Visualization and Alignment of the Muon Chambers at the H8 Combined Testbeam

Master's thesis by:
Eric Jansen

Supervisor:
Dr. C.W.J.P. Timmermans

HEN-457

August 29, 2005

Contents

Chapter 1

Introduction

1.1 LHC

The Large Hadron Collider (LHC) is currently being constructed at the CERN (European Organization for Nuclear Research) laboratory near Geneva. LHC occupies the almost circular tunnel previously used for the LEP accelerator, which has a circumference of 27 km and lies at a depth varying between 50 and 150 m underneath the surface. The LHC will collide proton beams. Each interaction will have a center of mass energy of 14 TeV. Four detectors are located at the interaction points of the proton beams, these form the hearts of the ATLAS, CMS, ALICE and LHCb experiments. Operation of the Large Hadron Collider is scheduled to start in 2007.

The main physics goal of the LHC is [1] to provide new insight into the mechanism of spontaneous symmetry-breaking in the standard model. This mechanism, which is believed to be responsible for particle masses, could manifest itself by the existence of the Higgs particle. It is this particle that LHC is trying to find.

1.2 ATLAS

ATLAS (A Toroidal LHC ApparatuS) is one of the two general purpose detectors at the LHC and as such it is designed for a very broad spectrum of physics. A picture of ATLAS is shown in figure 1.1. Starting from the beam pipe and proceeding outwards ATLAS contains [2]:

- The inner detector covers the first 1.15 m in radius. It contains three layers of pixel detector followed by four double layers of silicon strip detector. The outer region of the inner detector is used for the transition radiation tracker. The whole inner detector is placed in a 2 T solenoidal magnetic field. It has to provide efficient tracking

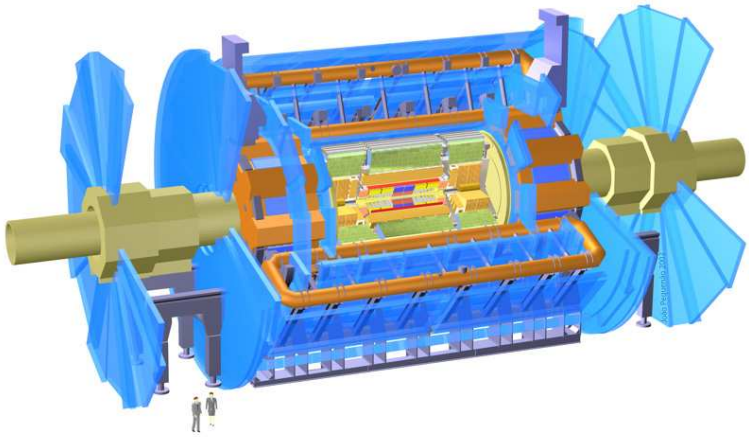

Figure 1.1: The ATLAS detector. (Picture by CERN)

at high luminosity for measuring high-p_{T} lepton-momentum. In combination with other detectors identification of leptons, photons and quarks should be possible.

- The calorimeter consists of a liquid argon electromagnetic sampling calorimeter followed by a scintillator-tile hadronic calorimeter. A good electromagnetic calorimetry is needed for measurement and identification of electrons and photons. The hadronic calorimeter has to provide full-coverage for accurate jet and missing transverse energy measurements. The calorimeters extend up to a radius of 4.25 m from the beam pipe.
- The muon spectrometer extends up to a radius of 10 m. It consists of three layers of detection chambers inside a 0.5 T toroidal magnetic field. It has to be able to do high-precision p_{T} measurements for muons at the highest luminosities using only information from the muon system itself.

To minimize the time needed for commissioning once construction of the full detector is complete, several test setups have been built to test individual subdetectors or combinations of subdetectors. The biggest effort of this kind for ATLAS was the data taking at the H8 combined testbeam, which took place from May to November 2004. At this testbeam, most of the subdetectors were present and this was the first combined data taking using the ATLAS software infrastructure. This thesis focuses on the muon chambers in the H8 testbeam setup.

Chapter 2

Combined testbeam setup

The ATLAS muon spectrometer contains four different types of muon detectors [2, 3]. It consists of monitored drift tube (MDT) chambers and resistive plate chambers (RPC) in the barrel, thin gap chambers (TGC) and monitored drift tube chambers in the endcaps and cathode strip chambers (CSC) in the forward regions. Details of these chambers will be discussed below.

