ALEPH Collaboration - CERN

ALPHA++

User's manual

ALEPH OO Physics Analysis Package

release 3.4

R. Cavanaugh, C. Delaere, G. Dissertori,
K. Huettmann, V. Lemaitre, O. van der Aa

Contents

1 Introduction

The ALEPH object oriented (OO) analysis package ALPHA++ is intended to simplify programs for physics analysis by the use of C++. As explained in the *ALPHA++* requirement list[3], the main ALPHA++ objectives are:

1. Convert the *ALEPH* data (preferably LEP2 data) from the BOS bank style into persistent objects and write them to an Objectivity/Db database

2. Rewrite a mini version of the *ALEPH* analysis package *ALPHA* in an object oriented computing language (C++), based on the Objectivity database

3. Compare standard and OO performance with regard to efficient access of the data

4. Test the software engineered by the *RD45* and the *ANAPHE/LHC++* project, which evaluate data storage and analysis option for the LHC experiments.

5. Provide some input/experience for the LEP archive project

6. Give an opportunity to learn OO analysis, programming and design

To do so, a C++ package (driver) has been implemented to loop over events just as ALPHA. The possible input files are the standard POTs, DSTs or MINIs (EPIO files), but also the Objectivity/Db database, hosting converted datasets.

Whereas the Objectivity database contains only a limited subset of the ALEPH data and Monte Carlo production and is used for test and demo purposes, all the EDIR files can be read and analysed, so that a full analysis can now be performed with ALPHA++.

The program structure is very simple (see appendix A). Three C++ methods are supplied by the user: job initialisation, event processing and job termination. In addition, a full class tree makes the analysis very easy, using objects like tracks, energy flows, jets, etc. (appendix B).

Alpha is still used internally to access data, so that the standard ALPHA cards, when meaningful, can be used. The main ALPHA physics routines are also implemented in C++ to use, modify and create the object classes.

This document describes all features of the ALPHA++ framework.

The main features are directly related to the ALPHA ones, so that we will regularly refer to the ALPHA manual. Whenever possible, the ALPHA conventions have been used, so that ALPHA++ should be natural to use for an ALPHA user. In addition, the outline of this manual is greatly inspired from the ALPHA one[1].

Finaly, the present manual should be complemented by the consultation of the ALPHA++ website: http://cern.ch/aleph-proj-alphapp . It contains all the information needed to start with ALPHA++, and all the documented code, processed with Doxygen[2].

2 Getting started

For a fast start, visit http://cern.ch/aleph-proj-alphapp/doc/start.html .

The platform supported by ALPHA++ is Linux, in particular the lxplus environment. To be able to use the ALPHA++ framework, first some environment variables have to be set[1]:

```
#-----------------------------------------
# ALPHA++ stuff (objy, root, cvs, ...)
#-----------------------------------------

# mydirectory is your work directory
# example: /afs/cern.ch/aleph/project/database/delaere
# should not mandatory contain any code
setenv MYOODIR  mydirectory
setenv PRO      /afs/cern.ch/aleph/project/database/pro
setenv DEV      /afs/cern.ch/aleph/project/database/dev
setenv PRODB    $PRO/db
setenv DEVDB    $DEV/db
setenv ALPHACARDS alpha++.cards
setenv ROOTSYS /afs/cern.ch/na49/library.4/ROOT/new/i386_redhat60/root
setenv PATH $ROOTSYS/bin:$PATH
setenv LD_LIBRARY_PATH $ROOTSYS/lib
source $DEV/user_env.csh
alias  roseinit 'source /pttools/Rose/releases/osf/config.csh'
alias  roserun  'rose'
alias  roseana  'analyzer'
alias  insureinit 'source /pttools/Insure/insure/linux/insure.csh'
setenv CVSROOT :pserver:cerncvs@pclhcb30.cern.ch:/local/alpha++
```

Some comments:

- Don't forget to set the MYOODIR field with your working directory.
- You may also have an already defined `LD_LIBRARY_PATH`.
  Test it first with `echo $LD_LIBRARY_PATH` and then use

[1] usually, it has to be placed in the user's ".tcshrc" file.

```
setenv LD_LIBRARY_PATH $ROOTSYS/lib:$LD_LIBRARY_PATH
```

- if you already use ROOT, you don't have to set the related variables.

The script sets some environment variables you need to access the database, the Anaphe/LHC++ tools, ROOT utilities and the CVS repository. The environnement script defines also standard ALPHA++ input/output files. To override the defaults, you can still redefine:

- `ALPHACARDS` : the ALPHA++ cards file
- `APPL_OUT` : the driver session output file

For example, setenv ALPHACARDS mycards.cards.

The software can be found on CVS. The scripts define the CVSROOT variable, so that you just have to checkout from the repository with:

```
"cvs co -P -r ALPHAPP-3_4 Applications"
```

The numbers indicate the version. The last version can be found on the ALPHA++ webpage. This command will create an "Applications" directory, containing all the code, and some examples. To compile an example, just type "make". The executable file will be created in the Linux directory. Execute the code by doing "Linux/acoplanar -l0" This should produce a test.root file.

To run the resulting code, you have to provide the configuration (cards) file. The default is `ALPHA++.input`. This file is similar to the ALPHA one, but with specific cards. Available cards are described in the next section.

To change the compiled code, just edit the GNUmakefile, and replace the USER entry by what is needed. (If the code you want to build is coded in test3.cpp, set "USER=test3") and run "make" again.

Three other test codes are available at startup:

- test1.cpp: print the Lorentz vector of all Energy flows.
- test2.cpp: select leptons and build jets. Some screen outputs.
- test3.cpp: look for the main vertex and compare it with the MC expectation. Compute the b-probability and fill ntuples.

To produce a PAW ntuple, just remove the .root extension in the HIST card placed in ALPHA++.input. Note that the PAW output is unavailable for acoplanar (the default compiled code), due to the use of columnwise-like commands, still not implemented for the ALPHA++ HBOOK-wrapper.

A sample-script to run a job on LSF is available on the web page.

3 Input cards

The file defined by the environment variable $ALPHACARDS contains the configuration cards used by ALPHA++. In this file are defined the events to be processed and some options. The cards available follow similar rules compared to ALPHA. Comments are preceeded by a "#" or a "!".

Availables cards are:

NEVT event selection by number. As with ALPHA, there are 2 synthaxes:

- NEVT n : read the n first events
- NEVT n1 n2 : read events from n1 to n2

Only events of the relevant class are read. There must be only one NEVT card.

CLAS select the EDIR classe(s) to read.
CLASS c1 c2 c3 : read events of class c1, c2 and c3.

DBTY database type. 3 obtions are availables:

- DBTY objy : access the federated Objectivity/Db database.
- DBTY epio : read the standard ALPHA files (POT, MINI, DST).
- DBTY epio/objy : read standard ALPHA files and open the Objectivity/Db database for HTL[4] output.

TATY transaction type: read or write. Up to now, only read transactions are implemented.

SETY session type: inter or batch. Batch mode is used for most of the analysis jobs. In interactive mode, the software will wait after each event for a user request, generaly generated by an external software. Up to now only AlVisu, the OO visualisation software is using this feature.

READ name of the file including the FILI statements for obejectivity input. The list of FILI statements included in this file is appended to the any list directly included in the file.

AFII ALPHA FILI

AFIO ALPHA FILO

AREA ALPHA READ

ASRU ALPHA SRUN

HIST the output histogram file. Depending on the file extension, the output will be produced by ROOT (*.root) or HBOOK (everything else).

SEED random generator seed

ACAR ALPHA card complement. When reading EDIR, you can specify additional ALPHA cards in a separate file. A standard ALPHA cardsfile is built using those cards, the info specified in the ALPHA++.input cardfile, and some additional cards are addede automatically:

```
EFLW
BSIZ 0.0120 0.0007
QFND
DWIN 1 0.035
NSEQ
ENDQ
```

MUID cuts for muon identification (see the section 10.5.2)

ELID cuts for electron identification (see the section 10.5.2)

TAID cuts for tau identification (see the section 10.5.2)

USER The user can define his own cards.
This card allows to define quantities that you will use in your own code (calibrations, cuts, debug, various options,...) The synthax is the following:

```
USER name value1 value2 ...
```

The first argument is the tag you assign to the card, the other are tag quantities being passed to the code. Even if any string can be used as tag, it is recommended to use 4 uppercase letters, like the other ALPHA cards. In the user code, the info will be available by

```
theCardsReader->getUserVards(),
```

returning a vector<pair<string,vector<double> > >. For each entry in the vector, the first element of the pair is the card name, and the second is the vector of values.

4 User routines

Only three routines have to be provided:

initialisation `void AlephExManager::UserInit()`

event analysis `bool AlephExManager::UserEvent(AlphaBanks& EventInfo)`

termination `void AlephExManager::UserTerm()`

Each example code provided gives an example of those three routines. In order to get the code compiled, you must include the following lines at the beginning of the code:

```
#include "AlephExManager.h"
#include "AlephCollection.h"
```

In addition, if you want to write in the output text file used by the driver, include the fout stream as follows:

```
#include <fstream.h>
extern ofstream fout
```

Other possible includes are:

- #include <vector> to directly use STL vectors
- #include "AlToolBox.h" to use the Tool Box (see the section 10.4)

If you want your own classes to be known by the driver, you have to put the declaration in UserClasses.h. UserClasses.h is included in the driver, and it should be possible to use them with ROOT (see the ROOT manual[5] for more details).

As shown figure 1, the job is handled by the driver. The user initialisation is called at the start of the job, after the standard ALPHA initialisation. There you can book histograms and/or ntuples, and perform other setup for the job. The user event analysis is called for each event The returned value is used to select intersting events during an interactive session. The user program termination is called before closing the output files and the BOS structure in memory.

