ALEPH 99-097
SOFTWR 99-002
October 8, 1999

EUROPEAN CENTRE FOR PARTICLE PHYSICS (CERN)

CERN Summer Student Project: ALPHA++

Abstract

Thrust and Jet finding algorithms have been implemented and used to perform a QCD analysis within the framework of ALPHA++, which is an object oriented version of the ALEPH analysis package ALPHA. Especially the three jet-events have been considered and their energies, multiplicities and inter-jet-angles have been studied.

Maria Hörndl
Maria.Horndl@cern.ch
e9525579@student.tuwien.ac.at

Supervisor: Günther Dissertori
Guenther.Dissertori@cern.ch

This work project was performed

by Maria Hörndl

during her stay as a Summer Student

at CERN from 29.6 - 24.9 1999

1 The ALEPH Detector at LEP

As the analysis has been done using data from the ALEPH experiment at CERN, a short description of this experiment is given first:

ALEPH is one of the four detectors at the large electron-positron collider (LEP). It has been designed to detect the products of e^+e^- collisions. Until 1994 LEP was running at a centre-of-mass energy (E_{cm}) of 91 GeV (LEP1) in order to persue its first aim, the investigation of the Z boson. From 1995 on the energy of LEP has been successively increased above the threshold of W pair production (LEP2) and has now reached a maximum of 200 GeV for the centre-of-mass energy. However, for this analysis only LEP1 data (from the year 1994) were taken.

The detector (Figure 1) consists of tracking components surrounded by the calorimeters. The tracking involves a silicon vertex detector (VDET), a cylindrical multiwire drift chamber (ITC) and a large time projection chamber (TPC) and serves for the measurement of charged particle tracks. The three detectors are immersed in a 1.5 T axial magnetic field from a superconducting solenoidal coil (situated between ECAL and HCAL). The calorimetry includes an electromagnetic calorimeter (ECAL) to identify electrons and photons, a hadron calorimeter (HCAL) for measurements of charged and neutral hadrons and a muon detection system (MUON) outside as the muons are the only particles penetrating through all the detector components. For a more detailed description of the different detector parts it is referred to [1], [2] or to the following WWW page:

`http://alephwww.cern.ch/EDUC/alephdesc.html`

.

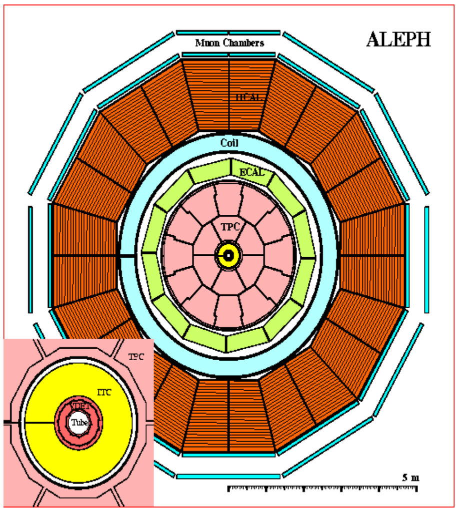

Figure 1: The ALEPH detector.

2 What is ALPHA++?

As now the trend in software strongly goes into the direction of object oriented (OO) programming, a new OO framework of the ALEPH physics analysis software is under development. Therefore a project called ALPHA++ has been set up. The ALPHA package which has served for analysis purposes until now was programmed in Fortran and organised in data structures called BOS banks.

The main goals of ALPHA++ are [3]:
• convert the ALEPH data from the Fortran BOS bank style into persistent objects and write them to an object database (Objectivity/Db)
• rewrite a mini version of the ALPHA package in object oriented C++ based on the object database
• compare the standard and OO performance with regard to efficient access of data
• test the software engineered by the RD45 and LHC++ projects
• provide input and experience for a possible archiving of ALEPH data

This Summer Student project concentrated on the second of the points listed above. Some algorithms, which of course already exist in ALPHA, were implemented into the C++ code, tested and compared to the Fortran version. Finally, a QCD analysis was developed.

3 Description of the ALPHA++ Analysis Program Class Diagram

In general there are two basic objects, namely the Tracks and the Vertices. A Track can be a charged track, a photon or a general Energy Flow Object [2]. The attributes of each Track are the total momentum with its components, the energy, the mass and the charge, in addition there can be more specific attributes depending on the type of the Track. Concerning the Vertices one must differentiate between the main vertex holding the position of the interaction point and secondary vertices.

This structure has been implemented in ALPHA++ (Figure 2, [3]) with the abstract classes *AlObject* for the Tracks and *AlVertex* for the Vertices with their basic attributes as virtual member functions, which means that no instances of these classes can be created. They act only as an abstract interface. The concrete transient classes *AlTrack* (charged tracks), *AlEflw* (Energy flow) and *AlGamp* (photons) are defined by inheritance from *QvecBase* which inherits from *AlObject* in case of the Tracks. For the Vertices the concrete transient classes *AlMainVertex* and *AlGenVertex* (secondary vertices), inheriting from *QvrtBase* which inherits from *AlVertex*, are created. These concrete classes are instantiated using the data contained in the *QVEC* objects (for the Tracks) and the *QVRT* objects (for the Vertices), which are filled from the objects *QVEC_ALPHA* and *QVRT_ALPHA*. These last two objects are copies of the memory used by the Fortran version of ALPHA.

The *AlphaBanks* class acts as a container for all the concrete classes mentioned before. These containers are organized as STL vectors using the Standard Template Library. When writing

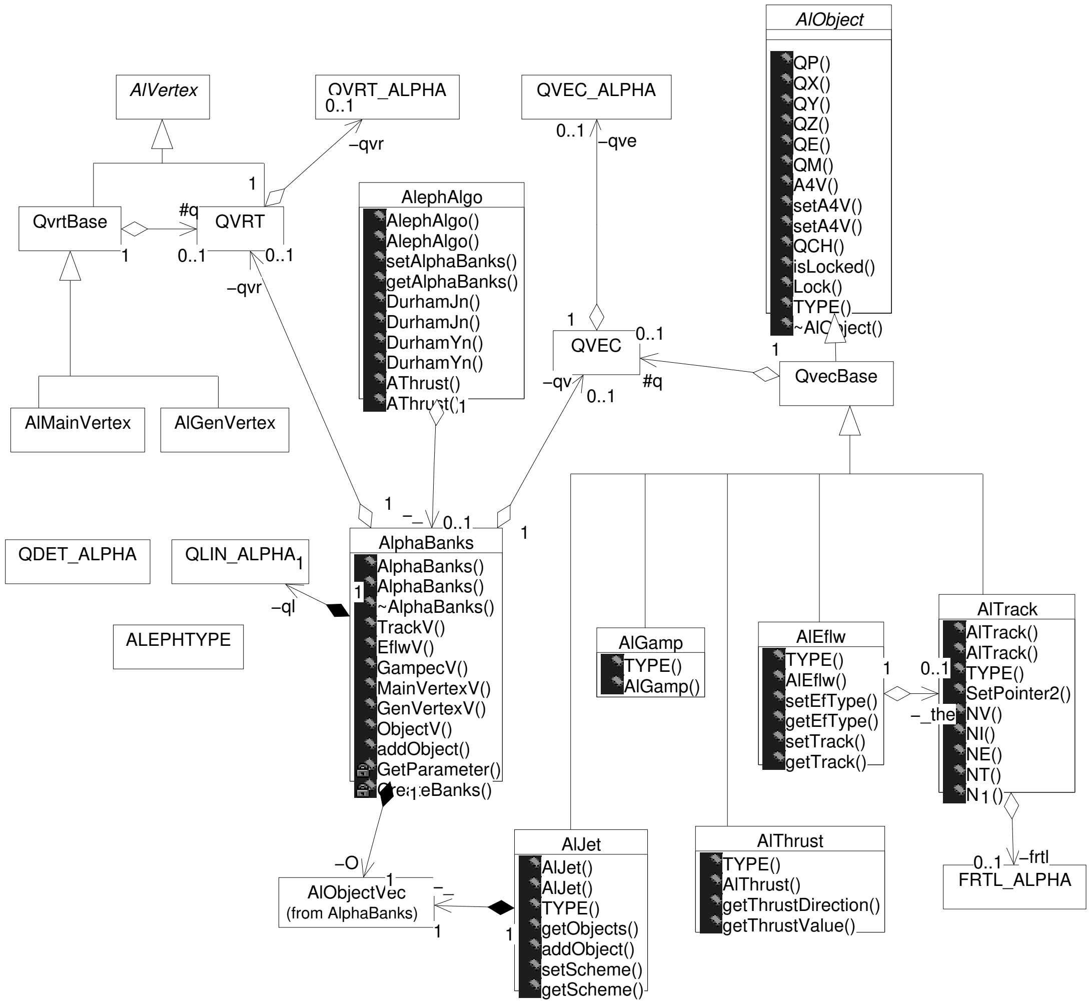

Figure 2: ALPHA++ analysis program class diagram.

a user analysis program one is only provided with a pointer to an *AlphaBanks* object, which then contains appropriate member functions to get access to all the other objects.

This structure already existed before this project was started. Now two additional classes have been added, which also inherit from *QvecBase* and therefore as well from *AlObject*, namely, the *AlThrust* and *AlJet* classes. These objects are created during runtime, when the Thrust and the Jet algorithms, which will be explained in the next chapters, are called and they of course involve all the specific properties of the Thrust and Jet objects.

The third new class, the *AlephAlgo* class, is an independent class. It actually contains the implementation of the Thrust and the Jet algorithms and it has a pointer to an *AlphaBanks* object to be connected to the objects used. So if the user wants to call one of the algorithms of the *AlephAlgo* class in his user analysis program, he has to include the Header file of *AlephAlgo*.

This is of course only a brief explanation of how the ALPHA++ classes are organized. Although this knowledge is not important for the QCD analysis which should be the main part of this report, it is nevertheless helpful for the understanding of the source code of the algorithms and the user analysis program which are listed in the appendix of this report.

4 Kinematics and Track Operations

First basic kinematics utility routines were added to the *QvecBase* class. These routines were taken over from ALPHA and adapted to the object oriented ALPHA++. Simplifications could be made using the CLHEP modules which provide three and four-vectors and useful operations on them.

The basic attributes of the Tracks are implemented in *QvecBase* as a four-vector called *_A4V* containing the energy and the (three-)momentum and a float value called *_qch* for the charge. All other properties for the Tracks can be calculated out of these five quantities.

For example, functions calculating $\beta = \frac{p}{E}$, $\gamma = \frac{1}{\sqrt{1-\beta^2}}$, the transverse momentum of a particle with respect to the beam line, the cosine of the polar angle (angle between the track and the beam line), the invariant mass of two particles, the scalar product of two momentum vectors, the sum of two four-momentum vectors, etc, were implemented. The source code of the *QvecBase* class, where all these functions can be found, can be looked up in the appendix.

In the next two chapters the more complex algorithms for calculating the jets and the Thrust axis of an event are explained.

5 The Thrust Algorithm

The Thrust value T is defined [4] as

$$T = \max_{\mathbf{n}_T} \frac{\sum_{i=1}^{m} |\mathbf{p}_i \cdot \mathbf{n}_T|}{\sum_{i=1}^{m} |\mathbf{p}_i|} \tag{1}$$

The sum goes over the three-momenta of all m available particles and the maximization is performed with respect to the unit vector $\mathbf{n}_T$. As the quantity is divided by the sum of the magnitudes of all three-momenta, the Thrust value is normalized and can therefore never take values greater than one. Searching for that vector $\mathbf{n}_T$, so that (1) becomes maximal, means looking for that axis for which the sum of the longitudinal momentum components along this axis is maximal. This axis $\mathbf{n}_T$ is called the Thrust axis of the event. These are the only properties an *AlThrust* object is supposed to have: a Thrust value and a Thrust direction.