Figure 2.1 shows a 3D plot of the muon chambers present at the H8 combined testbeam setup in 2004, which resembles a slice of ATLAS. Three barrel stations (shown in blue) are positioned as in an odd numbered (large) sector of ATLAS. From left to right we have the inner, middle and outer station. Further downstream is an endcap setup (shown in green) resembling part of two adjacent sectors of the endcap in ATLAS. Located around the middle endcap station are three TGCs (shown in magenta). They are normally mounted onto the inner and middle endcap stations, but they are used in a standalone mode here. Furthermore there is one CSC (shown in yellow) in between the endcap and barrel setup.

The right-handed coordinate system is chosen such that the x-axis is oriented in the direction of the beam, the y-axis is pointing upwards and the z-axis is orthogonal to x and y. Because of the cylindrical geometry of ATLAS itself, the software usually describes the testbeam setup in terms of the cylindrical coordinates $\rho = \sqrt{x^2 + y^2}$, $\phi = \arctan \frac{y}{x}$ and z.

2.1 Monitored drift tube chambers

The basic detection element of an MDT chamber is an aluminum tube of diameter 30 mm filled with a pressurized gas mixture. In the center of the tube is a tungsten-rhenium alloy wire which is at an electrical potential of 3270 V. The length of the tube depends on the chamber it is mounted in and varies from 0.70 m to 6.30 m.

A muon passing through this tube will ionize some of the gas molecules and the freed electrons will drift towards the wire. These electrons ionize new gas molecules and this avalanche effect leads to a signal amplification by a factor 2×10^4. This results in a

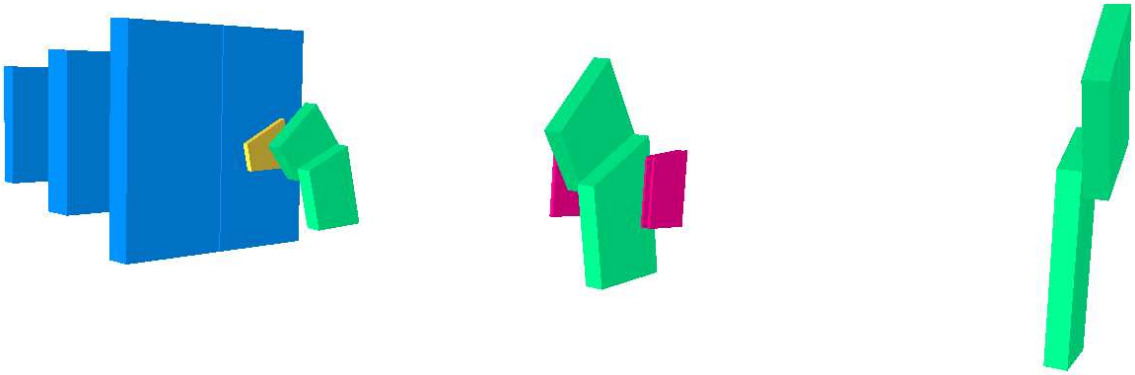

Figure 2.1: The muon chambers in the combined testbeam setup.

measurable signal on the wire. The quantities of interest of this signal are:

Leading edge time: this is the time at which the height of the signal exceeds a certain threshold. The leading edge of the signal originates from electrons created at the point closest to the wire. The time difference between the passage of the muon and the leading edge of the signal is a measure for the radius at which the tube was hit. This time difference is referred to as the drift time.

Trailing edge time: the electrons that form the trailing edge originated from ionizations near the wall of the tube. As this is a fixed distance from the wire, the trailing edge signal will have a fixed latency to the traversal of the particle.

Integrated signal: this corresponds to the amount of charge deposited on the wire. The integrated signal can be used to correct the leading edge time for different rise times of the signal and provides information about the gas gain.

A schematic drawing of an MDT chamber is shown in figure 2.2. An MDT consists of six parallel layers of drift tubes, arranged in two multilayers, each containing three[1] parallel layers of tubes. Combining the drift times measured in these six layers yields one coordinate with 40 µm precision and an angle with 3×10^{-4} rad precision in the plane perpendicular to the tubes. The MDTs do not ordinarily provide any information about the coordinate along the length of the tubes. However, a readout in so-called twin-tube mode [4] would make this information accessible.