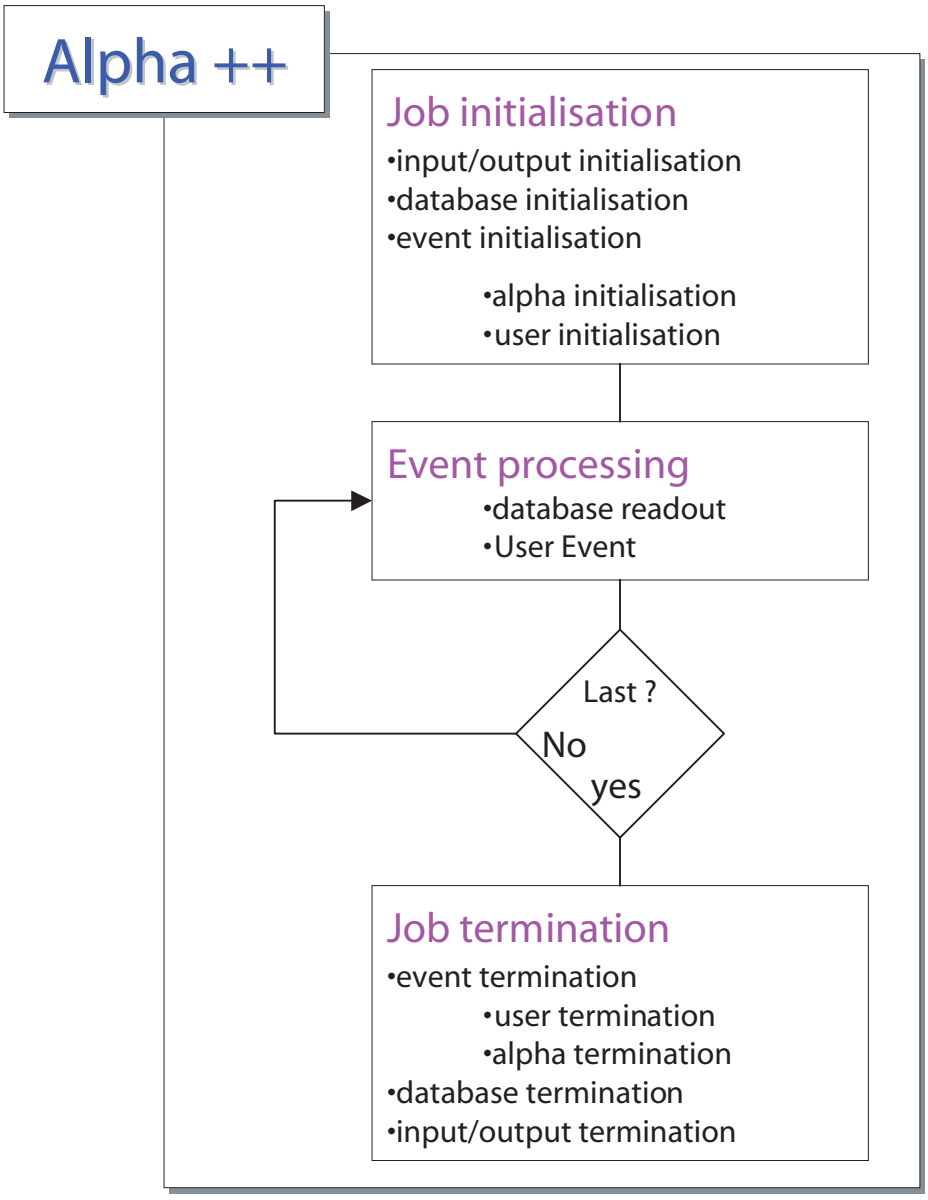

Figure 1: The ALPHA++ job processing.

5 Creating histograms and ntuples

The standard histogram packages of ALPHA++ are HBOOK and ROOT. The output file is given by the use of the HIST card in the alpha++.input cardfile, using the syntax "HIST file". If the file given is a .root file, the ROOT package is initialized and used for output; otherwise, the file is given as input of the standard ALPHA routines (so that an .exch file is created following ALPHA rules).

A default system-independent set of routines is provided to create a ntuple or a ROOT tree. Using those routines, the same code should be able to produce ROOT or PAW files, depending on the HIST card.

5.1 During the UserInit routine

The first step is the definition of the output contents. To describe the tree/ntuple, a set of routines is provided. No booking is needed anymore. To define the variables wanted you can use a mix of the following routines:

```
void TheNtupleWriter()->AddOutput(tpe obj, char* name)
void TheNtupleWriter()->AddOutput(tpe obj, unsigned int sze, ...)
void TheNtupleWriter()->AddScalableOutput(char* name, tpe obj,
                                                    unsigned int size)
```

Tpe is any C++ type.

5.1.1 Example 1: add an integer to the output list

```
int i;
TheNtupleWriter()->AddOutput(i,"var1");
```

There, var1 is the name of the variable.

Note that the object (i in the example) just tells the routines the type to be used, and can be deleted later. Another object will be used during the analysis, so that no global object needs to be defined.

5.1.2 Example 2: add several variables (of the same type) at once

```
int i;
TheNtupleWriter()->AddOutput(i,3,"var1","var2","var3");
```

The second argument is just the number of declared variables.

5.1.3 Example 3: add an array of float of variable size

```
float a;
TheNtupleWriter()->AddScalableOutput("array",a,100);
```

The third argument is here the MAXIMUM size of the array. ROOT still needs this to allocate a global memory area. Only the used part of this is written to the ROOT file.

5.1.4 Some coments

- If the class object is not a basic type, it will only work with ROOT (cfr. the ROOT documentation for more details).
- Up to now, the ScalableOutput is not implemented for HBOOK. It is intended to interface the column-wise ntuples.

5.2 In UserEvent

In the UserEvent routine the specific event is analysed and the reconstructed information is stored in the specific ntuple/tree variables.

5.2.1 Record an instant picture of a single variable

```
void TheNtupleWriter()->Keep(char* name, entry value)
```

example:

```
int myvar = 3;
TheNtupleWriter()->Keep("output",myvar);
```

5.2.2 Record an instant picture of a scalable variable (array)

```
void TheNtupleWriter()->Keep(char* name,entry* startofarray,int size)
```

example:

```
float myarray[10]={0,1,2,3,4,5,6,7,8,9};
TheNtupleWriter()->Keep("array",myarray,10);
```

There are also specific methods to store STL vectors, HepVectors, etc. In each case, the name must correspond to a Scalable Output:

```
void TheNtupleWriter()->KeepV(char* name , vector<entry> value)
void TheNtupleWriter()->Keep(char* name, HepLorentzVector& value)
void TheNtupleWriter()->Keep(char* name, Hep3Vector& value)
```

The object will be stored as a float array.

5.3 At the end of UserEvent

At the end of the routine, you just call

```
TheNtupleWriter->Fill();
```

to store everything in the histograming file, if the event is selected (e.g. after some preselection cuts).

5.4 More about the HBOOK interface

The methods presented in the previous section are intended to make the histogramming easy and independant of the output format/processing tool. Nevertheless, the capabilities of both HBOOK and ROOT are more developed, and you may want to use more specific methods. ALPHA++ is using the set of FORTRAN-wrapped HBOOK routines provided by Anaphe/LHC++ in hbook.h. Note that ALPHA++ already opens and closes this file.

5.5 More about the ROOT interface

The file defined in the HIST card is opened for ROOT by the driver. By default, ALPHA++ uses a tree called "analysis". At the end, the tree and the file are closed. Refer to the ROOT documentation to see how to place more objects in the file, or how to directly edit the analysis tree.

6 Event and Run informations

Event and run information are available via dedicated classes. To access the information, first instantiate the classes from AlphaBanks (AlphaBanks is passed by reference to the UserEvent routine. The instance will be referred to as EventInfo:):

```
AlEvent& myevent = EventInfo.Event();
AlRun&   myrun   = EventInfo.Run();
```

The use of references makes the code faster, which is crucial in the UserEvent routine (repeated N_{event} times).

The methods of these objects give access to the relevant information.

6.1 AlEvent methods

TimeInfo Time(); returns the event time.
TimeInfo is a utility class constructed from the standard ALPHA date and time strings. Information is obtained via public data members: msec, sec, min, hour, day, month, year. In addition, the comparison operators are defined.

int number() event number

int type() event type

int EdirClass() edir class pattern

int ErrorStatus() detector error status

float Energy() event Energy (QELEP)

float gen_e12() MC e12 generator result for the event. This value is generated when needed. Two subsequent calls will give differents results. If the seed is fixed, two executions of the code will give the same result.

bool IsTrigger(bool MINI) simulate the trigger word as the alpha routine "DECTRG" do. The argument must be "TRUE" when running on MINI, "FALSE" otherwise.

int GetRawTrigger() returns the raw trigger word.

6.2 AlRun methods

int energy() returns the mean run energy (as stored in the LEP header bank)

int ExperimentNumber() returns the experiment number

int number() returns the run number

int type() returns the run type

7 Handling objects

7.1 Access to the objects

Objects are available as vectors filled by the AlphaBanks class just before the user routine is called. Those transient objects are used during the analysis in replacement of the persistent objects stored in the epio files or Objectivity/Db databases.

The AlephCollection is a specific class defined for this purpose. The physics routines are methods of AlephCollection, so that they are easy to apply. The AlephCollection is deriving from the STL vector. In addition, the looperase method makes the selection easier.

To get the collections, just call the AlphaBanks methods:

```
AlephCollection<AlGamp*> mygampeccollection = EventInfo.GampPV();
AlephCollection<AlEflw*> myeflowcollection  = EventInfo.EflwPV();
AlephCollection<AlTrack*> mytrackcollection = EventInfo.TrackPV();
AlephCollection<AlTrack*> mystdv0collection = EventInfo.stdV0PV();
AlephCollection<AlTrack*> myv0trackcollection = EventInfo.V0trackPV();
AlephCollection<AlTrack*> mylongv0collection = EventInfo.longV0PV();
AlephCollection<AlMCtruth*> myMCcollection  = EventInfo.MCtruthPV();
AlephCollection<AlMuon*> mymuoncollection   = EventInfo.MuonPV();
AlephCollection<AlElec*> myeleccollection   = EventInfo.ElecPV();
AlephCollection<AlVertex*> myMCvertices   = EventInfo.MCverticesPV();
AlephCollection<AlVertex*> mysecvertices  = EventInfo.SecVerticesPV();
AlVertex* mymainvertex = EventInfo.MainVertexP();
```

Other methods, returning the original vector of objects as a reference, are available. It should be noticed that in the latter case it is possible to modify the object; a deleted track will definetly be lost. Using the collections of pointers, you can always retrieve a new full collection.