It can be shown by induction that (the proof is listed in the appendix)

$$\sum_{i=1}^{k} (\mathbf{p}_i \cdot \mathbf{n}_T) = \max \tag{2}$$

for

$$\mathbf{n}_T = \sum_{i=1}^{k} \mathbf{p}_i \tag{3}$$

if

$$\mathbf{p}_i \cdot \mathbf{n}_T \geq 0 \qquad \forall i \in \{1, ..., k\}$$

The numerator of equation (1) can be written as

$$\sum_{i \in N} |\mathbf{p}_i \cdot \mathbf{n}_T| = \sum_{i \in A} (\mathbf{p}_i \cdot \mathbf{n}_T) + \sum_{j \in B} \left((-\mathbf{p}_j) \cdot \mathbf{n}_T\right), \quad N = \{1, 2, 3, ..., m\} \tag{4}$$

where

$$\begin{array}{llll} A \subset N & \textit{and} & \mathbf{p}_i \cdot \mathbf{n}_T \geq 0 & \forall i \in A \\ B \subset N & \textit{and} & \mathbf{p}_i \cdot \mathbf{n}_T < 0 & \forall i \in B \end{array}$$

As the total momentum must be zero in the e^+e^- annihilation, momentum conservation provides an additional condition:

$$\sum_{i \in N} \mathbf{p}_i = \mathbf{0} = \sum_{i \in A} \mathbf{p}_i + \sum_{j \in B} \mathbf{p}_j \tag{5}$$

This delivers

$$T = 2 \cdot \sum_{i \in A} (\mathbf{p}_i \cdot \mathbf{n}_T) = \max \quad \textit{for} \quad \mathbf{n}_T = \sum_{i \in A} \mathbf{p}_i \quad (\mathbf{n}_T \neq \mathbf{0} \ \textit{because} \ A \subset N) \tag{6}$$

Therefore equation (1) acquires the maximum for that $\mathbf{n}_T$ which is the 'longest' vector which can be found out of any subset $A \subset N$ of N, where 'longest' means $|| \, \mathbf{n}_T \, || = \max$ and $\mathbf{n}_T = \sum_{i \in A} \mathbf{p}_i$. Thus all different subsets (there are 2^{m-1} possibilities of dividing m particles into two subsets) have to be studied.

What does the implemented algorithm called *AThrust* do? Two of the m particles are selected to calculate the cross product of their three-momenta to define a certain direction. Then the 'sum' of the three-momenta of all the remaining particles is built: if the scalar product between the cross-product vector and the three momentum of the particle is greater than zero, then the momentum of the particle is added, and if the scalar product is less than zero, the three-momentum of the particle is subtracted. There are four possibilities of how to consider the three-momenta of the two selected particles according to the four possible sign-combinations. The length of the maximum vector thus obtained is stored. The calculation has to be repeated for all different combinations of selecting two out of the m particles and the length of the maximum vector ever gained has to be kept. This maximum vector is just the Thrust axis of the event if there is no missing momentum vector. In the algorithm there is an additional iteration to take a possible missing momentum vector into account.

Two cases have to be considered separately. First, if there is only one particle, then $T = 1$ is obtained, and the Thrust axis is the normalized three-momentum of that particle itself. For two particles the Thrust axis is the normalized vector of the sum of the two three-momenta if the scalar product of these three-momenta is greater than zero, or the normalized vector of the difference between the three-momenta if the scalar product is less than zero, and the Thrust value lies somewhere between one half and one. This must be true for any Thrust value no matter how many particles are considered in the calculation. To make this easily comprehensible, the two extreme cases should be mentioned.

One of them is that there are two back-to-back particles, in this case $T = 1$ is obtained. The other extreme case is a perfectly spherical m-particle configuration resulting in $T = 1/2$. For every other event a Thrust value between these two extreme cases will be measured.

The shape of the distribution of the Thrust value is rather sensitive to radiative corrections, such as gluon radiation off quarks.

The function *AThrust* which can also be found in the appendix implements the Thrust algorithm as explained above and has an *AlThrust* object as its return value. Getting the Thrust axis and the Thrust value back from this object is achieved by calling its member functions *getThrustDirection()* and *getThrustValue()*.

6 The Jet Finding Algorithm

Typical events seen by the detector are rather complex having many particles distributed in jets (bundles of particles) over the entire detector volume. The reasons why particles are not distributed uniformly over the whole detector are first, that the particular bremsstrahlung spectrum enhances logarithmically the probability to observe low-angle radiation and second, that the running coupling constant increases logarithmically with decreasing energy, thus giving an additional enhancement of multiple low-angle and low-energetic radiation.

The Durham clustering algorithm, which has been implemented in ALPHA++, defines the following resolution criterium for two particles [4]

$$y_{ij} = \frac{2\min\left(E_i^2, E_j^2\right)(1-\cos\theta_{ij})}{E_{vis}^2} \tag{7}$$

where E_i and E_j are the particles' energies, θ_{ij} is the angle between them, and E_{vis} is the energy sum over all particles in the final state. This so called metric y_{ij} is computed for all particle pairs i, j in the final state. The smallest one is stored and compared to some predefined cut-off parameter $Ycut$. If $y_{i_{min}j_{min}} > Ycut$, then the algorithm stops and all remaining particles are defined as jets. But if $y_{i_{min}j_{min}} < Ycut$, the two particles i_{min} and j_{min} are combined (clustered) to build a new pseudo-particle. There are several possibilities (recombination schemes) of combining the four-momenta of the two particles. For example, in the $E - scheme$ the four-momenta are simply added and in the $E_0 - scheme$ the three momenta are rescaled after their four-momenta have been summed up in order to get a massless pseudo-particle. So the number of particles is reduced by one after the clustering step. Then the metric y_{ij} is computed again for the new final state and in this way the algorithm goes on until $y_{i_{min}j_{min}} > Ycut$ as mentioned before.

Therefore an *AlJet* object has, apart from its momentum and its charge, which is just the sum of the charges of all corresponding objects, two additional properties: the scheme, which stores the recombination scheme and a STL-vector of pointers to these *AlObjects* of which the jet was built.

The *DurhamJn* algorithm, whose source code is given in the appendix, returns a STL-vector of *AlJet* objects where one can get the recombination scheme and the objects of which the jet was created by simply using the member functions *getScheme()* and *getObjects()*. The number of jets (jet rate) found is just the size of the vector.

7 The User Analysis Program

After having explained the two algorithms which are the basis for the QCD analysis, a brief description of the analysis program is given. Its source code can be found in the appendix.

Data from the year 1994 were studied. At this time LEP was running at a centre-of-mass energy in the energy range of the Z particle (around 91 GeV). As it is intended to examine the properties of quark and gluon jets, decays of the Z boson are studied. One possible decay mode of the Z is to decay into a quark and an antiquark, where each of them hadronizes instantaneously into several hadrons producing jets. Therefore a two-jet event can be measured in the detector. But it is also possible that one of the quarks radiates a gluon, thus causing a three-jet event to be seen in the detector. These three-jet events (Figure 3) as described in the next chapter are used to investigate properties of quark and gluon jets.

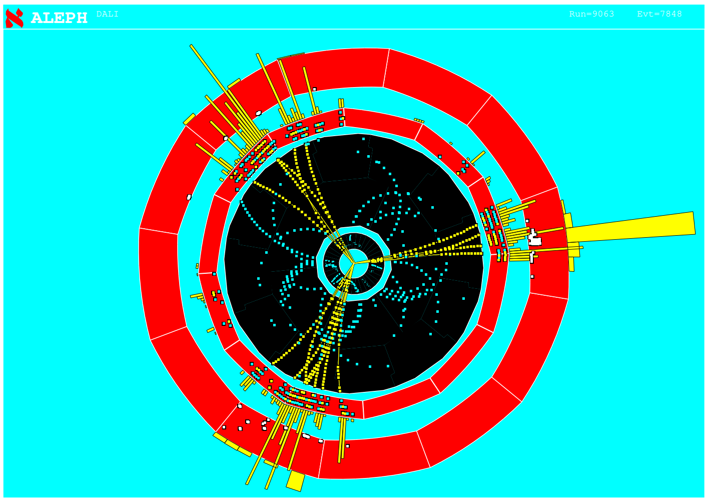

Figure 3: Z-decay into three jets: a quark-antiquark pair and a gluon.

Energy Flow objects were chosen for the studies, thus taking both charged and neutral particles into account. First some selections have to be made, to select only those events that stem from hadronic Z decays.

The charged tracks (of each event) have to fulfil the following criteria:

$$(d0 <= 2) \wedge (z0 <= 5) \wedge (|\cos\theta| <= 0.94) \wedge (ptch >= 0.2) \wedge (Ntpc >= 4)$$

$d0$ and $z0$ measure the distance of the track from the interaction point. $d0$ is the smallest distance in the $x - y$ plane between the projected track and the interaction point (since the magnetic field in the tracking components of the detector is rather uniform, trajectories of charged particles are helices), and $z0$ (z is assumed to be the direction of the beam axis) is the z-coordinate at $x^2 + y^2 = d_0^2$. These first two selection criteria are easily understandable: Particles created in the hadronic Z decay should come from the main interaction point.

Only charged tracks are selected where the angle θ to the beam direction is not too small because measurements in the detector are less precise under small angles. The transverse momentum $ptch$ for each charged track should not be smaller than 200 MeV to neglect particles moving along narrow spiral shaped tracks, and the number of space points $Ntpc$ which were measured in the TPC has to be greater than three.

All the tracks for which at least one of the criteria above fails are locked, which means, that they are not considered any more in the further studies.

Then the event selection takes place. Only events for which

$$(Nchgood >= 5) \wedge (Echgood >= 15) \wedge (|\cos\theta_{thr}| < 0.82)$$

is true, are taken for the following calculations. The requirements are: the total number $Nchgood$ of charged tracks that have passed the track selection must be at least five and their energy sum $Echgood$ must be at least 15 GeV. These cuts are made to reject backgrounds from $\gamma\gamma$ interactions and $Z \rightarrow \tau^+\tau^-$ decays. The last condition concerns the Thrust axis, because the angle θ_{thr} between the Thrust axis and the beam direction should be at least about 35 degrees. Here the Thrust algorithm plays an important role because events where a jet is too close to the beam pipe have to be excluded from the analysis, because in these cases some particles, which also belong to the jet, could be lost into the beam pipe. Therefore the last cut is important to ensure that the event is well contained within the detector.

After the track and the event selection, the intended studies can be performed. The jet rates for ten different values of $Ycut$ are calculated by calling the *DurhamJn* routine and storing the number of jets, which is just the size of the vector returned by *DurhamJn*, for each $Ycut$. Those events, where the jet rate is three and the $Ycut$ value equals 0.01, are analysed further. For this purpose the inter-jet-angles, the multiplicities, which are the numbers of the charged particles in each jet, the jet-momenta and the jet-energies of the three jets are stored in order of decreasing jet-energies. All the important data is written to a so called Ntuple, which is a two-dimensional array, characterised by a fixed number N, specifying the number of entries per event, and by a length, giving the total number of events. This Ntuple is then analysed by plotting different dependencies of the various quantities in PAW. More detailed information about Ntuples can be found in [5]. For help about PAW it is referred to [6].

This chapter is concluded by saying a few words about the amount of data used. It was started with about 11200 events, because until now not more of the data has been converted from ALPHA in the appropriate way to be available in ALPHA++. Due to the track and the event selection this number was reduced to around 8800 selected events. About 2700 three-jet events could be extracted from that for the detailed study of the three-jet events being described in the next chapter.

