSE;
      Int_t iMinMultOfRP = 400;
      Int_t iMaxMultOfRP = 600;

```

2469 *Example plot for multiplicity distribution with these settings is shown in Figure B.4.*

2470 One can also fix multiplicity to be the same for each event.

2471 **Example:** *If one wants to have the same fixed multiplicity of 500 for each event one can use the following settings:*

```

2472 Bool_t bMultDistrOfRPsIsGauss = kTRUE;
      Int_t iMultiplicityOfRP = 500;
      Double_t dMultiplicitySpreadOfRP = 0;

```

2473 These are all manipulations available at the moment with  $p_T$  spectra given in Eq. (B.3.1.1).

## 2474 B.3.2 Azimuthal distribution

2475 If the anisotropic flow exists, it will manifest itself in the anisotropic azimuthal distribution of outgoing particles measured  
 2476 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  $\eta$ . Orientation of reaction plane  $\Psi_{RP}$  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

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

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*

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

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;

```

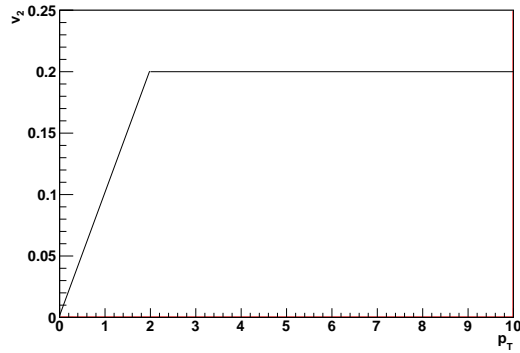


Figure B.5:  $p_T$  dependent elliptic flow.

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

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

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

then one should use the following settings

```

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

```

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

### $p_T$ dependent flow harmonics

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

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

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

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

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

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

```

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

```

Example plot is given in Figure B.5.

(Remark: Add further explanation here.)



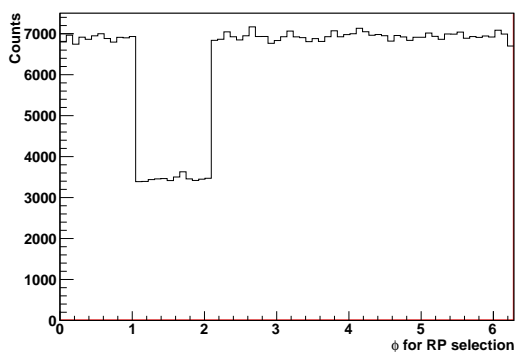


Figure B.6: Non-uniform acceptance.

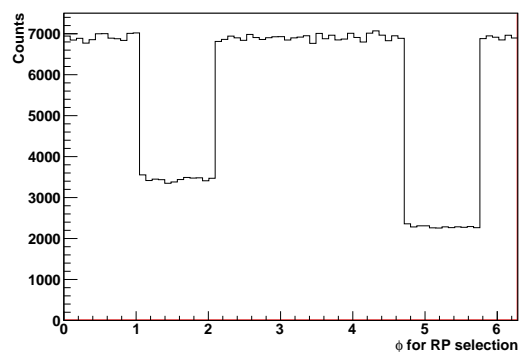


Figure B.7: Non-uniform acceptance.

### B.3.3 Nonflow

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

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

```
Int_t iLoops = 2;
```

### B.3.4 Detector inefficiencies

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

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

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

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

One can also simulate two problematic sectors.

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

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

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

---

# Index

- 2536 AddTask macro, 26
- 2537 afterburner, 21
- 2538 AliAnalysisManager, 26
- 2539 AliAnalysisTaskFilterFE, 39
- 2540 AliAnalysisTaskFlowEvent, 8
- 2541 AliAnalysisTaskFlowEvent::UserExec(), 21
- 2542 ALICE flow package, *see* flow package
- 2543 AliFlowAnalysisWithCumulants, 36
- 2544 AliFlowAnalysisWithFittingQDistribution, 36
- 2545 AliFlowAnalysisWithLeeYangZeros, 37
- 2546 AliFlowAnalysisWithLYZEventPlane, 37
- 2547 AliFlowAnalysisWithMCEventPlane, 33
- 2548 AliFlowAnalysisWithMixedHarmonics, 36
- 2549 AliFlowAnalysisWithMultiparticleCorrelations, 37
- 2550 AliFlowAnalysisWithQCumulants, 33
- 2551 AliFlowAnalysisWithScalarProduct, 35
- 2552 AliFlowCommonConstants, 19
- 2553 AliFlowCommonHist, 5
  - 2554 details, 20
- 2555 AliFlowCommonHistResults, 5
  - 2556 details, 20
- 2557 AliFlowEvent, 7
- 2558 AliFlowEvent::Fill(), 22
- 2559 AliFlowEventCuts, 9
- 2560 AliFlowEventSimple, 7, 29
- 2561 AliFlowTrackCuts, 7, 11
- 2562 AliFlowTrackSimple, 7
- 2563 AliFlowTrackSimpleCuts, 39
- 2564 AliROOT, 1
- 2565 AliStarEvent, 31
- 2566 AliStarEventReader, 31
- 2567 AliVEventHandler, 26
  - 2568 AliAODInputHandler, 26
- 2569 analysis framework, 26
- 2570 analysis manager, 26
- 2571 analysis train, 24
- 2572 AnalysisResults.root, 5
  