To loop on the collections, there are two possibilities:

```
for(AlephCollection<AlTrack*>::iterator itrack = mycollection.begin();
                                        itrack < mycollection.end();
                                        itrack++                    )
     {
        cout << (*itrack)->QX() << endl;
     }
```

or

```
for(unsigned int i=0;i<mycollection.size();i++)
    {
        cout << mycollection[i]->QX() << endl;
    }
```

Although the second solution is shorter in terms of typing and may seem more standard, it is in fact far more time consuming.

To erase elements in an AlephCollection, the looperase method is usefull:

```
for(AlephCollection<AlTrack*>::iterator itrack = mycollection.begin();
                                        itrack < mycollection.end();
                                        itrack++                    )
    {
       if((*idx)->QP()<1) looperase(itrack);
    }
```

Finally, two collections can be merged by "append(AlephCollection<type>)" and you can create a vector of pointers to the objects in the vector with the Pointers() method:

```
AlephCollection<AlJet> myjets;
...
AlephCollection<AlJet*> mypointers = myjets.Pointers();
```

7.2 The AlObject class

All the *ALPHA++* objects are derived from AlObject. AlObject is a pure virtual class, definig thus the general interface:

- HepLorentzVector A4V() : returns a Lorentz vector
- ALEPHTYPE TYPE() : returns the object type
- float QCH () : returns the charge
- (more)

The two classes directly deriving from AlObject are QvecBase, for all track-like objects, and QvrtBase, for all vertices.

7.3 The QvecBase class

The methods of QvecBase define some standard ALPHA routines, common to all track-like objects:

- HepLorentzVector A4V() : returns a Lorentz vector
- ALEPHTYPE TYPE() : returns the object type
- float QP(), QX(), QY(), QZ(), QE(), QM(), QCT(), QCH(), QPH(), QPT(), QBETA(), QGAMMA() The meaning of those routines is the same as for ALPHA.

Some additional methods take another QvecBase-descending object as argument. Also, the ALPHA manual should be consulted for more details.

- float QMSQ2(QvecBase j)
- QM2(QvecBase j)
- QDMSQ(QvecBase j)
- QPPAR(QvecBase j)
- QPPER(QvecBase j)
- QDOT3(QvecBase j)
- QDOT4(QvecBase j)
- QCOSA(QvecBase j)
- QDECA2(QvecBase j)
- QDECAN(QvecBase j)

Finally, ALPHA++ specific methods are the lock-related methods and the ability to sort objects.

There are 3 lock-related methods:

- Lock() will lock the object, setting a flag. A locked object will not be used by the various algorithms. The method takes as argument an optional parameter. Lock(1) will lock the object recursively. For example, locking a lepton will lock the associated track/Eflow object.
- In the same way, unlock() will unlock an object and unlock(1) will unlock recursively.
- isLocked() returns a boolean corresponding to the Lock status (true=locked).

The QvecBase-derived objects can be sorted, simply using the `operator<`. The sort criterium can be selected by defining the global variable `QvecBase::SortCriterium`. The defined values are 0,1,2,3,4, respectively for E, px, py, pz, pt.

7.4 The QvrtBase class

The QvrtBase class is very similar to the QvecBase one. The main methods are:

- A4V() returns the 4-vector. By definition, the energy of a vertex is 0.
- ALEPHTYPE TYPE() : returns the object type
- float QP(), QX(), QY(), QZ(), QE(), QM(), QCT(), QCH(), QPH(), QPT(), QBETA(), QGAMMA() The meaning of those routines is the same as for ALPHA.
- int KVN() : returns the ALPHA vertex number
- int KVNTYPE() : returns the vertex type
- double QVCHIF() : χ^2 of the vertex
- HepSymMatrix QVEM() : the covariance matrix
- float QVEM(int i) : a covariance matrix element

A vertex can of course be (un)locked. The methods are the same as for QvecBase objects: Lock(), unLock(), isLocked().

8 Tracks and vectorial objects

Here the additional methods specific to the different types of QvecBase-derived objects are described.

8.1 AlEflw: the Enery Flow object

The specific methods are:

```
EFLWTYPE GetEftype()  returns the Eflow type.
```

You can use the integer value, as explained in the ALPHA manual, or the corresponding enumerated list EFLWTYPE :
{Chargedtrack, Electron, Muon, V0Track, Electromagnetic, ECAL, HCAL, LCAL, SICAL}

```
AlTrack* getTrack() returns the associated track
```

8.2 AlGamp: the Gampec object

A first set of routines is provided in order to select the Lorentz vector to use with the "Q" routines. Possibilities are the corrected Lorentz vector from Gampeck, the standard Lorentz vector from QVEC and the raw Lorentz vector (from raw Gampeck info). Usualy, the corrected and the standard vectors are the same.

```
void UseCorrectedA4V();

void UseStandardA4V();

void UseRawA4V();
```

Another set of routines is intended to return the Lorentz vector, either from Gampeck or from QVEC.

```
HepLorentzVector GetCorrectedA4V() const;

HepLorentzVector GetStandardA4V() const;

HepLorentzVector GetRawA4V() const;
```

Then, various standard quality factors are provided:

```
//Energy fraction in stack 1 or 2
float EnergyFractionInStack(const int)

//Energy fraction in the 4 central towers
float EnergyFractionInCentralTowers()

// Distance to the closest track (cm)
float Isolation()

//Storey flag
int StoreyFlag()

// Quality flag
int QualityFlag()
```

Finally, the so called "advanced quality factors are also interfaced:

```
// Quality estimator (1 or 2) for photon
float QualityEstimator(const int)
```

```
// moment (1 or 2) from CLMONS analysis
float Moment(const int)

// Pi0 mass estmated from clmoms
float Pi0Mass()

// Expected fraction in 4 towers
float ExpectedEnergyFractionInCentralTowers()

// Geometrical correction
float GeometricalCorrection()

// Zero supression correction from Coradoc
float ZeroSupression()

// Probability to be a fake photon from Electromagnetic origin
float FakeEcalProbability()

// Probability to be a fake photon from Hadronic origin
float FakeHcalProbability()

// Pointer to the PGAC parent giving a fake photon
AlGamp* ParentGivingFake()

// Flag for fake determination
int FakeEquality()

// pointeur to PECO bank
int PecoObject()
```

8.3 AlMCtruth: the Monte Carlo truth information

```
// collection of mothers
AlephCollection<AlMCtruth*> getMotherVector()

// collection of daughters
AlephCollection<AlMCtruth*> getDaughterVector()

int PA() particle code

int NO() number of mothers

int ND() number of daughters

char* name() particle name
```

8.4 AlTrack: the track object

It gives a direct access to the FRTL, FRFT and TEXS bank entrie: TL(), TL2(), ... In addition, the following methods are defined:

```
float TMean() mean TM

float TL2sum() sum of TL2

float ADmean() mean of AD

int NSsum() total number of segments

// matching MC particles.
AlephCollection<AlMCtruth*>& getMatchingVector()

AlEflw* getEflw()  associated energy flow
```

8.5 AlThrust: the thrust object

The thrust object is build by the thrust algorithm.

```
Hep3vector getThrustDirection() returns the thrust direction.

float      getThrustValue()     returns the thrust value.
```

8.6 AlMuon: the muon object

```
AlEflw* getEflw() returns the associated energy flow object

AlTrack* getTrack() returns the associated track
```

Note that these values should be checked for being different from NULL, such as all the pointers returned by ALPHA++ methods.

In addition, the class gives access to the MUID bank.

Note about the Lorentz vector associated with the muons: By defnition, the Lorentz vector of a lepton is the one from the associated track.

8.7 AlElec: the electron object

```
AlEflw* getEflw() returns the associated energy flow object

AlTrack* getTrack() returns the associated track
```

Note that these values should be checked for being different from NULL, such as all the pointers returned by ALPHA++ methods.

In addition, the class gives access to the EIDT bank.

Note about the Lorentz vector associated with the electrons: By defnition, the Lorentz vector of a lepton is the one from the associated track.

8.8 AlJet: the jet object

Jet objects are returned by the various jet algorithms, as described later in this manual.

```
// list of the objects in the jet.
AlephCollection<AlObject*>& getObjects()

int getScheme()  scheme used to build the jet: 0 for E0, 1 for E.

int getMetric()  metric used for the jet: 1 for Jade, 0 for Durham.
```

8.9 AlTau: the tau object

The tau object is derived from AlJet and has thus all the jet properties and methods. In addition,

```
int getNch() returns the number of charged tracks.

float getEch() returns the total energy of charged tracks.

AlEflw* getEflw() returns the associated main Energy flow object.
```

By definition, the main Energy flow object is the most energetic one with the same sign as the tau.

8.10 AlBjet: the b-jet object

AlBjet is a jet, and thus has all the jet properties and methods. In addition, float getBprobability() returns the QIPBTAG probability for that jet.

9 Vertex objects

There are 2 vertex classes:

- AlVertex describes the vertices from the BOS.
- AlUsrVertex describes a user-defined vertex.

The methods are the QvrtBase ones. Nevertheless, AlUsrVertex returns a dummy ouput for KVN, QVCHIF and QVEM. For a AlUsrVertex, setA4V(HepLorentzVector) has to be used to set the position and "energy" of the vertex.

By definition, the user vertex type is 6.

9.1 Vertex Fitting

The AlVertexFitter class provides a general interface to vertex fitting routines. In the present implementation, the fit algorithms are those of the Fortran YTOP package in ALEPHLIB.