8 Results and Interpretation

First the n-jet rates were plotted as a function of $Ycut$ to check if the Durham algorithm works correctly and delivers the expected results (Figure 4). The points represent the real data, whereas the lines are the results from a Monte Carlo simulation using JETSET [7]. The n-jet rates for $n = 2, 3, 4, 5$ are shown. The distribution of the jet rates looks exactly like the one in [4]: the smaller the $Ycut$ value, the greater is the probability of having more than two jets, whereas for relatively large $Ycut$ values one will end up most likely with two jets. The three-jet rate predicted by JETSET is significantly higher than the measurement as already discussed in [4].

After that confirmation of the algorithms all the attention was paid to the three-jet events.

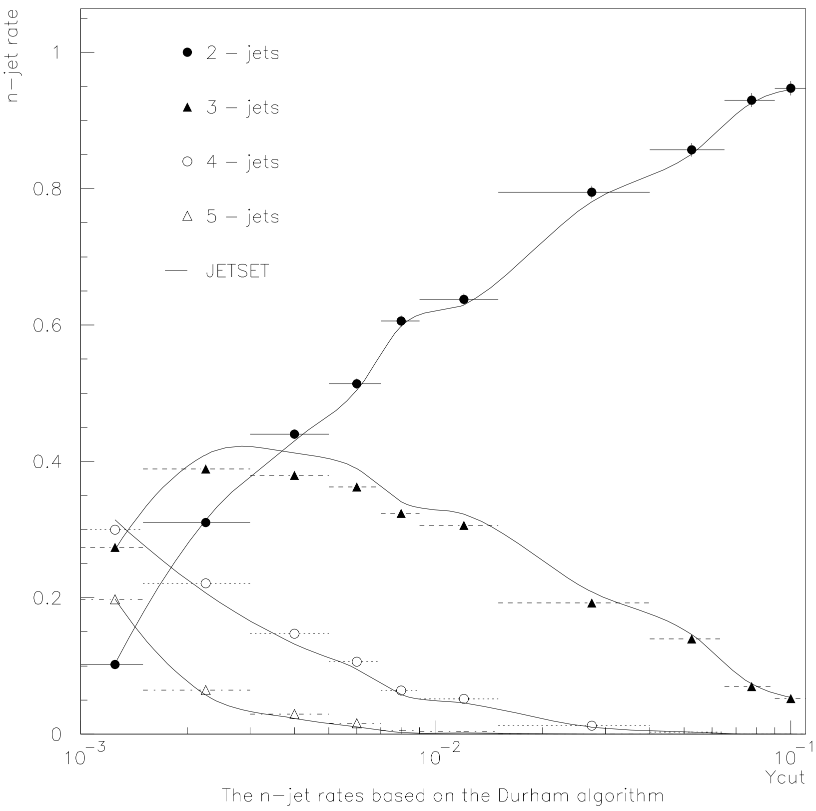

Figure 4: The n-jet rates as a function of $Ycut$.

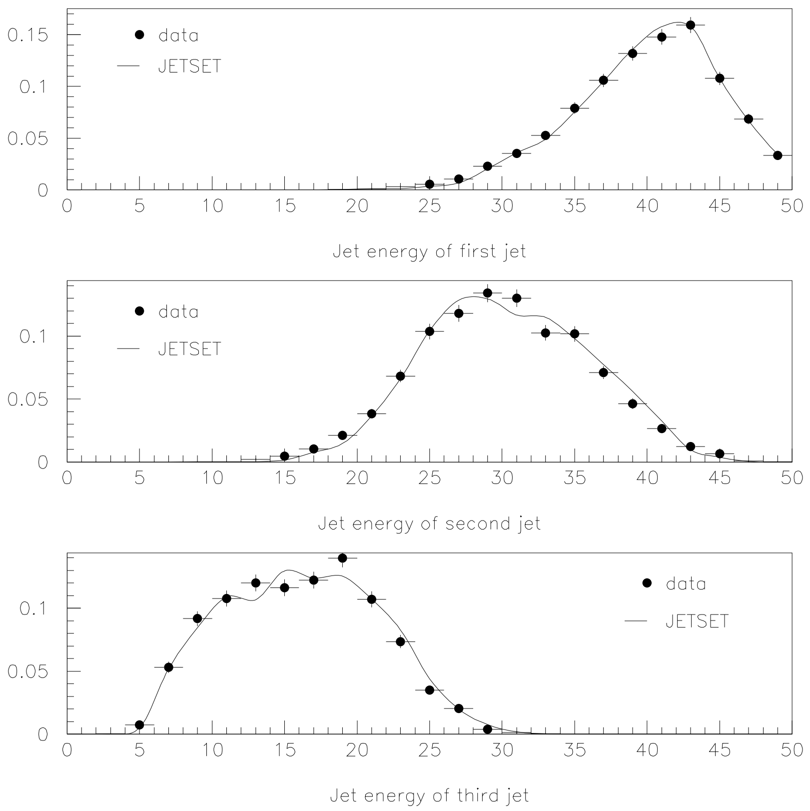

Figure 5: Distribution of the jet-energies in three-jet events.

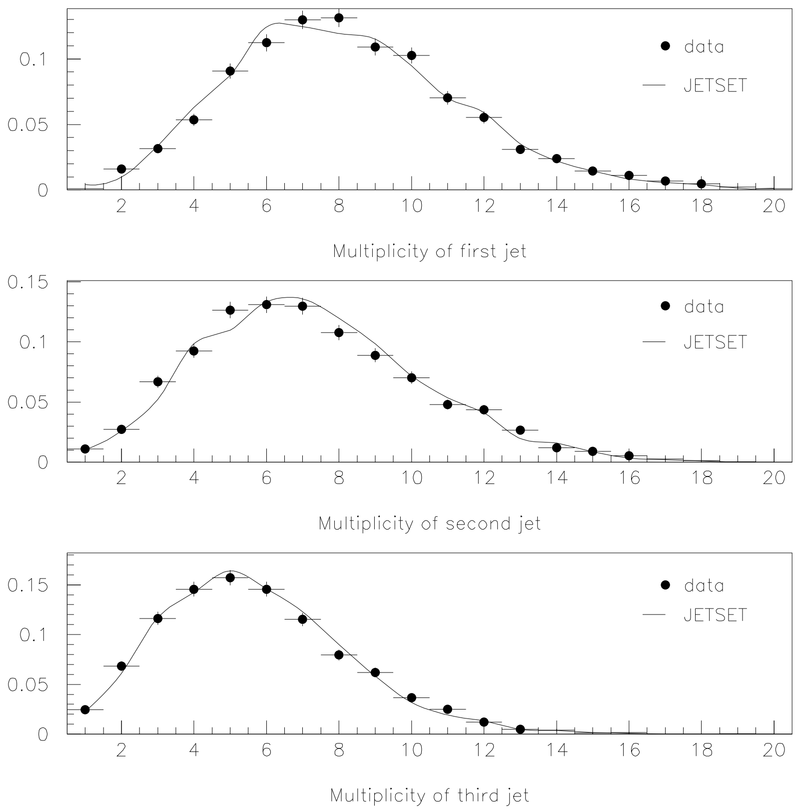

Figure 6: Distribution of the jet-multiplicities in three-jet events.

Figure 5 and 6 show the distributions of the basic attributes of the three-jet events, namely, the energies and the multiplicities of the three jets. As already mentioned the jets were sorted in order of decreasing jet-energies. When it is spoken about the first jet, it is always thought of the most energetic one and the third jet always is the least energetic one in the following considerations.

So how does one know which one of the three jets is the gluon jet? What are the differences between quark-induced and gluon-induced jets? In general the least energetic jet is most likely the gluon jet, because the gluon-bremsstrahlung spectrum peaks at lower energies. It has been assumed in this analysis that the least energetic jet in fact *is* the gluon jet without considering the probability that this is actually the case which is given in first order by [8]

$$P_{3=g} \propto \frac{x_1^2 + x_2^2}{(1-x_1)(1-x_2)} \tag{8}$$

where x_i are the scaled jet energies: $x_i = 2E_i^{jet}/E_{cm}$.

Gluons carry a stronger colour charge than quarks, therefore [9] a difference in their fragmentation is expected. Gluon jets have higher multiplicity than quark jets of the same energy. (In the plots shown here the multiplicity of the gluon jet is of course lower than that of the quark jets, because the gluon jet has lower energy, therefore the multiplicities could not be simply compared.) Gluon jets also have a softer hadron spectrum.

Figure 7 shows the distribution of the mean gluon jet multiplicity. The x-axis is divided into five energy bins (taking all the mean multiplicities lying in the corresponding intervals) and each energy bin is further divided into four $\Delta\theta$ subbins. $\Delta\theta = \theta_{max} - \theta_{min}$ is the difference of the opening angles (inter-jet-angles) to the other two jets. The first $\Delta\theta$ subbin receives all the multiplicities for which $\Delta\theta \in [0, \pi/4]$, for the second $\Delta\theta \in [\pi/4, \pi/2]$ is valid and so on. The vertical dispersion, which shows up at each fixed energy interval, clearly points out that the multiplicity is not only a function of the jet energy, but also depends on the inter-jet-angles.

Therefore one should better plot the multiplicity distribution as a function of the so called topology scale which combines the energies and inter-jet-angles in a suitable way so that only little vertical dispersion is achieved. Two scales seem appropriate to describe the gluon radiation off quarks [8]

$$Q_{gq} = E_g \sin\left(\frac{\theta_{gq}}{2}\right) \qquad\qquad Q_{g\bar{q}} = E_g \sin\left(\frac{\theta_{g\bar{q}}}{2}\right) \tag{9}$$

where E_g is the gluon jet energy, θ_{gq} indicates the angle between the gluon and the quark and $\theta_{g\bar{q}}$ is the angle between the gluon and the antiquark. These two scales can be combined into one effective scale

$$\bar{Q}_g = \sqrt{Q_{gq}Q_{g\bar{q}}} = \sqrt{Q_{max}Q_{min}} \tag{10}$$

with $Q_{max} = E_g \sin(\theta_{max}/2)$ and $Q_{min} = E_g \sin(\theta_{min}/2)$. If the mean gluon jet multiplicities are plotted as a function of $\bar{Q}_g$ [8], one could get a very good fit curve by performing a least-squares fit, assuming a quadratic dependence on $\lg Q$. This plot was not done here because of not having enough statistics. This would have led to yields of only few events in each bin and

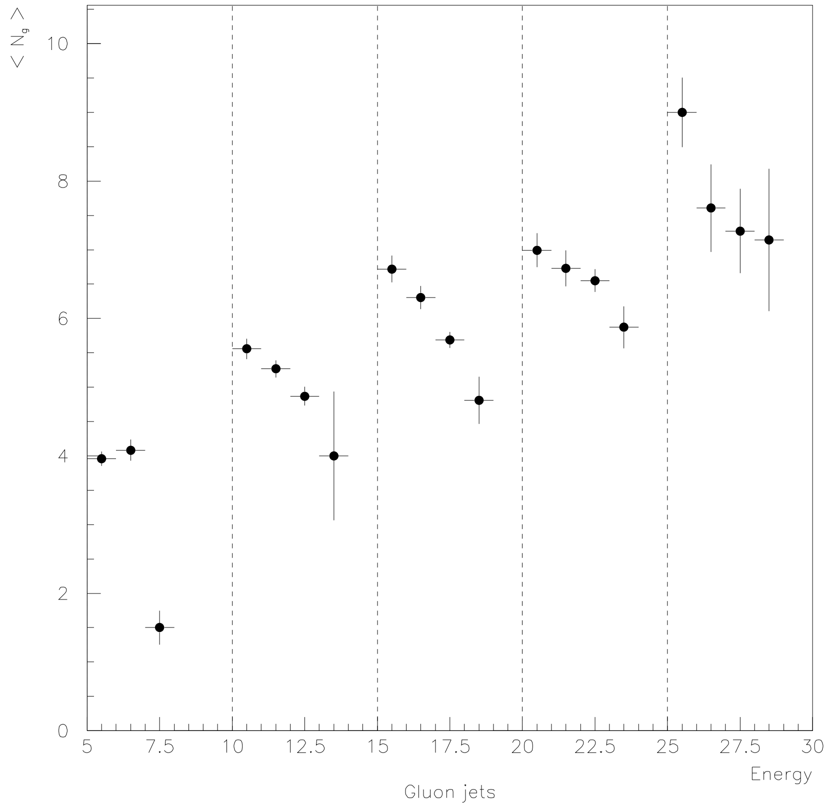

Figure 7: The mean gluon jet charged multiplicity plotted as a function of the jet-energy and $\Delta\theta$.

therefore large errors would have occurred. (As it has already been mentioned, only about 2700 three-jet events were left after all the selections.)