[1]The innermost chambers have four layers in each multilayer, all other chambers have three.

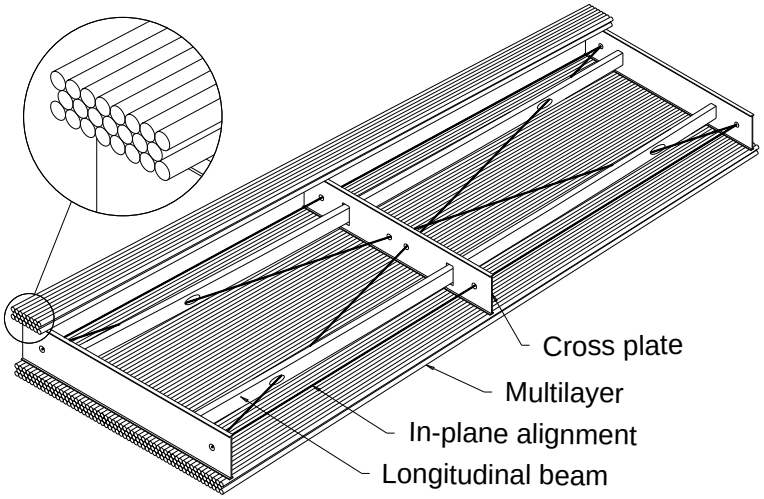

Figure 2.2: Structure of a monitored drift tube chamber [3].

2.2 Resistive plate chambers

In the barrel region of ATLAS the triggering and second coordinate measurement is provided by the resistive plate chambers. At the combined testbeam, triggering is done using scintillator plates, so the RPCs are only used for the second coordinate measurement.

The basis of an RPC is formed by two thin Bakelite plates with a 2 mm gas gap in between. These plates have a high volume resistivity ($10^{10}\,\Omega$cm). The outer surface of such a plate is covered with a thin layer of graphite paint followed by a layer of insulation. The graphite films are connected to a high voltage power supply to create a uniform electric field inside the gas gap. On the outside of this structure, insulated from the graphite layer, are the pick up strips. On one side these strips are oriented parallel to the MDT tubes (the longitudinal or η strips[2]) and on the other side they are oriented perpendicular to the tubes (the transversal or ϕ strips).

When a muon passes through the gas gap it will ionize some of the gas and it creates an avalanche of charged particles. This charge will drift towards the plates where it can be detected by the capacitative effect on the pick up strips.

Two of these detector layers, overlapping with the next set to avoid dead areas, are clamped together between an aluminum and foam sandwich structure to form a chamber (see figure 2.3).

The MDT chambers in the middle barrel station are covered with RPCs on both sides. In the outer barrel station, only one side is covered. In such a setup, with six consecutive layers of RPCs, a resolution of about 6 mm in η and ϕ is obtained. The time resolution of

[2] $\eta = -\ln\left(\tan\frac{\theta}{2}\right)$ being the so-called pseudo-rapidity (with θ the angle with the z-axis).

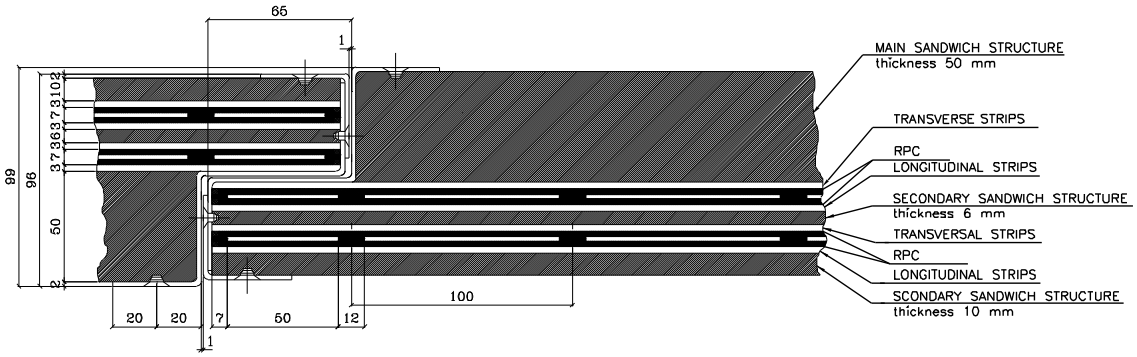

Figure 2.3: Structure of a resistive plate chamber [3].

an RPC is 1.5 ns.