- 2573 compareFlowResults, 6
- 2574 connecting containers, 28
  
- 2575 event selection, 8
  - 2576 caveats, 10
  - 2577 data types, 10
  - 2578 event cuts, 9
  - 2579 parameters, 9
  - 2580 setters, 9
  - 2581 trigger selection, 8
- 2582 example, 26
  - 2583 AliAnalysisTaskFlowEvent, 27
  - 2584 connecting containers, 28
  - 2585 event selection, 27
  - 2586 launch analysis, 29
  - 2587 track selection, 27
  - 2588 trigger selection, 27
- 2589 ExchangeContainer, 28
  
- 2590 filterbit, 12
- 2591 Finish(), 25
- 2592 flow analysis method, 7
- 2593 flow analysis methods, 33
- 2594 flow event, 7
- 2595 flow package, 1
- 2596 flow track, 7
- 2597 flowchart, 7
  
- 2598 GRID, 25
  
- 2599 initialize methods, 4
- 2600 input data, 7
- 2601 InputContainer, 28
  
- 2602 LEGO framework, 39
- 2603 libPWGflowBase, 7
- 2604 libPWGflowTasks, 7
- 2605 libraries, AliROOT, 3
- 2606 libraries, ROOT, 3
  
- 2607 mergedAnalysisResults, 25
- 2608 methods, 33
  - 2609 AliFlowAnalysisWithCumulants, 36
  - 2610 AliFlowAnalysisWithFittingQDistribution, 36
  - 2611 AliFlowAnalysisWithLeeYangZeros, 37
  - 2612 AliFlowAnalysisWithLYZEventPlane, 33, 37
  - 2613 AliFlowAnalysisWithMixedHarmonics, 36
  - 2614 AliFlowAnalysisWithMultiparticleCorrelations, 37
  - 2615 AliFlowAnalysisWithQCumulants, 33
  - 2616 AliFlowAnalysisWithScalarProduct, 35
- 2617 Monte Carlo input, 7
  
- 2618 non uniform acceptance, 41
- 2619 NUA, 41
  
- 2620 On the fly, 3
- 2621 output file, 5
- 2622 OutputContainer, 28
  
- 2623 particle identification, 12
  - 2624 caveats, 14
  - 2625 methods, 13
- 2626 particles of interest, 4
- 2627 POI, *see* particles of interest
  
- 2628 Q-cumulant, 7
  
- 2629 redoFinish.C, 25
- 2630 reference particles, 4

2631 RP, *see* reference particles  
2632 run.C, [26](#)  
2633 runFlowOnTheFlyExample.C, [3](#)  
2634 runStarFlowAnalysis.C, [31](#)

2635 scalar product, [7](#)  
2636 STAR input, [7](#)  
2637 steering macro, [26](#)

2638 TBrowser, [5](#)  
2639 TChain, [26](#)  
2640 TClonesArray, [30](#)  
2641 Terminate, [24](#)  
2642 TFileMerger, [25](#)  
2643 TNamed, [6](#)  
2644 track cut object, simple, [4](#)  
2645 track selection, [10](#)  
2646     AOD filterbit, [12](#)  
2647     AOD tracks, [12](#)  
2648     ESD tracks, [11](#)  
2649     parameter type, [11](#)  
2650     particle identification, [12](#)  
2651     VZERO, [15](#)  
2652 track weights, [18](#)  
2653 TTree, [1](#)

2654 UserCreateOutputObjects, [24](#)  
2655 UserExec, [24](#)

2656 VZERO, [11](#), [15](#)  
2657     calibration, [15](#)  
2658     LHC10h, [16](#)  
2659     LHC11h, [16](#)  
2660     caveats, [17](#)

2661 xml, [24](#)