The multiplicity plot as a function of the energy and $\Delta\theta$ was also done for the quark jets, but only the highest energetic quark jet was taken into account for that (Figure 8). Again, the

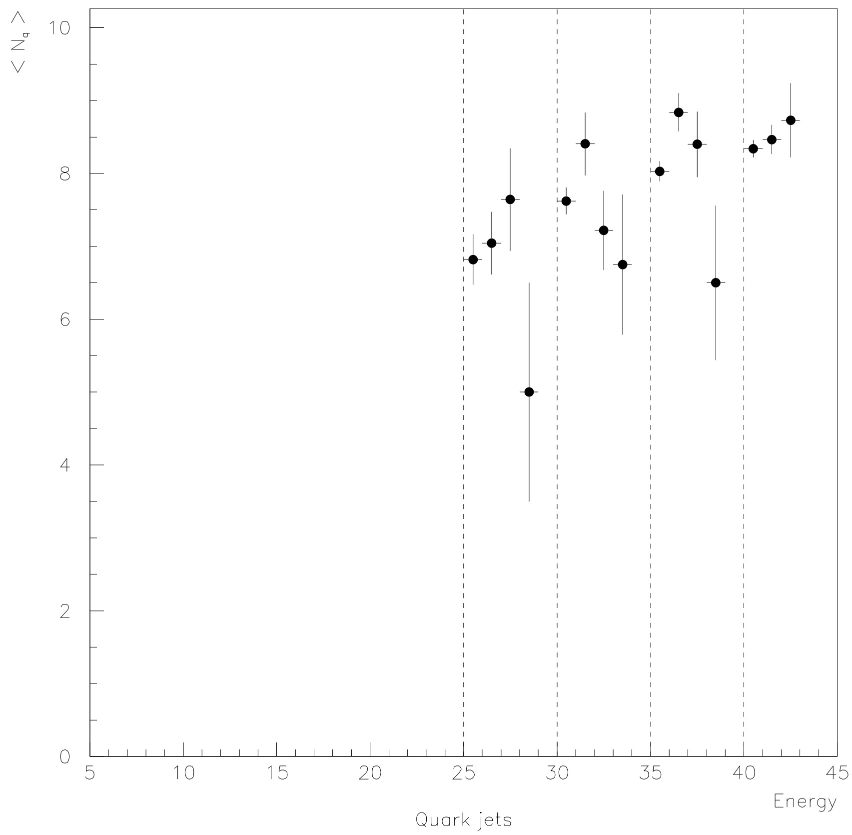

Figure 8: The mean quark jet charged multiplicity plotted as a function of the jet-energy and $\Delta\theta$.

multiplicities cannot be described by the energy dependence alone, as there is also a dependency on the inter-jet-angles. Once more the multiplicity plot as a function of the topology scale which is now described by

$$Q_{qg} = E_q \sin\left(\frac{\theta_{qg}}{2}\right) \approx Q_{min} \tag{11}$$

where E_q is the quark jet energy and θ_{qg} is the angle between the quark and the gluon, was left out. As the angle to the closest jet predominantly gives the desired angle θ_{qg}, Q_{qg} can be substituted by Q_{min} without making a too large error. The multiplicity plot as a function of Q_{min} can be looked up in [8].

Why are the topological scales (10) and (11) the appropriate quantities to describe the multiplicities in a simple way? If it is supposed that the gluon and (anti-)quark colour charges screen each other, a scale that depends on the direction (and not only on the energy) should better be chosen such as (10) and (11). This tends to E_q in case of the quark jets when the quark and the gluon are back-to-back, which means, colour screening is minimal, and vanishes as the quark and the gluon become parallel, which leads to maximal colour screening. So the scale (11) is related to the quark's transverse momentum with respect to the gluon direction. The same argumentation can also be applied to the gluon jets, but in (10) the transverse momenta of both the quark and the antiquark with respect to the gluon direction are considered.

At the end of this chapter the problem of statistics available on the Objectivity database should be discussed. In Figure 8 a large error is indicated for the last data point in the energy interval 35-40 GeV. Therefore the multiplicities for the four different $\Delta\theta$ intervals were plotted separately only for this single energy interval (Figure 9).

In the plot for $\Delta\theta \in [3\pi/4, \pi]$ there are only two entries. This of course explains the large error bar seen at the corresponding data point of Figure 8.

To point out the mutual dependencies of the energies, the multiplicities and the inter-jet-angles once more, Figure 10 shows the $\Delta\theta$ distribution for three different energy intervals: the lower the energy, the more widespread the distribution is.

The same analysis has been performed again using the conventional analysis package ALPHA. The results obtained from the Fortan algorithms were exactly the same, no deviations could be discovered in any of the plots. Therefore the Thrust and Jet finding algorithms which have been implemented in ALPHA++ are working properly and the performed QCD analysis can be really trusted. The only weak point of it is the low statistics, but nevertheless general tendencies can be seen very well.

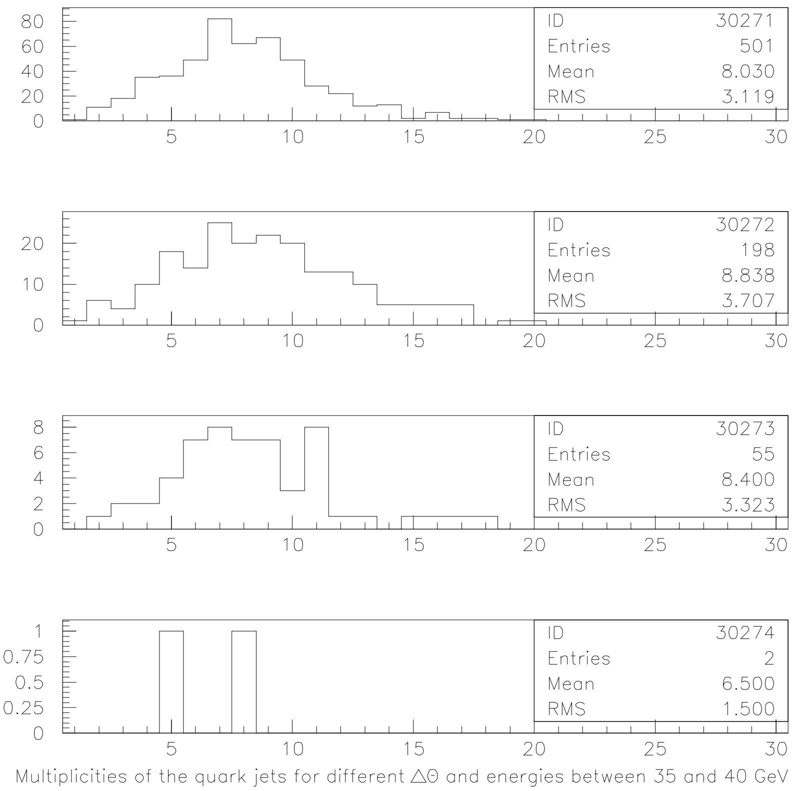

Figure 9: Multiplicities of the quark jets for $\Delta\theta \in [0, \pi/4]$, $[\pi/4, \pi/2]$, $[\pi/2, 3\pi/4]$, $[3\pi/4, \pi]$ and energies of 35-40 GeV.

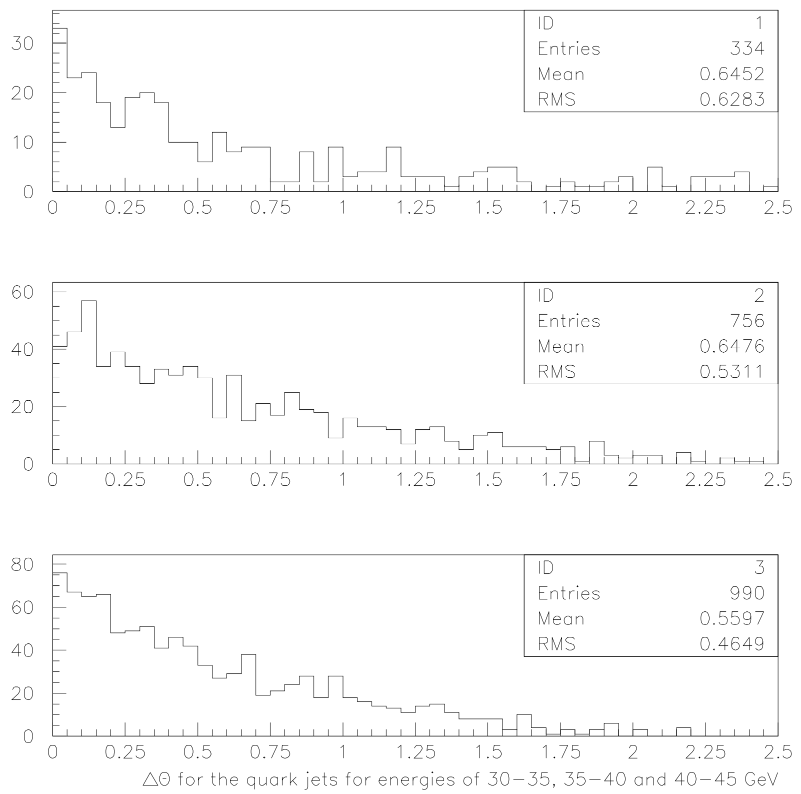

Figure 10: $\Delta\theta$ distribution for the quark jets for three energy intervals: 30-35, 35-40 and 40-45 GeV.

9 Conclusion and Outlook

Two new algorithms, namely the Thrust and the Jet algorithm, have been added to the object oriented ALPHA++ package of the ALEPH experiment. These served as the basis for a QCD analysis, where gluon and quark jets were analysed. It was shown that their multiplicities not only depend on the energies of the jet, but also on the inter-jet-angles. Therefore topological scales are more suitable for describing the multiplicities. Furthermore the general dependencies of the n-jet rates on the cut-off parameter $Ycut$ of the clustering routine were shown.

Still left to be done are performance studies of the algorithms, which certainly are of great interest if a large amount of data should be analysed.

10 Acknowledgements

I wish to thank my supervisor Günther Dissertori who spent quite a lot of time to support me performing this Summer Student project.

References

[1] D. Decamp et al., ALEPH Collaboration, *ALEPH: A Detector for Electron-Positron Annihilations at LEP*, Nucl. Instr. and Meth. **A294** (1990) 121.

[2] D. Buskulic et al., ALEPH Collaboration, *Performance of the ALEPH Detector at LEP*, Nucl. Instr. and Meth. **A360** (1995) 481.

[3] The ALEPH Collaboration and D. Düllmann, ALEPH 99-056, CONF 99-031, Contribution to the EPS-HEP Conference, Tampere, Abstract 15-707, July 1999.
`http://alephwww.cern.ch/~disserto/objty/Welcome.html`

[4] R. Barate et al., ALEPH Collaboration, *Studies of Quantum Chromodynamics with the Aleph Detector*, Phys. Rep. **294**, (1998) 1.

[5] HBOOK:Statistical Analysis and Histogramming, CERN Program Library.

[6] PAW: Physics Analysis Workstation, CERN Program Library.