2.3 Thin gap chambers

In the endcaps of ATLAS triggering and measurement of the ϕ coordinate is done by thin gap chambers. A TGC (shown in figure 2.4) consists of two G10 plates covered with a graphite layer. In between these plates is a gas gap. The inside of the plates is covered with a layer of graphite and these layers function as cathode planes. In the middle of this gap, at 1.4 mm distance from the cathode planes, is a plane of anode wires spaced 1.8 mm apart. The anode wires are parallel to the MDT tubes in the endcap, so they measure the η coordinate.

Two or three of these structures are bonded together with a paper honeycomb structure in between to form a doublet or a triplet TGC module. On the outer cathode planes, copper strips are then added to measure the ϕ coordinate. This is also done by making use of the capacitative effect, similar to the RPC pick up strips.

2.4 Cathode strip chambers

The fourth type of muon chamber is the cathode strip chamber. A CSC is a multiwire proportional chamber used in the forward region, because it can handle higher counting rates. While for an MDT chamber drift times can range up to 500 ns, for a CSC this is only about 30 ns.

One CSC is present at the testbeam setup, but it has not contributed any data.

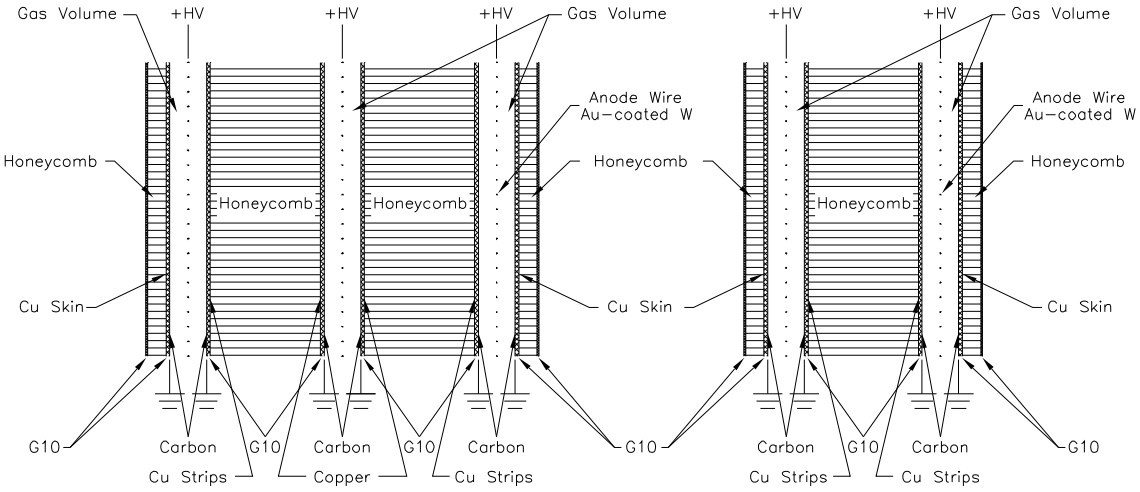

Figure 2.4: Structure of a thin gap chamber triplet (left) and doublet (right) [3].

Chapter 3

Software

3.1 Athena

For ATLAS a new software framework is being developed to accommodate the simulation as well as the reconstruction and analysis of event data. This framework is called Athena. One of the main goals of Athena is [5] to let the users concentrate on their physics analysis and separate them as much as possible from the implementation details of the rest of the software. This is achieved by using an object-oriented programming language. For Athena, C++ was chosen.

In an object-oriented language a program is composed of objects that group the data together with procedures for manipulating it. Instead of modifying the data directly, the data is now modified by calling these procedures. This way the interaction between objects takes place via well-defined interfaces, without the need to know any of the implementation details. This is exactly the level of abstraction needed in this case.

An instructive example of this object-oriented approach is the following: consider two objects representing geometrical shapes, say a cube and a sphere, that we would like to know the volume of. In a normal programming language we would need to know the exact shape we're dealing with, and what their parameters are, so we can do two separate calculations for the cube and the sphere. In an object-oriented language, both objects would implement a `volume()` method for this task. Now the program interested in the volume of the shape simply calls the `volume()` method, without needing to know any other properties of the object. Even if someone would decide to replace the cube by a cylinder the program would still function properly. Naturally, the same calculation still has to be done, but the organization of the program has improved.