The vertex fit refits the track parameters of a set of input tracks and calculates the 4-vector and error matrix of the new track (the reconstructed decaying object) at the fit vertex. Depending on the fit constraints to be set, three different classes are available, which are derived form the general vertex fitter class. An object of one of those classes must be instanciated once:

```
Vfit my_vfit;          // simple vertex fit (no additional constraints)
MassCVfit my_mcvfit;     // mass constrained fit
VertexCVfit my_vcvfit;     // vertex constrained fit
```

For the latter two, additional constraints have to be specified, e.g.:

```
double K0s_mass = 0.511;
my_mcvfit.setMassC(K0s_mass);

AlVertex* theMainVertex = event.MainVertexP();
my_vcfit.setVertexC(theMainVertex);
```

Note that the vertex is passed as a pointer. The constraints can be changed anytime, e.g. in order to test different mass hypotheses.

The actual fit is performed by calling the method `doFit` with an AlephCollection of pointers to QvecBase objects (AlTrack, AlUserTrack or AlEflow). The number of input tracks in the collection must not be larger than 10.

```
AlephCollection<QvecBase*> my_collection;
...
```

```
// track selection
...
AlUserTrack mother_track;
AlUsrVertex end_vertex;
bool failed = my_vfit.doFit(my_selection, mother_track, end_vertex);
```

If the fit has been sucessful, the return value will be false and the reconstructed track and the vertex it is pointing to will be returned through the AlUsrVertex and AlUserTrack argument, respectively, The AlUserTrack will also have a pointer to the end vertex and a vector of pointers to the tracks it has been build of. The χ^2 of the fit is accessible through the QVCHIF() method of AlUserVertex. Kinematic quantities and covariance matrices can likewise be accessed by the respective methods of AlUsrVertex and AlUserTrack. Note that the track parameters of the input tracks remain unchanged, and that no new objects are added to the ALPHA banks.

10 Event topology and physics routines

All the routines described in this chapter perform loops over the elements of an AlephCollection. The objects to be considered are selected by removing "bad" objects from the collection. In addition, locked objects are not used.

10.1 Missing energy, mass, momentum

Two methods can be used to compute such quantities:

- GetSum() computes the sum QvecBase Object.
- GetMiss(qelep) computes the missing QvecBase object, with respect to qelep.

In both cases, the result is returned as a full QvecBase object, having a Lorentz vector. In the first case, this vector is the sum of all the vectors of the collection the method is applied on. In the second case, this is the result of the difference of this sum with a vector of energy qelep (in most of the cases the LEP energy) and momentum 0. From the resulting objects, it is easy to get the enery (QE()), mass (QM()) or momentum (QP()).

Example:

```
//To compute the missing transverse momentum:
```

```
// get the collection of energy flow objects
AlephCollection<AlEflw*> myobjects = EventInfo.EflwPV();
// compute the mmissing object
QvecBase mymissingobject =
          myobjects.GetMiss(EventInfo.Event().Energy());
// get transverse momentum
cout << mymissingobject.QPT() << endl;
```

10.2 Event topology

They are four general topology routines:

- float Sphericity()
- float Aplanarity()
- float Planarity()
- AlThrust AThrust()

The AlThrust object includes the thrust direction and value. They are accessed using the getThrustDirection() and getThrustValue() methods.

Example:

```
//To compute the thrust:

// get the collection of energy flow objects
AlephCollection<AlEflw*> myobjects = EventInfo.EflwPV();
// compute the thrust object
AlThrust mythrust = myobjects.AThrust();
// get the thrust value
cout << mythrust.getThrustValue() << endl;
```

10.3 Jet clustering

The content of a collection can be clustered to form jets. Two kinds of metric are available: Durham and Jade. Two schemes can be used for each metric: E (energy/momentum conservation) and E0 (momentum is rescaled to get a massless particle).

To use the Durham metric, call:

```
DurhamJet(float theycut, int scheme, float energy)\\
```

The scheme is 0 for E0 and 1 for E. The third argument is the event energy. If it is set to 0, the total visible energy of the set of objects is used. In the same way, JadeJet can be choosen to use the Jade metric. In both cases, a fixed number of jets can be obtained setting the ycut to -n (where n is the number of jets wanted).

The object produced is an AlephCollection of AlJet. The AlJet object is derived from QvecBase, and is described in a previous section.

Example:

```
// to compute jets:

// get the collection of energy flow objects
AlephCollection<AlEflw*> myobjects = EventInfo.EflwPV();

// call the method DurhamJet
float Ycut = -2.;// force 2 jets
int scheme = 1;  // E-scheme
AlephCollection<AlJet> Jn = myobjects.DurhamJet(Ycut, scheme, 0);

// get the momentum of each jet
for(AlephCollection<AlJet>::iterator ijet=Jn.begin();
                                     ijet<Jn.end();
                                     ijet++)
    cout << ijet->QP() << endl;
```

10.4 B-tag

The QIPBTAG ALPHA routine has been wrapped from FORTRAN. It produces a set of b-tagged jets.

This method is NOT defined in AlephCollection, so the syntax is quite different. You have first to instanciate the toolbox:

```
AlToolBox mytoolbox;
```

Then, just call:

```
pair< AlephCollection<AlBjet>, vector<float> > result;
result = mytoolbox.ProduceBjets(EventInfo);
```

As usual, EventInfo is the name of the AlphaBanks object.
The resulting pair contains:

- an AlephCollection of AlBjet
- a vector of the probabilities for each track to come from the main vertex. If the probability has not been computed by QIPBTAG, the value is set to 0. The last element of this vector contains the event probability.

10.5 Lepton identification

10.5.1 fundamental principles

Muon identification is based on the search for a signal from the muon chambers, associated to the tracking in the hadronic calorimeter. Every track with enough energy is extrapolated to the HCAL, where the number of active planes, and the signal multiplicity are looked for. Three quantities are defined: N_{fir} is the number of fired planes, N_{exp} is the expected number along a muon path and N_{10} is the number of fired planes within the 10 allowed planes. These quantities are shown on figure 2.

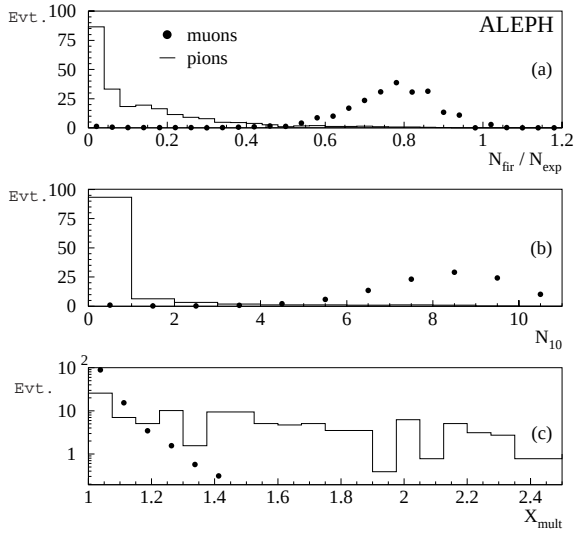

Figure 2: Muon id quantities

A muon candidate is validated if:

- The track is a “good track” (standard requirements on P, TPC hits, d0, z0)

- $N_{fir}/N_{exp} > 0.4$, $N_{10} > 4$ and multiplicity < 1.4
- There is a matching muon chamber signal.

Electron identification is a bit more tricky. There are two possible ways to identify an electron: the energy pattern in the ECAL or the TPC dE/dx information might be used. The first possibility is more powerful, and will be prefered as long as the electron is not in a crack or an overlap. A set of estimators is built at reconstruction time, which describes, in units of standard deviation the difference between the expected mean value and the actual value.

- **R1** (energetic balance) = $\frac{E-p}{\sigma_1}$
- **R2** (transverse shape) = $\frac{E_4/P-\langle E_4/P\rangle}{\sigma_2}$
- **R3** (longitudinal shape) : $\frac{a-\langle a\rangle}{\sigma_3}$
- **R4** (longitudinal shape) : $\frac{b-\langle b\rangle}{\sigma_4}$
- **R5** (specific ionisation) : $\frac{I-\langle I\rangle}{\sigma_5}$

E and p are the energy and the momentum, E_4 is the energy deposit in the 4 closest ECAL towers. The parameters a and b describe the longitudinal profile, using the following fit:

$$\frac{dE}{dS} = S^{1/b-1}e^{-a/b S} \ .$$

Figure 3 shows the R2 distribution ($R_{transverse}$), R3 distribution ($R_{longitudinal}$) and R5 distribution(R_I).

Hard cuts ($-1.5 < R2$, $-2 < R3 < 3$ et $-3 < R5$) give a high purity (99%) and 60% efficiency. An isolated electron is selected requiring $R_2 > -3$.In this case the efficiency is 99%[2].

An electron candidate is validated if

- The track fullfills the previously described requirements.
- The track is a "good track" (standard requirements on P, TPC hits, d0, z0).
- It is not already flagged as a muon.

The tau identification is the most difficult. Only the decay products are seen, so that we'll only see one of the following decays:

[2]Those values are LEP2 specific and used for calibration purposes. The actual values to use are set by the user (see next section).

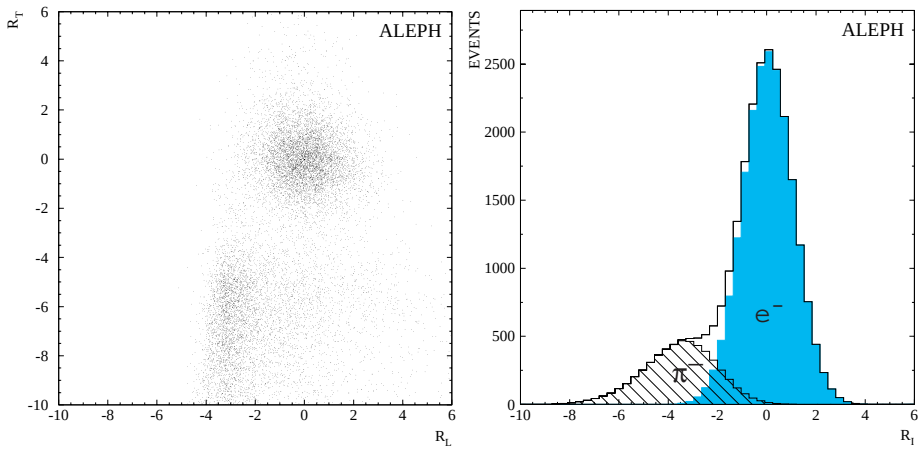

Figure 3: Electron id quantities.