[7] T. Sjöstrand, *High-Energy Physics Event Generation with PYTHIA 5.7 and JETSET 7.4*, Comput. Phys. Commun. **82** (1994) 74.

[8] R. Barate et al., ALEPH Collaboration, *The Topology Dependence of Charged Particle Multiplicities in Three-Jet Events*, Z. Phys. **C76** (1997) 191.

[9] Z. Fodor, *Differences between Quark and Gluon Jets at LEP*, Phys. Lett. **B263** (1991) 305.

11 Appendix

11.1 Proof of Equations (2) and (3) in the Thrust-Calculation

Proof that $\sum_{i=1}^{k}(\mathbf{p}_i\cdot\mathbf{n}) = \max$ for $\mathbf{n}=\sum_{i=1}^{k}\mathbf{p}_i$ if $\mathbf{p}_i\cdot\mathbf{n}\geq 0\ \forall\ i\in\{1,....,k\}$.

Induction
a) k=2

$$T=\mathbf{p}_1\cdot\mathbf{n}+\mathbf{p}_2\cdot\mathbf{n}=|\mathbf{p}_1|\,|\mathbf{n}|\cos\phi_1+|\mathbf{p}_2|\,|\mathbf{n}|\cos\phi_2$$

where ϕ_1 is the angle between $\mathbf{p}_1$ and $\mathbf{n}$ and ϕ_2 is the angle between $\mathbf{p}_2$ and $\mathbf{n}$. $\phi=\phi_1+\phi_2$ is the angle between $\mathbf{p}_1$ and $\mathbf{p}_2$. ϕ is fixed and

$$\mathbf{p}_i\cdot\mathbf{n}\geq 0 \qquad \forall i\in\{1,2\}$$

is fulfilled! $\Rightarrow$

$$T=|\mathbf{p}_1|\,|\mathbf{n}|\underbrace{\cos\phi_1}_{\geq 0}+|\mathbf{p}_2|\,|\mathbf{n}|\underbrace{\cos(\phi-\phi_1)}_{\geq 0}$$

$$\frac{dT}{d\phi_1}=-|\mathbf{p}_1|\,|\mathbf{n}|\sin\phi_1+|\mathbf{p}_2|\,|\mathbf{n}|\sin(\phi-\phi_1)$$

$$\frac{dT}{d\phi_1}=0 \quad\Leftrightarrow\quad |\mathbf{p}_1|\sin\phi_1=|\mathbf{p}_2|\sin\phi_2 \quad\Leftrightarrow\quad \mathbf{n}=\mathbf{p}_1+\mathbf{p}_2$$

(the last connection can be easily worked out by drawing a parallelogram with the side lengths $\mathbf{p}_1$ and $\mathbf{p}_2$ and the diagonal $\mathbf{n}$)

b) Assumption: $T_m=\sum_{i=1}^{m}(\mathbf{p}_i\cdot\mathbf{n})=\max$ for $\mathbf{n}=\sum_{i=1}^{m}\mathbf{p}_i$.
To be shown: $T_{m+1}=\sum_{i=1}^{m+1}(\mathbf{p}_i\cdot\mathbf{n}')=\max$ for $\mathbf{n}'=\sum_{i=1}^{m+1}\mathbf{p}_i$
(always: $\mathbf{p}_i\cdot\mathbf{n}\geq 0$, $\mathbf{p}_i\cdot\mathbf{n}'\geq 0$)

$$T_{m+1}=\sum_{i=1}^{m+1}(\mathbf{p}_i\cdot\mathbf{n}')=\sum_{i=1}^{m+1}[(\mathbf{p}_i\cdot\mathbf{n})+(\mathbf{p}_i\cdot\tilde{\mathbf{n}})]=$$

$$=\underbrace{\sum_{i=1}^{m}(\mathbf{p}_i\cdot\mathbf{n})}_{T_m}+\mathbf{p}_{m+1}\cdot\mathbf{n}+\sum_{i=1}^{m+1}(\mathbf{p}_i\cdot\tilde{\mathbf{n}})$$

All terms are positive $\Rightarrow$ Maximum of T_{m+1} is acquired when all single terms acquire maximum.
$\Rightarrow$

$$T_{m+1}\leq T_m+\mathbf{p}_{m+1}\cdot\mathbf{n}+\underbrace{\left(\sum_{i=1}^{m}\mathbf{p}_i\right)}_{=\mathbf{n}}\cdot\tilde{\mathbf{n}}+\mathbf{p}_{m+1}\cdot\tilde{\mathbf{n}}=$$

$$= T_m + \big(\mathbf{p}_{m+1} + \tilde{\mathbf{n}}\big) \cdot \mathbf{n} + \underbrace{\mathbf{p}_{m+1} \cdot \tilde{\mathbf{n}}}_{\textit{max for } \tilde{\mathbf{n}}=\mathbf{p}_{m+1}}$$

Therefore $\mathbf{n}'$ can be written as

$$\mathbf{n}' = \mathbf{n} + \tilde{\mathbf{n}} = \sum_{i=1}^{m+1} \mathbf{p}_i$$

To continue the calculation of T_{m+1}:

$$\leq T_m + 2\mathbf{p}_{m+1} \cdot \mathbf{n} + \mathbf{p}_{m+1}^2 = T_m + \mathbf{p}_{m+1} \cdot \mathbf{n} + \mathbf{p}_{m+1} \cdot \mathbf{n}'$$

$$= \left[\left(\sum_{i=1}^{m} \mathbf{p}_i\right) + \mathbf{p}_{m+1}\right] \cdot \mathbf{n} + \mathbf{p}_{m+1} \cdot \mathbf{n}' = \mathbf{n}' \cdot \mathbf{n} + \mathbf{p}_{m+1} \cdot \mathbf{n}' =$$

$$= \mathbf{n}' \cdot \big(\mathbf{n} + \mathbf{p}_{m+1}\big) = \sum_{i=1}^{m+1} \mathbf{p}_i \cdot \left(\sum_{j=1}^{m+1} \mathbf{p}_j\right)$$

q. e. d.

11.2 AlObject.h

Header file of the abstract interface *AlObject*:
Author : G. Bagliesi

```
#ifndef _ALEPHOBJECT_H_
#define _ALEPHOBJECT_H_

// need CLHEP stuff
#include ”CLHEP/Vector/LorentzVector.h”

// list all possible object types
enum ALEPHTYPE {OBJECT,ALPHA_QVEC,ALPHA_QVRT,TRACK,EFLOW,ECAL,HCAL,
GAMPEC,MAINVERTEX,GENVERTEX,ALJET,ALTHRUST};

// the class
class AlObject {


public:
   virtual float QP() = 0;
   virtual float QX() = 0;
   virtual float QY() = 0;
   virtual float QZ() = 0;
   virtual float QE() = 0;
   virtual float QM() = 0;
   virtual HepLorentzVector A4V() = 0;
   virtual void setA4V(float x, float y, float z, float e) = 0;
   virtual void setA4V(HepLorentzVector vec) = 0;
   virtual float QCH() = 0;
   virtual int isLocked() = 0;
   virtual void Lock() = 0;
   virtual ALEPHTYPE TYPE(){return OBJECT; }
   virtual  AlObject(){}


};

#endif
```

11.3 QvecBase.h

Header file of the *QvecBase* class:
Authors: G. Bagliesi, M. Hörndl

```
#ifndef _QVECBASE_H_
#define _QVECBASE_H_

#include "AlObject.h"
#include "AlphaCopy.h" //contains QVEC, QVEC_ALPHA, QVRT, ..

// the class, inherits from AlObject
class QvecBase : public AlObject {


public:
   // constructor
   QvecBase();
   //copy constructor
   QvecBase(const QvecBase & origin);
   // set the pointer to the QVEC object
   virtual void SetPointer(QVEC* qvec);
   // set the Lorentz Vector
   virtual void setA4V(float x, float y, float z, float e);
   virtual void setA4V(HepLorentzVector vec){_A4V = vec;}
   // return the Lorentzvector
   virtual HepLorentzVector A4V();
   // set the charge
   virtual void setQCH(float qch){_qch = qch;}
   // get the type
   virtual ALEPHTYPE TYPE(){return OBJECT;}
   // copies of the ALPHA utility functions (see ALPHA manual)
   virtual float sign(float k);
   virtual float QP(); // magnitude of three momentum
   virtual float QX(); // x-component of three momentum
   virtual float QY(); // y-component of three momentum
   virtual float QZ(); // z-component of three momentum
   virtual float QE(); // energy
   virtual float QM(); // mass
   virtual float QCH(); // charge
   virtual float QCT(); // cos (polar angle)
   virtual float QPT(); // φ = azimuth
   virtual float QBETA(); // beta
   virtual float QGAMMA(); // gamma
   virtual float QMSQ2(QvecBase j); // (inv. mass)^2 of current particle and j
   virtual float QM2(QvecBase j); // inv. mass of current particle and j
```

```
    virtual float QDMSQ(QvecBase j); // mass^2 of 4-mom difference
    virtual float QPPAR(QvecBase j); // mom. comp. of current part. parallel to j
    virtual float QPPER(QvecBase j); // mom. comp. of current part. perpend. to j
    virtual float QDOT3(QvecBase j); // 3 dim cross product
    virtual float QDOT4(QvecBase j); // 4 dim cross product
    virtual float QCOSA(QvecBase j); // cos (angle between current part. and j)
    virtual float QDECA2(QvecBase j); // cos (decay angle)
    // add two objects
    virtual QvecBase add(QvecBase j);
    // get decay angle, same result as QDECA2, but using CLHEP
    virtual HepDouble DECAY_ANG(QvecBase j);
    // the locking
    virtual void Lock(); // lock the track
    virtual int isLocked(); // =0 if not locked, else locked
    virtual float QDB(); // distance of closest aprroach to beam axis
    virtual float QZB(); // z-coord. of track point where QDB is measure
    virtual float QDBS2(); // error on QDB
    virtual float QZBS2(); // error on QZB
    virtual float QBC2(); // chi^2 due to QDB and QZB


protected:
   HepLorentzVector _A4V; //this is a data member containing the instance
                          // of the four vector
   QVEC* qvec; // pointer to QVEC
   int _locked;    // lock status
   float _qch;     // charge


};

#endif
```

11.4 QvecBase.cpp

Implementation file of the *QvecBase* class:
Authors: G. Bagliesi, M. Hörndl

```
#include <stdlib.h>
#include <vector.h>
#include <math.h>
#include "QvecBase.h"

int QvecBase::NERR=0;

//copy constructor
QvecBase::QvecBase(const QvecBase& origin)

{
 _A4V = origin._A4V;
 _locked = origin._locked;
 _qch = origin._qch;
 qvec = origin.qvec;
}

float QvecBase::sign(float k) {return k/fabs(k);}

// return the Lorentz vector
HepLorentzVector QvecBase::A4V() { return _A4V; }

// set the Lorentz vector
void QvecBase::setA4V(float x, float y, float z, float e)

{
 _A4V.setPx(x);
 _A4V.setPy(y);
 _A4V.setPz(z);
 _A4V.setE(e);
}

// add two objects
QvecBase QvecBase::add(QvecBase j)

{
 HepLorentzVector A4Temp = A4V() + j.A4V();
 QvecBase Temp;
 Temp.setA4V(A4Temp);
 Temp.setQCH(QCH() + j.QCH());
 return Temp;
}
```

```
// set the pointer to the memory
void QvecBase::SetPointer(QVEC* pointer)
{
 qvec = pointer;
 setA4V(qvec->QX(), qvec->QY(), qvec->QZ(), qvec->QE());
 _qch = qvec->QCH();
 _locked = 0;
}
// appreviations for standard ALPHA functions
float QvecBase::QCH(){return _qch;}
float QvecBase::QX() {return A4V().px();}
float QvecBase::QY() {return A4V().py();}
float QvecBase::QZ() {return A4V().pz();}
float QvecBase::QP() {return A4V().rho();}
float QvecBase::QE() {return A4V().e();}
float QvecBase::QM() {return A4V().m();}
float QvecBase::QCT() { return A4V().cosTheta();}
float QvecBase::QPH() { return A4V().phi(); }
float QvecBase::QPT() { return A4V().perp(); }
float QvecBase::QBETA() { return A4V().rho()/A4V().e();}
float QvecBase::QGAMMA() { return 1./sqrt((1.-QBETA())*(1.+QBETA())); }

float QvecBase::QMSQ2(QvecBase j)
{
 HepLorentzVector A4Temp = A4V() + j.A4V();
 return A4Temp.m2();
}
float QvecBase::QM2(QvecBase j)
{
 HepLorentzVector A4Temp = A4V() + j.A4V();
 return A4Temp.m();
}
float QvecBase::QDMSQ(QvecBase j)
{
 HepLorentzVector A4Temp = A4V() - j.A4V();
 return A4Temp.m2();
}
float QvecBase::QPPAR(QvecBase j) { return QDOT3(j)/j.A4V().rho();}
float QvecBase::QPPER(QvecBase j) { return A4V().perp(j.A4V()); }
float QvecBase::QDOT3(QvecBase j) { return Hep3Vector(A4V()).dot(j.A4V());}
float QvecBase::QDOT4(QvecBase j) { return A4V().dot(j.A4V());}
float QvecBase::QCOSA(QvecBase j)
```

```
{
 return cos(A4V().angle(Hep3Vector(j.A4V())));
}

float QvecBase::QDECA2(QvecBase j)

{
 double P_1[4], P_2[4];
 P_1[0] = (double) A4V().px();
 P_1[1] = (double) A4V().py();
 P_1[2] = (double) A4V().pz();
 P_1[3] = (double) A4V().e();
 double PMAG = sqrt(pow(P_1[0],2) + pow(P_1[1],2) + pow(P_1[2],2));
 P_2[0] = (double) j.A4V().px();
 P_2[1] = (double) j.A4V().py();
 P_2[2] = (double) j.A4V().pz();
 P_2[3] = (double) j.A4V().e();
 double PP = sqrt( pow((P_1[0]+P_2[0]),2) + pow((P_1[1]+P_2[1]),2)
                                          + pow((P_1[2]+P_2[2]),2));
 double BETA = PP / ( P_1[3]+P_2[3] );
 double PPAR = ( P_1[0] *(P_1[0]+P_2[0]) + P_1[1] *(P_1[1]+P_2[1])
                                         + P_1[2] *(P_1[2]+P_2[2]) )/PP;
 PP = PPAR - BETA * P_1[3];
 return (float) (PP / sqrt(pow(PP,2.) + (PMAG-PPAR)*(PMAG+PPAR)*(1.-BETA)*(1.+BETA)));
}

HepDouble QvecBase::DECAY_ANG(QvecBase j)

{
 QvecBase temp= (*this).add(j);
 HepLorentzVector v = temp.A4V();
 HepLorentzVector v1 = A4V();
 v1.boost(-v.boostVector());
 return cos(v.angle(Hep3Vector(v1)));
}

// the locking
void QvecBase::Lock() { _locked = 1 ; }
int QvecBase::isLocked() { return _locked; }

float QvecBase::QDB() { return qvec->DB(); }
float QvecBase::QZB() { return qvec->ZB(); }
float QvecBase::QDBS2() { return qvec->SD(); }
float QvecBase::QZBS2() { return qvec->SZ(); }
float QvecBase::QBC2() { return qvec->CB(); }
```

11.5 AlThrust.h

Header file of the *AlThrust* class:
Author: M. Hörndl

```
#ifndef _ALTHRUST_H_
#define _ALTHRUST_H_

#include "QvecBase.h"

class AlThrust: public QvecBase {

public:
   //default constructor
   AlThrust();
   ALEPHTYPE TYPE(){return ALTHRUST;}
   Hep3Vector getThrustDirection();
   float getThrustValue();

};

#endif
```

11.6 AlThrust.cpp

Implementation file of the *AlThrust* class:
Author: M. Hörndl

```
#include "AlThrust.h"
// default constructor : has to do something,
// since it is not created out of qvec
AlThrust::AlThrust()

{
 _A4V.setE( 0.);
 _A4V.setPx(0.);
 _A4V.setPy(0.);
 _A4V.setPz(0.);
 _qch = 0.;
 _locked = 0;
 qvec = NULL;
}

Hep3Vector AlThrust::getThrustDirection()
{ return Hep3Vector(A4V()); }

float AlThrust::getThrustValue()
{ return A4V().e(); }
```

11.7 AlJet.h

Header file of the *AlJet* class:
Author: M. Hörndl

```
#ifndef _ALJET_H_
#define _ALJET_H_

#include <ospace/config.h>
#include <ospace/stl/vector.h>
#include "QvecBase.h"

class AlJet: public QvecBase {

public:
   // default constructor
   AlJet();
   // copy constructor
   AlJet(const AlJet& oldAj);
   ALEPHTYPE TYPE(){return ALJET;}
   vector<AlObject*>& getObjects();
   void addObject(AlObject* a);
   void setScheme(int s);
   int getScheme();
private:
  vector<AlObject*> _objects; // objects of which jet was built
  int _scheme; // recombination scheme

};

#endif
```

11.8 AlJet.cpp

Implementation file of the *AlJet* class:
Author: M. Hörndl

```
#include ”AlJet.h”
// default constructor : has to do something,
// since it is not created out of qvec
AlJet::AlJet()

{
 _A4V.setE( 0.);
 _A4V.setPx(0.);
 _A4V.setPy(0.);
 _A4V.setPz(0.);
 _qch = 0.;
 _locked = 0;
 qvec = NULL;
 _objects.resize(0);
 _scheme = 0;
}

// copy constructor : NOTE: THE VECTOR OF POINTERS
// TO ITS OBJECTS IS JUST COPIED!! COULD BE DANGEROUS...

AlJet::AlJet(const AlJet& oldAl)

{
 _objects.resize(oldAl._objects.size());
 for (int i = 0; i ¡ oldAl._objects.size(); i++)
   {
    _objects[i] = oldAl._objects[i];
   }
 _scheme = oldAl._scheme;
}

vector<AlObject*>& AlJet::getObjects()
{ return _objects; }

void AlJet::addObject(AlObject* a)
{ _objects.push_back(a); }

void AlJet::setScheme(int s)
{ _scheme = s; }

int AlJet::getScheme()
{ return _scheme; }
```

11.9 AlephAlgo.h

Header file of the *AlephAlgo* class:
Author: M. Hörndl

```
#ifndef _ALEPHALGO_H_
#define _ALEPHALGO_H_

#include ”AlphaBanks.h”
#include ”AlJet.h”
#include ”AlThrust.h”
#include ”AlObject.h”

#include <ospace/config.h>
#include <ospace/stl/vector.h>
#include ”CLHEP/Matrix/Matrix.h”

class AlephAlgo {

public:
  AlephAlgo();
  AlephAlgo(AlphaBanks* alpha);
  void setAlphaBanks(AlphaBanks* alpha);
  AlphaBanks* getAlphaBanks();

  //jet algorithms

  // There are two functions that are slightly different in their parameters' list.
  // The vector<AlJet> DurhamJn(int Scheme, float Ycut, vector<AlObject*>Pext) routine
  // performs the Jet algorithm with all the objects of Pext
  // whereas vector<AlJet> DurhamJn(int Scheme, float Ycut, ALEPHTYPE type)
  // only takes not-locked objects of the appropriate type for doing the clustering routine
  // (=calling the other routine).

  vector<AlJet> DurhamJn(int Scheme, float Ycut, ALEPHTYPE type);
  vector<AlJet> DurhamJn(int Scheme, float Ycut, vector<AlObject*> Pext);

  // The functions DurhamYn which are organized in the same way as the DurhamJn routines
  // calculate the Ycut values for different numbers of jets and return them in the form
  // of a vector of float values. For this purpose the clustering algorithm has to be changed slightly.
  // Instead of stopping if the condition y_{imin jmin} > Ycut is fulfilled you now continue
  // the algorithm until only two jets are left and store the different Ycuts
  // you get for different numbers of jets in an vector.

  vector<float> DurhamYn(int Scheme, ALEPHTYPE type);
  vector<float> DurhamYn(int Scheme, vector<AlObject*> Pext);
```

```
  //thrust algorithms

  // Again there are two functions for the Thrust routine.
  // The AlThrust AThrust(vector<AlObject*>Pext) routine performs the algorithm
  // with all the AlObjects to which the vector Pext has pointers, whereas the
  // AlThrust AThrust(ALEPHTYPE type) routine chooses the not locked objects
  // of the appropriate type and then does the thrust calculation with these selected objects
  // (by calling the other routine).

  AlThrust AThrust(ALEPHTYPE type);
  AlThrust AThrust(vector<AlObject*> Pext);

private:
  AlphaBanks* _alpha;

};

#endif
```