- $\mu^{\mp}\bar{\nu}_{\mu}\nu_{\tau}(\gamma)$ (17.37%)
- $e^{\mp}\bar{\nu}_{e}\nu_{\tau}(\gamma)$ (19.56%)
- $h^{\pm}, \geq 0$ neutrals, $\geq 0\ K^{0}_{L}$, ν_{τ} (49.51%) ("1 prong")
- $h^{\mp}h^{\mp}h^{\pm}, \geq 0$ neutrals, ν_{τ} (15.18%) ("3 prong")
  h is a π or a K.

The two first cases are covered by the muon/electron identification. One or Three charged tracks will be looked for, with additional HCAL activity, making a mini-jet with a mass close to the tau mass. The Jade metric is used with

$$y_{cut} \approx \left(\frac{6}{E_{LEP}}\right)^{2}$$

and the E scheme.

A tau candidate is validated if:

- A Jade jet is formed with one or three charged tracks.
- Tracks are standard "good tracks"
- The total charged energy is large enough (as fixed by the pmin cut).

10.5.2 ALPHA++ implementation of lepton identification

To identify and manipulate leptons, three classes are defined: AlMuon, AlElec and AlTau. Classes describing electrons and muons give access to data used to select them. There is also a pointer to the associated track and energy flow object. The tau object is derived from AlJet.

It is possible to separately select electrons, muons and taus. In both cases, the procedure is the same (figure 4):

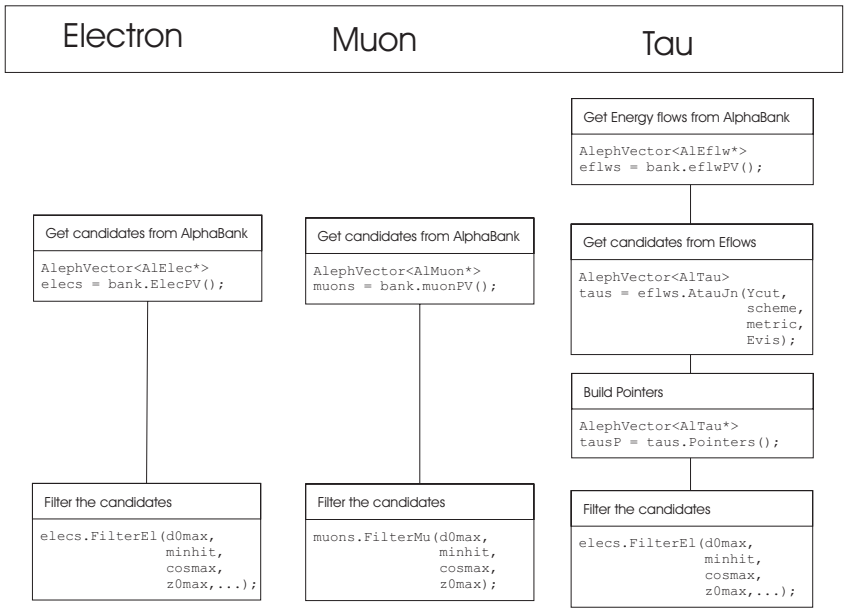

Figure 4: Selection procedure for electrons, muons and taus.

Get a list of candidates

Electron and muon idendification parameters are stored in two banks: EIDT and MUID. The first step is intended to get a set of candidates. For electrons and muons, this is a set of all the objects with a EIDT (MUID) entry. The Lorentz vector is taken from the associated track. For the taus, this is a user-generated set, produced using the *AtauJn()* method.

The list is thus obtained using:

- `AlephCollection<AlMuon*> muons = bank.muonPV() ;`
- `AlephCollection<AlElec*> elecs = bank.elecPV() ;`

- ```
  AlephCollection<AlEflw*> eflws = bank.eflwPV() ;
  vector<float> taidcuts = theCardsReader()->TAIDcuts();
  // parameters are: Ycut,scheme,metric,Evis
  AlephCollection<AlTau> taus = eflws.ATauJn(taidcuts,Evis);
  AlephCollection<AlTau*> tausP = taus.Pointers() ;
  ```

Note the tau-specific step, where the user produces a collection of pointers from the collection of candidates.

Standard parameters for *AtauJn()* are:

Parameter	Value	Description
Ycut	$\simeq 0.001$	Cut used to build the jets.
Scheme	0	0 for E0 ou 1 for E
Metric	1	1 for Jade, 0 for Durham.
Evis	QELEP	Total energy.
		If 0 is chosen, the visible energy is used.

The standard usage is thus: AtauJn(ycut,1,1,E_{LEP}), with $y_{cut} = \left(\frac{5.8}{E_{LEP}}\right)^2$.

Filter the candidates

The second step is the actual filtering, using more precise cuts.

```
vector<float> muidcuts = theCardsReader()->MUIDcuts();
vector<float> elidcuts = theCardsReader()->ELIDcuts();
vector<float> taidcuts = theCardsReader()->TAIDcuts();
muons.FilterMu(muidcuts) ;  // d0max,pmin,minhit,cosmax,z0max
//parameters are: almup,d0max,pmin,minhit,cosmax,
//                z0max,nwirem,r3cut,r2cut0l, r2cut1l,
//                r2cut2l,r2cuth,r5cutl,r5cuth
elecs.FilterEl(almup,elidcuts);
tausP.FilterTau(taidcuts) ;   // pmin,d0max,minhit,cosmax,z0max
```

Parameters are taken from the driver - as in the example - and reflect the MUID, TAID and ELID parameter cards or may be set “manually” by the user. If a card is not provided, default values are used.

A vector of muons, which can be empty, must be given to the electron filtering algorithm. It is used to grant that no electron is alreday flagged as a muon. If an empty vector is given, the test has to be done afterwards by the user.

Default parameters are the same as in *ALPHA*:
```

Parameter	Value	Description
d0max	0.5 (2 for τ)	minimal distance to the beam
Pmin	1	minimal lepton momentum
Minhit	4	minimal number of hits in the TPC
Cosmax	0.95	maximal cosine of the track-beam angle
Z0max	10	minimal distance to the interaction point along z
Nwirem	40	minimal number of usefull wires for dE/dx
R3cut	1000	upper limit on R3 (dE/dx)
R2cut0l	-3	lower limit for R2 in a clean HCAL area
R2cut1l	-7	lower limit for R2 in a crack
R2cut2l	-5	lower limit for R2 in a overlap
R2cuth	1000	upper limit for R2.
R5cutl	-0.5	lower limit for R5.
R5cuth	1000	upper limit for R5.

10.6 Kinematical fit

Constrained kinematical fits on an event can be performed in ALPHA++ thanks to an interface to the (fortran) ABCFIT program. A detailed description of the ABCFIT program can be found in [7]. The original code used in ALPHA++ was taken in the directory:
shift50: /aleph/wwtf/Common_Ntuples/tools/w1.31/abcfit/
and corresponds to a slightly upgraded version compared to one described in [7].

10.6.1 Description of ABCFIT

The ABCFIT package is designed to do constrained kinematic fitting of multi-particle events. The ABCFIT package employs an analytical approach by means of Newton-Raphson minimisation combined with Lagrange multipliers, resulting in an iterative algorithm.
Some key features include:
- variable number of particles (maximum 7 are allowed);
- the concept of unmeasured particles;
- numerous parameterisations are included and it is possible to introduce additional;
- non-diagonal input covariance matrices of fit parameters;
- parameter values and covariance matrices can be determined as a function of true or measured quantities;
- user specified constrain values;
- Gaussian or Breit-Wigner behaviour of masses.

To run the program, an external files is needed, which name depends on the options chosen for the fit: the file `aibi_evol_<itypp*njet>_<itevol-k0n>.dat` is used for parametrisation definition and parameter evolution. Each particle momentum is described via a longitudinal momentum scale, a, and two transverse components, b and c. In the option PARID(i) = kmn, n chooses the kind of jet parametrisation:

```
n  = 0 ALEPH like parametrisation
         --->      -->      ------>      ---->
         pi_r = ai*pim + bi*u_theta + ci*u_phi
n  = 1 DELPHI like parametrisation
         --->           -->      ------>      ---->
         pi_r = exp(ai)*pim + bi*u_theta + ci*u_phi
```

The DELPHI parametrisation has 2 main advantages: the ai distribution looks like more Gaussian for low momentum jet, and it avoids fit results with reverse jet momentum. The option ITEVOL = kmn fixes the choices on the variation of ai/bi/ci parameters with jet momentum and angle, on the covariance matrix and on the reference system: use the particles true or measured values for binning. The preferred values are PARID(i) = 1 and ITEVOL = 1 for the fit, which then need the file `aibi_evol_1111_001.dat` in case NJET=4, or `aibi_evol_11111_001.dat` in case NJET=5. Examples of different parametrisation files used by ALEPH WW analysis at 183 GeV and 189 GeV can be found in [7].

The description of the IN/OUT variables of the routine is presented bellow, followed by an example of how to use ABCFIT in ALPHA++.

10.6.2 IN/OUT variable description

```
        F_ABCFIT(NJET,NUP,PX_REC,PY_REC,PZ_REC,PE_REC
         ,ITF,CVAL,MO,GO,PARID,ITEVOL
         ,PX_FIT,PY_FIT,PZ_FIT,PE_FIT
         ,CHI2T,NDF,IERR);
*============================================================
* IN:
*   NJET        = Number of objects in event =<7
*   NUP         = Number of Unmeasured Particles in event
*                 (must be last in p*_rec)
*   PX_REC(7)  = x-momentum of up to 7 objects
*   PY_REC(7)  = y-momentum of up to 7 objects
*   PZ_REC(7)  = z-momentum of up to 7 objects
*   PE_REC(7)  = energy     of up to 7 objects
*   ITF         = mn
```