11.10 AlephAlgo.cpp

Implementation file of the *AlephAlgo* class:
Author: M. Hörndl

```
float max(float i, float j) { return i > j ? i : j; }
float max(double i, float j) { return i > j ? i : j; }
float min(float i, float j) { return i < j ? i : j; }
float min(double i, float j) { return i < j ? i : j; }

#include ”AlephAlgo.h”
#include <fstream.h>
extern ofstream fout;
//default constructor
AlephAlgo::AlephAlgo()

{ fout << ”Error ” << endl;
  exit(1); }

//constructor where AlphaBanks is set
AlephAlgo::AlephAlgo(AlphaBanks* alpha)
{ _alpha = alpha; }

void AlephAlgo::setAlphaBanks(AlphaBanks* alpha)
{ _alpha = alpha; }

AlphaBanks* AlephAlgo::getAlphaBanks()
{ return _alpha; }

////////////////////////////////////////////////////////////////////////
//jet algorithms
////////////////////////////////////////////////////////////////////////

vector<AlJet> AlephAlgo::DurhamJn(int Scheme, float Ycut, ALEPHTYPE type)

{ /*
  Routine to calculate number of jet and jet momenta for a given Ycut
  Input: float Ycut     Ycut value
         int Scheme     Recombination Scheme
                             Scheme = 0 : E0 - Scheme
                             Scheme = 1 : E - Scheme
  vector<AlObject*> selObjs    Vector of Pointer to AlObjects read out
                               from ObjectV() according to type
  Output: vector<AlJet> jet    Vector of Jets
  */
```

```
  AlphaBanks* b = getAlphaBanks();
  if (((type != TRACK) && (type != EFLOW)) && ((type != GAMPEC) && (type != ALJET)))
    {
      cout << " DurhamJn : WARNING : type " << type << " not supported! " << endl;
      exit(1);
    }
  vector<AlObject*> selObjs;
  // do the object selection
  for(vector<AlObject*>::iterator iobj=b->ObjectV().begin(); iobj < b->ObjectV().end(); iobj++)
    {
      if ( ((*iobj)->TYPE() == type) && ((*iobj)->isLocked() == 0) )
        {
          selObjs.push_back(*iobj);
        }
    }
  //call the appropriate cluster routine
  return DurhamJn(Scheme, Ycut, selObjs);
}
//////////////////////////////////////////////////////////////////////
vector<AlJet> AlephAlgo::DurhamJn(int Scheme, float Ycut, vector<AlObject*> Pext)
{ /*
  Routine to calculate number of jet and jet momenta for a given Ycut and
  a given vector of pointers to AlObjects
  Input: float Ycut    Ycut value
         int Scheme    Recombination Scheme
                             Scheme = 0 : E0 - Scheme
                             Scheme = 1 : E - Scheme
  int Np    Number of particles
  vector<AlObject*> P    Vector of Pointer to AlObjects
  Output: vector<AlJet> jets    Vector of Jets
  */

  int Np = Pext.size();
  AlphaBanks* b= getAlphaBanks();

  int Imin, Jmin, IminC, Njet;
  float Ymin, charge;

  vector<AlObject*> P; // copy of Pext, so that changes of P do not influence Pext

  vector<int> Iflag;
  // Copy momenta to internal vector P, initialize
  // rescale all momenta to zero mass in case E0
  float Evis = 0.;
```

```
for(int i=0; i<Np; i++)
  {
    QvecBase* tmp1 = (QvecBase*)(Pext[i]);
    AlObject* tmp = new QvecBase(*tmp1);
    P.push_back(tmp);
    Iflag.push_back(0);
    float e = P[i]->A4V().e();
    Evis = Evis + e;
    float PnClus = Hep3Vector(P[i]->A4V()).mag();
    if (Scheme == 0)
      {
        float px = P[i]->A4V().px()*P[i]->A4V().e()/(max(pow(10,-7),PnClus));
        float py = P[i]->A4V().py()*P[i]->A4V().e()/(max(pow(10,-7),PnClus));
        float pz = P[i]->A4V().pz()*P[i]->A4V().e()/(max(pow(10,-7),PnClus));
        P[i]->setA4V(px,py,pz,e);
      }
  }

HepMatrix Y(Np,Np);
HepMatrix StoreObj(Np,Np);

vector<AlJet> jets(Np);

for(int i=1; i<=Np; i++)
  for(int j=1; j<=Np; j++)
    StoreObj(i,j)=0;


// Check if # of particles is o.k.
if (Np < 2)
  {
    jets.resize(0);
    fout << "there is only one particle, therefore DurhamJn cannot be performed" << endl;
    return jets;
  }

Njet = Np;
IminC = -1;
Ymin = 0.;
// loop over # of jets
while ((Njet >= 1) && (Ymin <= Ycut))
   {
     Ymin = 100000.;

     // loop over combination of particles
```

```
for(int i=0; i<Np-1; i++)
  {
    if (Iflag[i] == 1) continue;
    for (int j=i+1; j<Np; j++)
      {
        if (Iflag[j] == 1) continue;
        // if nothing has changed for i and j
        if (((i != IminC) && (j != IminC)) && (Njet != Np))
          {
            if (Y(i+1,j+1) < Ymin)
              {
                Ymin = Y(i+1,j+1);
                Imin = i;
                Jmin = j;
              }
            else {}
          }
        else
          {
            //else compute metric now
            Y(i+1,j+1) = 2.*min(pow(P[i]->A4V().e()/Evis,2),pow(P[j]->A4V().e()/Evis,2))*
                           (1. - cos(P[i]->A4V().angle(Hep3Vector(P[j]->A4V()))));
            if (Y(i+1,j+1) < Ymin)
              {
                Ymin = Y(i+1,j+1);
                Imin = i;
                Jmin = j;
              }
          } // of else
      } // for int j ....
  } // for int i ....
// compare Ymin with Ycut
if (Ymin > Ycut)
  {
    //fill vector of jets
    int i = 0;
    int iclus = 0;
    while (i < Np)
       {
         if (Iflag[i] == 0)
           {
             jets[iclus].setA4V(P[i]->A4V());
             jets[iclus].setScheme(Scheme);
             charge = P[i]->QCH();
             jets[iclus].addObject(Pext[i]);
             for(int j=0; j< Np; j++)
```

```
                { if (StoreObj(i+1,j+1) == 1)
                      {
                        jets[iclus].addObject(Pext[j]);
                        charge = charge + P[j]->QCH();
                      }
                }
              jets[iclus].setQCH(charge);
              iclus ++;
            }
          i++;
        }
    jets.resize(iclus);
  }
else
  {
    // recombine according to E-Scheme
    if (Scheme == 1)
      {
        P[Imin]->setA4V(P[Imin]->A4V() + P[Jmin]->A4V());
      }

    // recombine according to E0-Scheme
    if (Scheme == 0)
      {
        float e = P[Imin]->A4V().e() + P[Jmin]->A4V().e();
        float P2i = Hep3Vector(P[Imin]->A4V()+P[Jmin]->A4V()).mag();
        if (P2i <= pow(10,-14))
          {
            cout << " Error: Division by zero! " << endl;
            exit(1);
          }
        float px = e * (P[Imin]->A4V().px() + P[Jmin]->A4V().px())/P2i;
        float py = e * (P[Imin]->A4V().py() + P[Jmin]->A4V().py())/P2i;
        float pz = e * (P[Imin]->A4V().pz() + P[Jmin]->A4V().pz())/P2i;
        P[Imin]->setA4V(px,py,pz,e);
      }

    // lock the other track
    Iflag[Jmin] = 1;
    // Store Jmin and all objnumbers of Jmin in StoreObj
    StoreObj(Imin+1,Jmin+1)=1;
    for(int j=0; j < Np; j++)
      {if (StoreObj(Jmin+1,j+1) == 1) StoreObj(Imin+1,j+1) = 1; }
    IminC = Imin;
    Njet --;
  } // else
```

```
    } //while

  // fill the jets into AlObject* vector ObjectV
  for (vector<AlJet>::iterator ije=jets.begin();
                               ije < jets.end(); ije++)
    {
      b->addObject(ije);
    }

  // delete all the copies of the AlephObjects
  for (vector<AlObject*>::iterator io = P.begin(); io ¡ P.end(); io++ )
    {
      AlObject* tmp = *io;
      // set the pointers in the vector to NULL, to be sure
      // that the vector doesn't suffer from deleting the memory
      *io = NULL;
      delete tmp;
    }
  // give the jets vector back
  return jets;
}
///////////////////////////////////////////////////////////////////////

vector<float> AlephAlgo::DurhamYn(int Scheme, ALEPHTYPE type)

{
 /*
 Routine to calculate Yn, where Yn is minimum metric in case
 of a n-jet event

 vector<AlObject*> selObjs    Vector of AlObject* read out from ObjectV()
                              according to the type
 Input: int Scheme    Recombination Scheme
                          Scheme = 0 : E0 - Scheme
                          Scheme = 1 : E - Scheme
 Output: vector<float> Yn    Vector of Yn
 */

 AlphaBanks* b = getAlphaBanks();

 if (((type != TRACK) && (type != EFLOW)) && ((type != GAMPEC) && (type != ALJET)))
   {
     cout << " DurhamYn : WARNING : type " << type << " not supported! " << endl;
     exit(1);
   }
 vector<AlObject*> selObjs;
```

```
  // select the objects
  for(vector<AlObject*>::iterator iobj=b->ObjectV().begin(); iobj < b->ObjectV().end(); iobj++)
    {
      if ( ((*iobj)->TYPE() == type) && ((*iobj)->isLocked() == 0) )
        {
          selObjs.push_back(*iobj);
        }
    }
  // call the appropriate clustering routine
  return DurhamYn(Scheme,selObjs);
}
//////////////////////////////////////////////////////////////////////////

vector<float> AlephAlgo::DurhamYn(int Scheme, vector<AlObject*> Pext)
{
  /*
  Routine to calculate Yn, where Yn is minimum metric in case
  of a n-jet event

  vector<AlObject*> Pext     Vector of pointers to AlObjects, which this procedure
                             gets from outside
  int Np    Number of external particles
  Input: int Scheme    Recombination Scheme
                             Scheme = 0 : E0 - Scheme
                             Scheme = 1 : E - Scheme
  Output: vector<float> Yn    Vector of Yn
  */

  int Np=Pext.size();
  vector<HepLorentzVector> P;

  int Njets, Imin, Jmin, IminC;
  float Ymin;

  vector<int> Iflag;

  // Copy momenta to internal HepLorentzVector P, initialize
  // rescale all momenta to zero mass in case E0
  float Evis = 0.;
  for(int i=0; i<Np; i++)
    {
      P.push_back(Pext[i]->A4V());
      Iflag.push_back(0);
      Evis = Evis + P[i].e();
      float PnClus = Hep3Vector(P[i]).mag();
```

```
    if (Scheme == 0)
      {
        P[i].setPx(P[i].px()*P[i].e()/(max(pow(10,-7),PnClus)));
        P[i].setPy(P[i].py()*P[i].e()/(max(pow(10,-7),PnClus)));
        P[i].setPz(P[i].pz()*P[i].e()/(max(pow(10,-7),PnClus)));
      }
  }
vector<float> Yn(Np-2);
HepMatrix Y(Np,Np);

// Check if # of particles is o.k.
if (Np < 3)
  {
    Yn.resize(0);
    fout << "there are only two tracks, therefore DurhamYn cannot be performed" << endl;
    return Yn;
  }

Njets = Np;
IminC = -1;
//loop over # of jets

while (Njets >= 3)
  {
    Ymin = 100000.;

    //loop over combination of particles

    for(int i=0; i < Np-1; i++)
      { if (Iflag[i] == 1) continue;
        for (int j=i+1; j < Np; j++)
          { if (Iflag[j] == 1) continue;
            // if nothing has changed for i and j
            if (((i != IminC) && (j != IminC)) && (Njets != Np))
              {
                if (Y(i+1,j+1) < Ymin)
                  {
                    Ymin = Y(i+1,j+1);
                    Imin = i;
                    Jmin = j;
                  }
                else {}
              }
            else
              {
                //else compute metric now
```

```
                Y(i+1,j+1) = 2.*min(pow(P[i].e()/Evis,2),pow(P[j].e()/Evis,2))*
                             (1. - cos(P[i].angle(Hep3Vector(P[j]))));
                if (Y(i+1,j+1) < Ymin)
                  {
                    Ymin = Y(i+1,j+1);
                    Imin = i;
                    Jmin = j;
                  }
              } // else
          } // for int j ....
      } // for int i ....

    // store Ymin in Yn
    Yn[Njets-3] = Ymin;
    Njets –;

    // recombine according to E-Scheme
    if (Scheme == 1)
      {
        P[Imin] = P[Imin] + P[Jmin];
      }

    // recombine according to E0-Scheme
    if (Scheme == 0)
      {
        P[Imin].setE(P[Imin].e() + P[Jmin].e());
        float P2i = Hep3Vector(P[Imin]+ P[Jmin]).mag();
        if (P2i <= pow(10,-14))
          {
            cout << " Error: Division by zero! " << endl;
            exit(1);
          }
        P[Imin].setPx(P[Imin].e() * (P[Imin].px() + P[Jmin].px())/P2i);
        P[Imin].setPy(P[Imin].e() * (P[Imin].py() + P[Jmin].py())/P2i);
        P[Imin].setPz(P[Imin].e() * (P[Imin].pz() + P[Jmin].pz())/P2i);
      }
    // lock the other track
    Iflag[Jmin] = 1;
    IminC = Imin;
  } //while
 return Yn;
}

//////////////////////////////////////////////////////////////////////////
//thrust algorithms
//////////////////////////////////////////////////////////////////////////
```

```
AlThrust AlephAlgo::AThrust(ALEPHTYPE type)

{
  /*
  Algorithm: try initial trust axis from sum of projections along
  all cross products between pairs of particles. Take
  the axis with maximum thrust value and iterate once
  more.

  The HepLorentzVector _A4V of thrust objects gets the thrust vector as Hep3Vector
  and the thrust value into the fourth (energy) component.
  */

  AlphaBanks* b = getAlphaBanks();
  if (((type != TRACK) && (type != EFLOW)) && ((type != GAMPEC) && (type != ALJET)))
    {
      cout << " AThrust : WARNING : type " << type << " not supported! " << endl;
      exit(1);
    }
  vector<AlObject*> selObjs;
  // select the objects
  for(vector<AlObject*>::iterator iobj=b->ObjectV().begin(); iobj < b->ObjectV().end(); iobj++)
    {
      if ( ((*iobj)->TYPE() == type) && ((*iobj)->isLocked() == 0) )
        {
          selObjs.push_back((*iobj));
        }
    }
  // call the appropriate thrust routine
  return AThrust(selObjs);
}

/////////////////////////////////////////////////////////
AlThrust AlephAlgo::AThrust(vector<AlObject*> Pext)

{
  /*
  Algorithm: try initial trust axis from sum of projections along
             all cross products between pairs of particles. Take
             the axis with maximum thrust value and iterate once
             more.

  The HepLorentzVector _A4V of thrust objects gets the thrust vector as Hep3Vector
  and the thrust value into the fourth (energy) component.
  */
```

```
AlphaBanks* b = getAlphaBanks();

vector<Hep3Vector> p;
int Np =Pext.size(); //number of particles
AlThrust thr; //variables of thr are zero as set by the default constructor

// select the three-momenta
for(int i=0; i < Np; i++)
  {
    p.push_back(Hep3Vector(Pext[i]->A4V()));
  }

Hep3Vector qtbo;
Hep3Vector zero(0.,0.,0.);
float vnew;

// for more than 2 tracks
if (Np > 2)
  {
    float vmax = 0.;
    Hep3Vector vn, vm, vc, vl;

    for(int i=0; i < Np-1; i++)
      { for(int j=i+1; j < Np; j++)
          {
            vc = p[i].cross(p[j]);
            vl = zero;

            for(int k=0; k<Np; k++)
              {
                if ((k != i) && (k != j))
                  {
                    if (p[k].dot(vc) >= 0.)
                      {
                        vl = vl + p[k];
                      }
                    else
                      {
                        vl = vl - p[k];
                      }
                  }
              }

            // make all four sign-combinations for i,j

            vn = vl + p[j] + p[i];
```

```
        vnew = vn.mag2();
        if (vnew > vmax)
          {
            vmax = vnew;
            vm = vn;
          }

        vn = vl + p[j] - p[i];
        vnew = vn.mag2();
        if (vnew > vmax)
          {
            vmax = vnew;
            vm = vn;
          }

        vn = vl - p[j] + p[i];
        vnew = vn.mag2();
        if (vnew > vmax)
          {
            vmax = vnew;
            vm = vn;
          }

        vn = vl - p[j] - p[i];
        vnew = vn.mag2();
        if (vnew > vmax)
          {
            vmax = vnew;
            vm = vn;
          }
      } // of for j
  } // of for i

// sum momenta of all particles and iterate
for(int iter=1; iter<=4; iter++)
  {
    qtbo = zero;
    for(int i=0; i< Np; i++)
      {
        if (vm.dot(p[i]) >= 0.)
          {
            qtbo = qtbo + p[i];
          }
        else
          {
            qtbo = qtbo - p[i];
```

```
          }
        }
      vnew = qtbo.mag2();
      if (vnew == vmax) { break;}
      fout << endl << " I have to iterate again " << endl << endl;
      vmax = vnew;
      vm = qtbo;
    }
  } // of if Np > 2

else
 {
   if (Np == 2)
     {
       if (p[0].dot(p[1]) >= 0.)
         {
           qtbo = p[0] + p[1];
         }
       else
         {
           qtbo = p[0] - p[1];
         }
     }
   else if (Np == 1)
         {
           qtbo = p[0];
         }
   else
     {
       qtbo = zero;
       // thrustvalue and direction are set to zero by default constructor
       b->addObject(&thr);
       return thr;
     }
 }

// normalize thrust -division by total momentum-

float vsum = 0.;
for(int i=0; i < Np; i++)
 {
   vsum = vsum + p[i].mag();
 }

vnew = qtbo.mag();
float v = vnew/vsum;
```

```
  float x = qtbo.x()/vnew;
  float y = qtbo.y()/vnew;
  float z = qtbo.z()/vnew;

  thr.setA4V(x, y, z, v);

  // fill the thrust object into AlObject* vector ObjectV
  b->addObject(&thr);

  return thr;
}
```

11.11 User Analysis program

Description of it: see Chapter 7
Author: M. Hörndl

```
#include <fstream.h>
#include "AlephExManager.h"
#include "Hbook.h"
#include "AlephAlgo.h"
#include "AlphaCopy.h"

extern ofstream fout;
static int Ngood = 0;
static int Nev = 0;


static float Ycut[10] = { 0.001, 0.0025, 0.004, 0.0055, 0.0085,
                          0.01, 0.025, 0.04, 0.08, 0.1 };

/////////////////////////////////////////////////////////////////////
// User Init (routine for initiation, done before User Event)
/////////////////////////////////////////////////////////////////////
void AlephExManager::UserInit()

{
 // book an ntuple
 int ntags = 28;
 char** mytags = new char*[ntags];
 mytags[0] = "NjY1";
 mytags[1] = "NjY2";
 mytags[2] = "NjY3";
 mytags[3] = "NjY4";
 mytags[4] = "NjY5";
 mytags[5] = "NjY6";
 mytags[6] = "NjY7";
 mytags[7] = "NjY8";
 mytags[8] = "NjY9";
 mytags[9] = "NjY10";
 mytags[10] = "P1E";
 mytags[11] = "P1X";
 mytags[12] = "P1Y";
 mytags[13] = "P1Z";
 mytags[14] = "P2E";
 mytags[15] = "P2X";
 mytags[16] = "P2Y";
 mytags[17] = "P2Z";
```

```
 mytags[18] = "P3E";
 mytags[19] = "P3X";
 mytags[20] = "P3Y";
 mytags[21] = "P3Z";
 mytags[22] = "NCH1";
 mytags[23] = "NCH2";
 mytags[24] = "NCH3";
 mytags[25] = "psi12";
 mytags[26] = "psi13";
 mytags[27] = "psi23";

 Hbookn(500,"myNtuple",ntags,mytags);
 delete [] mytags;
 fout << endl << " =====> in UserInit " << endl;
}
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////
// User Event (this routine is called for every event!)
//////////////////////////////////////////////////////////////////////
void AlephExManager::UserEvent(AlphaBanks& bb)

{
  // set the ntuple array
  float ntvar[28] = { 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ,
                      0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ,
                      0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. };

  Nev++;
  typedef AlphaBanks::AlObjectVec::iterator ObjItr;
  AlephAlgo a(&bb);

  // only EFLOW objects are used, therefore lock all the other objects
  for(ObjItr it = bb.ObjectV().begin(); it < bb.ObjectV().end(); it++)
    {
     if ((*it)->TYPE() != EFLOW)
       {
         (*it)->Lock();
       }
    }

  // do track selection
  for(ObjItr it = bb.ObjectV().begin(); it < bb.ObjectV().end(); it++)
    { if ((*it)->isLocked() == 0)
       {
```

```
        int type = ((AlEflw*)(*it))->getEfType();
        QvecBase* tmp = (QvecBase*)(*it);

        // selection of good charged tracks
        if (type <=3)
          {
            int Ntpc; // get Ntpc
            AlTrack* theAssTrack = ((AlEflw*)(*it))->getTrack();
            if (theAssTrack == NULL)
              {
                Ntpc = 0;
              }
            else
              {
                Ntpc = theAssTrack->NT();
              }

            float ptch = tmp->QPT(); // transverse momentum
            float costheta = tmp->QCT();
            float d0 = tmp->QDB();
            float z0 = tmp->QZB();

            // lock if bad
            if ((d0 > 2.) || (z0 > 5.) || (fabs(costheta) > 0.94)
                          || (ptch < 0.2) || (Ntpc < 4))
              {
                (*it)->Lock();
              }
          }
        // all Eflow objects with eflow type > 5 are locked
        if (type > 5)
          {
            (*it)->Lock();
          }
      }
  }

// check if Nchgood and Echgood are big enough
int Nchgood = 0;
int Echgood = 0.;
for(ObjItr it = bb.ObjectV().begin(); it < bb.ObjectV().end(); it++)
  {
    if ((*it)->isLocked() == 0)
      {
        int type = ((AlEflw*)(*it))->getEfType();
        if (type <=3)
```

```
          {
            Nchgood ++;
            Echgood += (*it)->QE();
          }
        }
  }

//compute thrust axis with all good Eflows
AlThrust theThrust = a.AThrust(EFLOW);
Hep3Vector theThrAxis = theThrust.getThrustDirection();
float costhr = theThrAxis.cosTheta();

// final event selection
if ((Nchgood >=5) && (Echgood >=15.) && (fabs(costhr) < 0.82))
  {
    // now he have a good event
    Ngood++;

    // store all not locked objects in the vector<AlObject*> selobj;
    vector<AlObject*> selobj;
    for(ObjItr it = bb.ObjectV().begin(); it < bb.ObjectV().end(); it++)
      {
        if (((*it)->isLocked() == 0) &&
          ((*it)->TYPE() == EFLOW))
          {
            selobj.push_back(*it);
          }
      }

    // calculate various properties with the non locked objects

    // compute the different jet-rates as a function of Ycut
    // 10 different Ycut values set

    int jetr;
    HepLorentzVector jetmomentum[3];
    float multiplicity[3] = {0., 0., 0.};
    float interjetangle[3];

    for(int i=0; i<10; i++)
      {
        vector<AlJet> Jn = a.DurhamJn(1, Ycut[i], selobj);
        jetr = Jn.size();
        ntvar[i] = (float)(jetr);

        // take three jet events for Ycut = 0.01
```

```
if (( i == 5) && ( jetr == 3 ))
  {
    for(int j=0; j<3; j++)
      {
        jetmomentum[j] = Jn[j].A4V();

        // calculate the multiplicity
        for(ObjItr it1 = Jn[j].getObjects().begin();
                          it1 <Jn[j].getObjects().end(); it1 ++)
          {
            int type = ((AlEflw*)(*it1))->getEfType();
            if (type <=3)
              {
                multiplicity[j]++;
              }
          }
      }
    // store the jets so that the energy decreases from jetmom[0] to jetmom[2]
    int ready = 0;
    while (ready == 0) {
       ready = 1;
       for (int k = 0; k < 2; k++) {
         for (int l = 1; l < 3; l++) {
           if (l != k)
             { if (jetmomentum[l].e() > jetmomentum[k].e()) {
                 HepLorentzVector mom = jetmomentum[k];
                 jetmomentum[k] = jetmomentum[l];
                 jetmomentum[l] = mom;
                 float mult = multiplicity[k];
                 multiplicity[k] = multiplicity[l];
                 multiplicity[l] = mult;
                 ready = 0;
             } // of if
           } // of if
         } // of for
       } // of for
    } // of while

    // calculate the interjetangles
    interjetangle[0]=jetmomentum[0].angle(Hep3Vector(jetmomentum[1]));
    interjetangle[1]=jetmomentum[0].angle(Hep3Vector(jetmomentum[2]));
    interjetangle[2]=jetmomentum[1].angle(Hep3Vector(jetmomentum[2]));

    for (int ih=0; ih<3; ih++)
      {
        // fill the rest of the ntuple array
```

```
                ntvar[10+4*ih] = jetmomentum[ih].e();
                ntvar[11+4*ih] = jetmomentum[ih].px();
                ntvar[12+4*ih] = jetmomentum[ih].py();
                ntvar[13+4*ih] = jetmomentum[ih].pz();
                ntvar[22+ih] = multiplicity[ih];
                ntvar[25+ih] = interjetangle[ih];
              }
          } // of if 3jet
      } // of for over Ycut

    // fill the ntuple array into the ntuple
    int id = 500;
    hfn_(&id, ntvar);
  } //of if good event
 fout << endl << " =====> in UserEvent " << Nev << endl;
}

//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////
// User Term (User Termination routine is called at the end)
//////////////////////////////////////////////////////////////////////
void AlephExManager::UserTerm()

{
 fout << endl << " =====> in UserTerm " << endl;
 fout << endl << " Number of selected events = " << Ngood << endl << endl;
}
```