```
*     ---> n    = 1 Four momentum and energy conservation,
*                    Som{j=1,njet}p_fit(j) = cval
*          n    = 2 Energy momentum conservation
*                 + m(1,2) = m0(1)
*                 G0 is not used
*          n    = 3 Energy momentum conservation
*                 + alpha(1,2)*m(1,2) = m0(1)
*                 sigma(alpha) = G0(1)/m0(1)
*          n    = 4 Energy momentum conservation
*                 + m(1,2)-m(3,4) = 0
*          n    = 5 Energy momentum conservation
*                 + alpha(1,2)*m(1,2)-alpha(3,4)*m(3,4) = 0
*                 sigma(alpha(1,2)) = G0(1)/m0(1)
*                 sigma(alpha(3,4)) = G0(2)/m0(2)
*          n    = 6 Energy momentum conservation
*                 + m(1,2)=m0(1)  m(3,4)=m0(2)
*          n    = 7 Energy momentum conservation
*                 + alpha(1,2)*m(1,2)=m0(1)  alpha(3,4)*m(3,4)=m0(2)
*          n    = 8 Energy momentum conservation
*                 + E(1,2)=E(3,4) "equal energy"
*     -->  m    = 0 In case of mass constraint, Gaussian fit is used
*               = 1 In case of mass constraint, Breit-Wigner fit is
*               used, i.e. alpha parameters have a BW distribution
*   CVAL(4)     = value of 4-momentum constrains Px,Py,Pz,E
*               (e.g. 0,0,0,ecm). Can be used to switch of some of the
*               constraints:
*               If abs(cval(1)).gt.abs(cval(4)) Px constraint is not
*               applied. Same holds for cval(2) (Py) and cval(3) (Pz).
*               If cval(4).lt.0.0 E constraint is not applied, BUT
*               remember to set |value| such that you don't switch of
*               Px, Py and Pz constraints by accident....
*   M0(3)       = value of up to 3 mass-constrains (e.g. mw,mw)
*   G0(3)       = width of up to 3 mass-constrains (e.g. w-width)
*   PARID(7)    = kmn
*            n  = 0  Fit parameterisation a la ALEPH
*               = 1  Fit parameterisation a la DELPHI
*               = 2  Fit parameterisation using P,theta,phi
*               = 3  Fit parameterisation using Px,Py,Pz
*            m  = 0  Fitted jet energies scale with fitted momenta
*               1  Fitted jet energies is determined from fixed input
*                  jet mass and fitted momenta
*               2  Fit jet mass using a 4'th parameter a la ALEPH:
*                        M_fit = d*M_reco
```

```
*                 3  Fit jet mass using a 4'th parameter a la DELPHI:
*                        M_fit = exp(d)*M_reco
*                 4  Fit jet mass using a 4'th parameter:
*                        E_fit = (exp(d+log(E_reco/P_reco-1))+1)*P_fit
*            k  = 0  Full correction on jet momentum (// and T)
*               = 1  Only correction on jet momentum (//)
*               = 2  Full correction on jet momentum, but longitudinal
*                    parameter is assumed to be unmeasured
*               = 3  switch on k=1 _and_ k=2
*   ITEVOL      = kmn
*            n  > 0  Evolution with flavour/jet characteristics:
*            m  = 0  Diagonal input covariance matrix
*               = 1  Full non-diagonal input covariance matrix
*            k  = 0  binning in reco values for correction-factors
*               = 1  binning in true values for correction-factors
* OUT:
*   PX_FIT(7)  = fitted X-momentum of up to NPARTICLES objects
*   PY_FIT(7)  = fitted Y-momentum of up to NPARTICLES objects
*   PZ_FIT(7)  = fitted Z-momentum of up to NPARTICLES objects
*   PE_FIT(7)  = fitted energy     of up to NPARTICLES objects
*   CHI2T       = total chi2
*   NDF         = number of degree of freedom
*   IERR        = error flag
*
* ERROR OUTPUT:
*   IERR        <  0 results are irrelevant
*               >= 0 results fulfils constrains and can be used
*   (for details on IERR ouput value, see:
* http://alephwww.cern.ch/~hansenjo/ALEPH-ONLY/abcfit/callstruc.html)
*
```

10.6.3 Example

In this piece of *ALPHA++* code, one forces the event to be clustered into 6 jets, and a kinematical fit is performed with the option ITF=1, i.e. momentum and energy conservation. The options chosen are ITEVOL = 1 (diagonal reco-binning) and PARID(i) = 1 (fit parametrisation a la DELPHI), and the code needs the parametrisation file available on the web: `aibi_evol_111111_001.dat`.

```
// define input and output of ABCFIT
  float PX_REC[7];
  float PY_REC[7];
```

```
  float PZ_REC[7];
  float PE_REC[7];
  float PX_FIT[7];
  float PY_FIT[7];
  float PZ_FIT[7];
  float PE_FIT[7];
  float CVAL[4];
  float M0[3];
  float G0[3];
  float CHI2T;
  float CHI2;
  int NJET;
  int NUP;
  int ITF;
  int ITEVOL;
  int ITYPP[7];
  int NDF;
  int IERR;
  for (int i=0 ; i<7 ; i++)
  { PX_REC[i]=0.;
    PY_REC[i]=0.;
    PZ_REC[i]=0.;
    PE_REC[i]=0.;
  }
// set values of mass and width
  for (int i=0; i<3; i++) M0[i]=0.;
  for (int i=0; i<3; i++) G0[i]=0.;
// set values of four-momentum constrains for the fitted system
// QELEP is the centre-of-mass energy of the initial state.
  CVAL[0]=0.;
  CVAL[1]=0.;
  CVAL[2]=0.;
  CVAL[3]=QELEP;

// make 6 jets using Durham PE
  Ycut = -6.;      // force 6 jets
  scheme = 1;      // E-scheme
  AlephCollection<AlEflw  *> alefp=EventInfo.EflwPV();

  if (alefp.size() > 5)
  {
    f6jets = alefp.DurhamJet(Ycut, scheme, 0);
    f6jetsP = f6jets.Pointers();
```

```
    QvecBase::SortCriterium = 3;
    // sort the vector using the component 3 (e) of A4V
    sort(f6jets.begin(),f6jets.end());
    for (int i=0; i<6 ; i++) f6j[i]=f6jets[5-i].A4V();

// ABC fit info:
      for (int i=0 ; i<6 ; i++)
      { PX_REC[i]=f6j[i].x();
        PY_REC[i]=f6j[i].y();
        PZ_REC[i]=f6j[i].z();
        PE_REC[i]=f6j[i].e();
      }

// 6 jets:  no mass fixed;  ITYPP[i]=1 for all i
      NJET=6;
      NUP=0;
      ITF=1;
      ITEVOL=1;
      for (int i=0; i<7; i++) ITYPP[i]=1;
      F_ABCFIT(NJET,NUP,PX_REC,PY_REC,PZ_REC,PE_REC
       ,ITF,CVAL,M0,G0,ITYPP,ITEVOL
       ,PX_FIT,PY_FIT,PZ_FIT,PE_FIT
       ,CHI2T,NDF,IERR);

      if (IERR > -1)
      {
// recompute some variables using PX_FIT,PY_FIT,PZ_FIT,PE_FIT
// save the chi2 value
      CHI2=CHI2T/NDF;
      }
  }
```

11 Interactive Mode

The specific AlVisu documentation can be an interesting complement to this section.

All the already presented features are mainly used in the so called “batch” mode. In this mode, *ALPHA++* runs over all the specified events, and generaly fills a tree/ a ntuple.

Another possible mode in the “interactive” mode. In interactive mode, *ALPHA++* will wait for a coded request from stdin and reply to stdout. A typical use of this is to connect stdin/stdout to the stdout/sdtin of an external program. This is the way AlVisu, the *ALEPH OO Visualisation Tool*, gets data from *ALEPH* databases.

Internally, there is an AlephInteractiveHandler which will send and receive messages in interactive mode. This will not be discussed here.

There is a set of possible requests, and the corresponding answers, predefined in the driver, in order to perform the basic data extraction of evente information, tracks, leptons and jets. The full protocol is presented in annexe C.

In addition, the user can define his own functions, to call to produce a given result in interactive mode. This is done by subclassing the AlephInteractiveHandler and by instanciating the new class via the templated AlephRegisteredAction<T>.

Shortly, what you have to do:

- Create a new class, deriving from AlephAbstractInteractiveFunction.
- Implement `virtual string Name()` returning the name of the function
- Implement `virtual int Code()` returning a number larger than 500
- Implement `virtual vector<pair<string,float> > OptionsList()` returning a vector of pair (option name, default value)
- Implement `virtual void Run(vector<float>& options,AlphaBanks& EventInfo)` the actual function (UserEvent-like)
- In `UserInit()`, instanciate `AlephRegisteredAction<T>`, templated with your new class.

The Run method is called when needed with a vector of options and a reference to the AlphaBanks, so that you can do there all what you are used to do in UserEvent. In addition, there is a method

`void SendMessage (int code, vector< float > &options, string comment)`

available, that can be used to send well-formated messages (with code, options and comment. See annexe C for more details). Nevertheless, you must respect some conventions in order to have AlVisu understanding your function.

When returning the objects resulting from your calculation, the first option, called the index, must be a decreaing number, equal to zero for the last object returned. The second argument must be the object type. 1: a track, 2: an energy flow, 3: a lepton, 4: a jet. The next options are then function of the object type:

- for a track: px, py, pz, ch
- for an eflow: E, px, py, pz, ch
- for a lepton: E, px, py, pz, ch, flavour(1:e 2:μ 3:τ)
- for a jet: $cos(\theta)$, px, py, pz. $cos(\theta)$ gives the cone half-angle.

An example can be found in the test1.cpp file. Here it is:

```
class testclass:public AlephAbstractInteractiveFunction
{
        public:
        testclass(AlephInteractiveHandler*ptr):
        AlephAbstractInteractiveFunction(ptr) {}
        virtual string Name() {return "test";}
        virtual int Code() {return 555;}
        virtual void Run(vector<float>& options,AlphaBanks& EventInfo)
                {
                  // dummy routine: returns always the same track,
                  // eflow and the same jet
                  vector<float> output;
                  output.push_back(2);
                  output.push_back(1); // a track
                  output.push_back(3); // px
                  output.push_back(0); // py
                  output.push_back(0); // pz
                  output.push_back(0); // ch
                  SendMessage(Code(),output,"Dummy test routine");
                  output.clear();
                  output.push_back(1);
                  output.push_back(2);  // an eflow
                  output.push_back(10); // E
                  output.push_back(0);  // px
                  output.push_back(10); // py
                  output.push_back(0);  // pz
                  output.push_back(1);  // ch
                  SendMessage(Code(),output,"Dummy test routine");
                  output.clear();
                  output.push_back(0);
                  output.push_back(4);   // a jet
                  output.push_back(0.5); // sin theta ~ radius
                  output.push_back(0);   // px
                  output.push_back(0);   // py
                  output.push_back(5);   // pz
                  SendMessage(Code(),output,"Dummy test routine");
                }
        virtual vector<pair<string,float> > OptionsList()
                {
                  vector<pair<string,float> > output ;
                  pair<string,float> mypair;
                  mypair.first = "option 1";
                  mypair.second = 10;
```

```
                output.push_back(mypair);
                mypair.first = "dummy 2";
                mypair.second = 3.1415;
                output.push_back(mypair);
                return output;
              }
};

void AlephExManager::UserInit()
{
        AlephRegisteredAction<testclass> mytestclass;
}
```

In this example, nothing physical is done: a set of messages is just build and send. It will produce a track, an eflow and a jet in AlVisu. A more interesting example is the code written for the default actions (event, tracks, leptons, jets); it can be found in the `DefaultInteractiveActions.cpp` file.

References

[1] http://alephwww.cern.ch/LIGHT/alpha.html

[2] http://www.doxygen.org/index.html

[3] http://cern.ch/aleph-proj-alphapp/doc/goals.html

[4] http://wwwinfo.cern.ch/asd/lhc++/HTL/

[5] http://root.cern.ch/root/RootDoc.html

[6] *ALEPH* Collaboration, "*ALEPH*: A detector for electron-positron annihilation at LEP", *Nucl. Inst. Meth.* **A294** (1990) 121.

[7] http://alephwww.cern.ch/~hansenjo/ALEPH-ONLY/abcfit/abcfit.html

A *ALPHA++* driver

The main class is the *AlephSession*. Three methods are called one after the other: Initialize(), Run() and Terminate(). Those three "states" are propagated to the different managers:

1. *AlephDbManager* controls the database access (BOS or Objectivity/Db) and maps it to the *AlphaBank*.

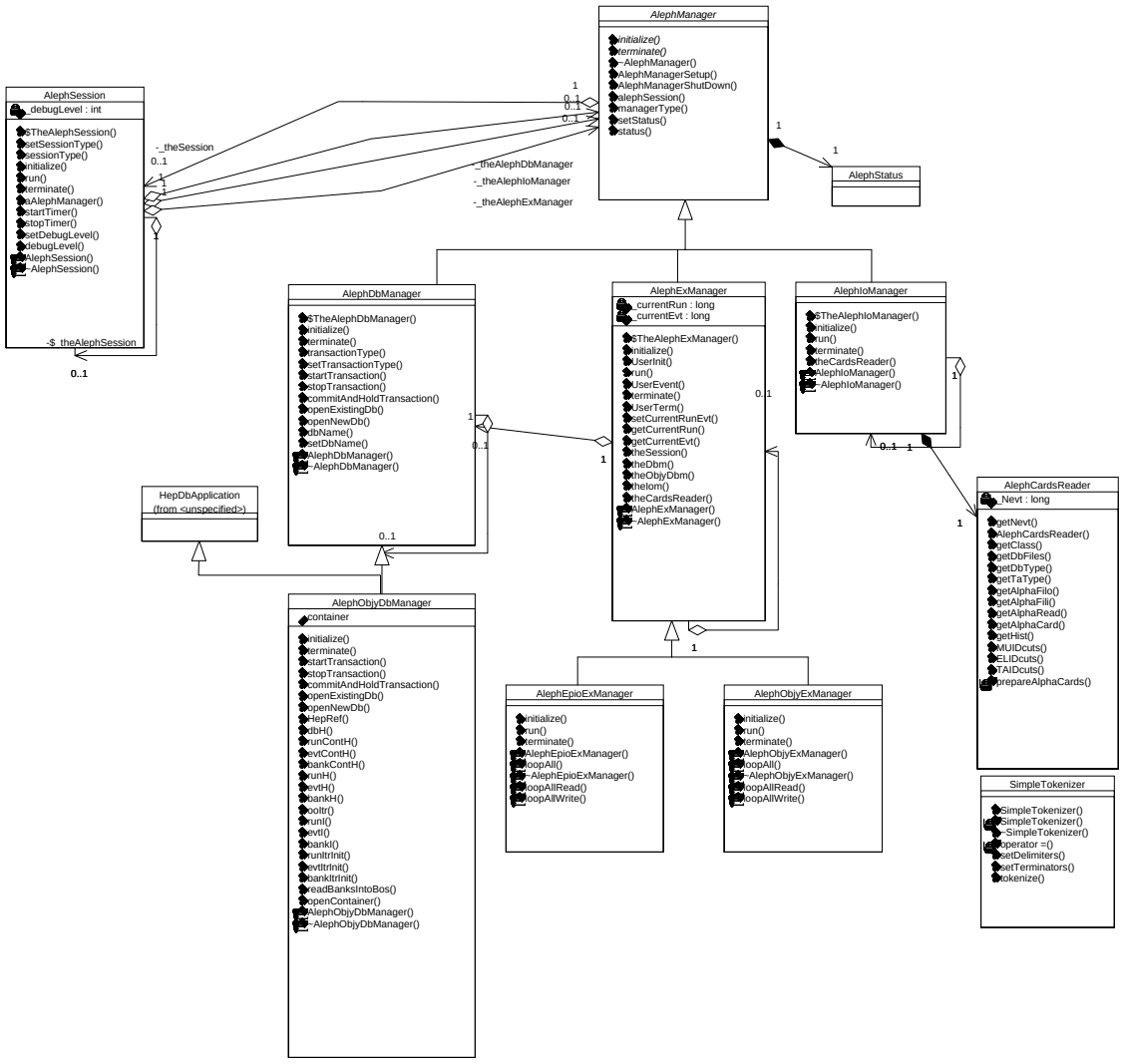

Figure 5: Simplified UML diagram of the *ALPHA++* Driver. We can see the AlephSession, controling a set of AlephManagers, derived from the base class AlephManager. On the right, the AlephCardReader accesses the configuration file.

2. *AlephIoManager* controls the IO operations. It accesses the configuration file and the output (root or hbook) file.

3. *AlephExManager* manages the main execution loop over runs and events and calls the *UserEvent* method.

Given the db type (BOS or Objectivity/Db), a specific instance of AlephDbManager and AlephExManager is used. Note that the managers are singletons.

B Analysis class diagram

Here is a simplified UML diagram of the ALPHA++ analysis part. This diagram is not complete but illustrates the main structure of the object representation of data. For the complete and most up-to-date diagrams, see the HTML documentation on the web page.

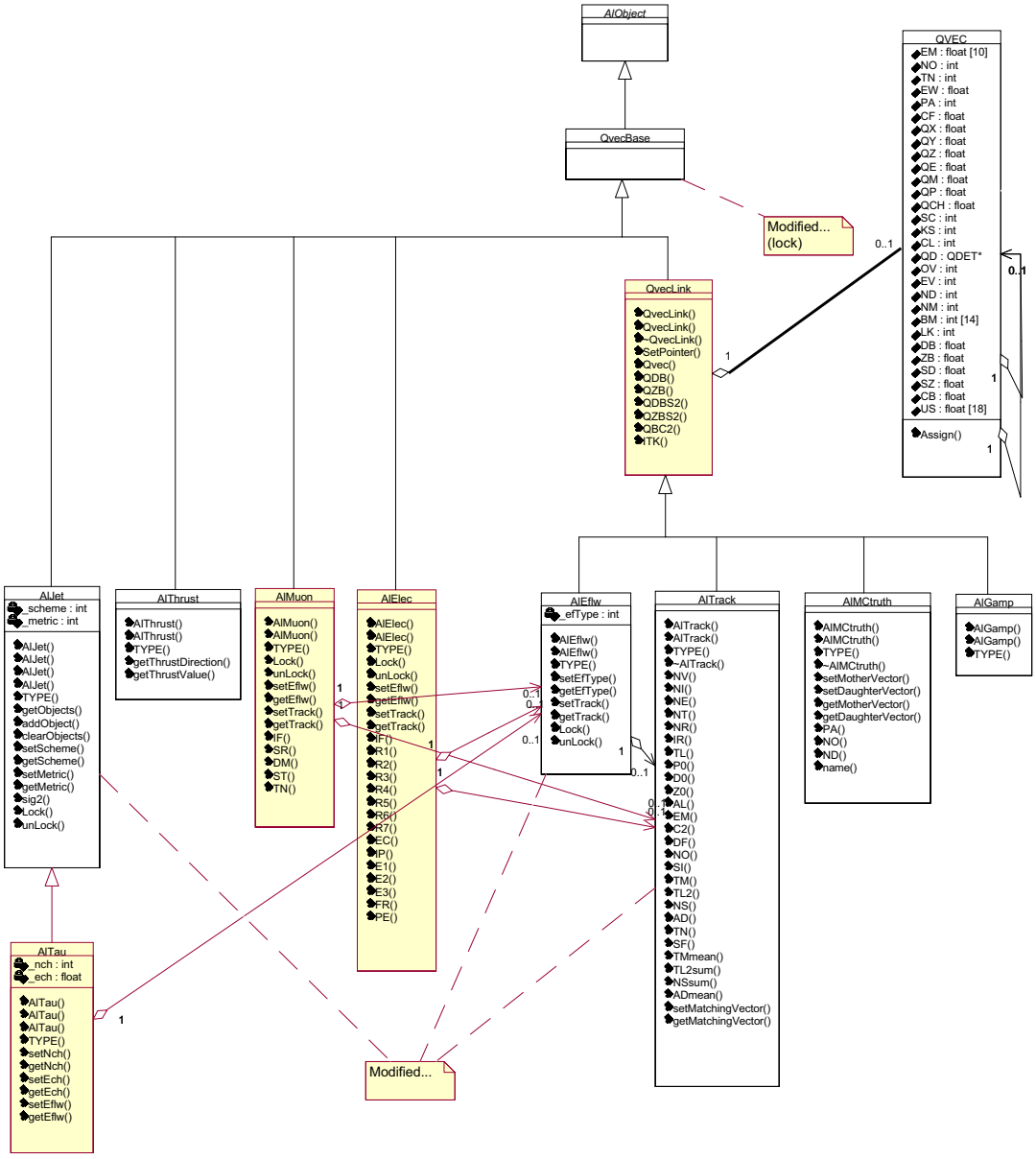

Figure 6: Simplified UML diagram for *ALPHA++*. Part 1.

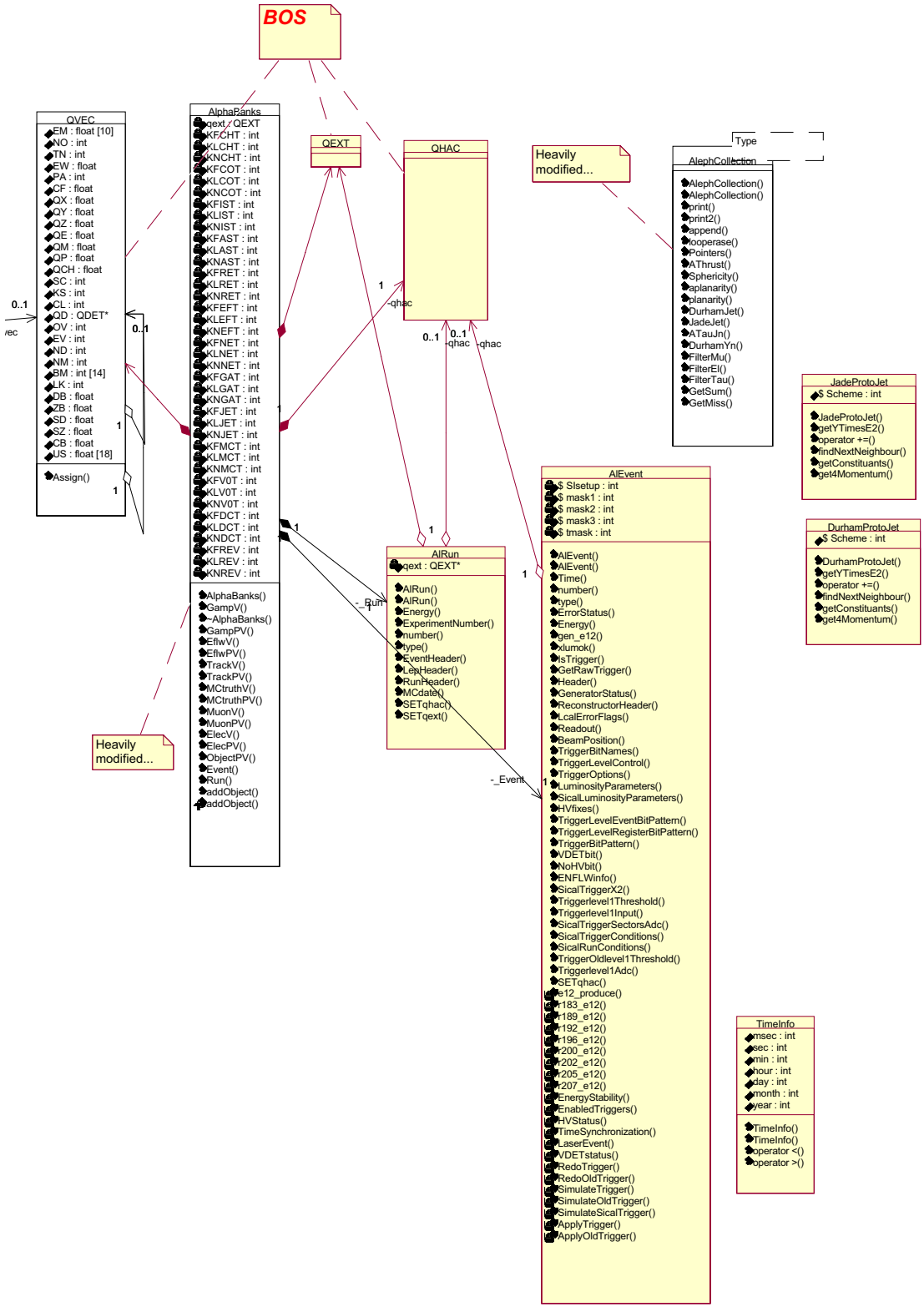

Figure 7: Simplified UML diagram for *ALPHA++*. Part 2.

C Full Interactive protocol description

This section will describe the present state of the protocol designed for an easy communication betwen *ALPHA++* and any external software. Up to now, only AlVisu is using this feature. This protocol is still allowed to change in the future, in order to be more general or flexible.

The main features of the protocol are:

1. It's based on the standard input/output. The advantages are that it can be directly used by the user, and that it can be "piped" for example with ssh.
2. Messages are simple mostly human-readable strings.
3. Build on a request - reply basis. The external component will always request some data, using a numeric code. *ALPHA++* will the reply with a set of messages containing the same code. The only exections are the initialisation and termination phases.

Any message must be formated like this:

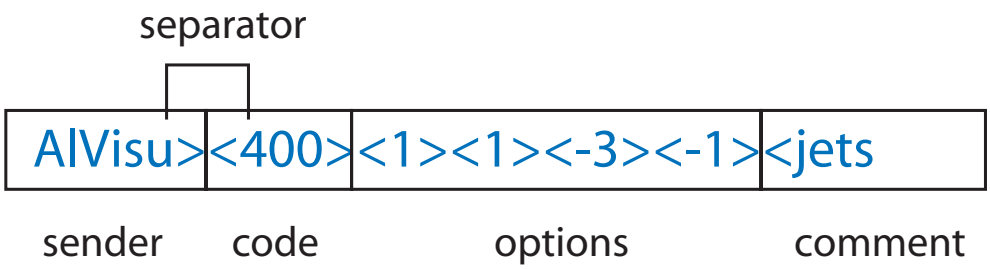

Figure 8: Standard message

It is made of a variable set of substrings, separated by "><". The first string is the sender name. Up to now, only *ALPHA++* and AlVisu are allowed. There is then the operation code (opcode) followed by a set of options. The first option is generaly called the index. The message is terminated by a comment string, generaly meaningless.

C.1 Initialisation phase

During the initialisation, *ALPHA++* will send a list of the available functions (standard and user's ones) with a list of the options (with default values).

The following sequence will be repeated for each function:

1. First, the function header is send:

code	-1
options	- function index ; number of options; function opcode
comment	function name

2. Then, for each option:

code	-1
options	function index ; option index ; function opcode ; default option value
comment	option name

3. When all the functions are send, *ALPHA++* will send the "Ready" message and wait for the first event request.

code	0
options	0
comment	Ready

C.2 Normal - Data exchange - phase

During the normal phase of data exchange, various messages are emited and received. There are the various user's messages, and some standard messages. The format of the user's messages will not be described here again. Please refer to the correspond section in the present manual. The standard messages correspond to default functions implemented.

next event

request:

code	100
options	0: all events 1: only events with UserEvent() returning true.

answer:

A set of 6 messages is send, with each time the code 100. The index varies from 1 to 6. Other options are:

index 1	event number
index 2	run number
index 3	year ; month ; day ; hour ; min ; sec
index 4	edir class
index 5	msum ; esum
index 6	acoplanarity ; acolinearity ; thrust ; thrust_x ; thrust_y ; thrust_z ; aplanarity ; planarity ; sphericity

tracks

request:

code	200
options	eflow ; Energy cut ; $cos\theta$cut ; sanity

Eflow can take 3 values: 0: no request; 1: tracks; 2:energy flows. Sanity is a pure boolean. If it is 1, only "good" tracks/eflows are selected.

answer:

code	200
index	from n to 0. Indicates the track index.
options	E ; px ; py ; pz ; charge

leptons

request:

code	300
options	flavour

flavour is a binary field. It is computed as $e + 2\mu + 4\tau$ and describes which type of leptons are requested.

answer:

code	300
index	from n to 0. Indicates the lepton index.
options	E ; px ; py ; pz ; charge ; flavour

flavour is equal to 1 for electrons, 2 for muons and 3 for taus.

jets

request:

code	400
options	metric ; scheme ; ycut ; E

metric is 0 for Jade and 1 for Durham. scheme is 0 for E0 and 1 for E. E is used as is if $E > 0$; if $E = 0$, the visible energy is used; if $= -1$, qelep is used.

answer:

code	400
index	from n to 0. Indicates the jet index.
options	$cos\theta$; px ; py ; pz

θ is the cone half-angle.

C.3 Termination phase

When the last event has been processed, ALPHA++ will terminate as it would in normal mode. Just before exiting, it will send the termination message:

sender	ALPHA++
code	999
options	0
comment	Done

The external component can request an earlier program termination by sending the message

sender	external (AlVisu)
code	999
options	0
comment	Terminate

As usual, comments are meaningless.

D How to find more information

All the available information can be found on the ALPHA++ website:
http://cern.ch/aleph-proj-alphapp .

In addition to this manual, there is :

- the doxygen documentation
- the up-to-date code
- sample analysis codes
- FAQ (to come)

In addition, it is recommended to consult the ALPHA manual.