Athena is built up of mostly two types of objects:

Data objects contain the actual data. A hit, a track or a collection of these are examples of data objects. Obviously a lot of these objects will need to be created to store

an event in memory, so data objects need to be small. A track only contains the parameters of the track, not the methods to fit itself.

Algorithms contain the procedures that operate on these objects and possibly create new data objects. For example a track fit algorithm that uses a collection of hits to fit a track and create a corresponding track object.

This clear separation might seem somewhat like a deviation from the concepts of object-oriented programming, but the reasons for this will become clear when we discuss the way the algorithms interact with the data objects.

The basis of an Athena session is the job options script. This script, written in the Python scripting language, controls which algorithms are run and the order in which this happens.

Algorithms cannot pass the data objects to each other directly, as this would fix the order in which the algorithms are connected. This is solved by retaining the objects in memory and maintaining pointers to the objects outside the algorithm. Another algorithm can then retrieve the pointer to an object again and access it.

This software architecture is often referred to as the "blackboard model", as the data store can be thought of as a blackboard. You can read the data you need from it, then you do your calculations and write your results back onto the blackboard, without the need to worry about who wrote the initial data and who will be using your results.

Athena uses two data stores: a transient and a persistent data store. The transient data store is used for temporarily storing the event data, it is emptied after an event is processed. Data in the persistent data store remains in memory for the entire session. This is used, for example, to store the detector geometry.

The available algorithms and data objects for Athena are grouped into packages. The following sections will discuss some of the software packages relevant for this thesis.

3.1.1 Detector geometry description

The detector description for ATLAS is provided by the GeoModel package. This package provides a method to access the central ATLAS geometry database from within Athena.

For consistency between simulated and reconstructed data, both the detector description used for simulation and the one used for reconstruction are generated from the same database. For reconstructing an event the positions of the readout elements suffice, but for simulation all material a particle might scatter off has to be included as well. This means that, in addition to all the readout elements, materials such as the magnet coils and the support structures need to be parametrized as well. GeoModel does provide all this information.

Prior to each run an Athena service reads the applicable geometry from this central database and stores this into the persistent data store by means of detector manager

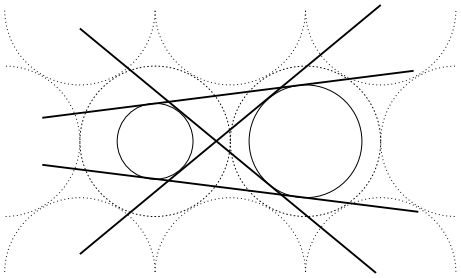

Figure 3.1: Four possible segments for two hits.

objects. These objects can be accessed from the algorithms and provide procedures for accessing the properties of arbitrary detector elements.

3.1.2 Muon track reconstruction

For reconstructing tracks in the muon spectrometer the MOORE (Muon Object Oriented REconstruction) package is used. The reconstruction of a muon track is done in four steps [6].

The reconstruction starts by looking for track segments in ϕ. In the testbeam there is no magnetic field, and in ATLAS the magnetic field in the muon system is toroidal. There is no bending of the tracks in the x-y plane. All the muon trajectories are straight lines and the ϕ segments are basically a collection of hits with approximately the same ϕ coordinate. They are created by histogramming all hits from the ϕ measuring chambers (RPC, TGC and CSC). The bins that exceed a certain threshold are accepted as a segment.

Next is the creation of segments in the ρ-z plane. In ATLAS, this would be the plane in which the muon trajectories are curved. On the scale of a single chamber though, approximating the curved tracks by straight track segments is possible.

In the ρ-z plane the segments are created from the information given by the MDT tubes, which are oriented perpendicular to this plane. The trajectory of the muon should be tangent to the drift circles of the tubes.

The reconstruction starts by selecting two hits, which will yield four possible segments (see figure 3.1). These segments are then extrapolated to the other layers of the chamber to find other hits that might lie on them. This step is repeated until all hits are assigned to a segment. Segments that are allowed by the cut values are kept for the next stage of the reconstruction.

Once the ρ-z segments are created, they are combined with the ϕ segments to create the fine segments. These fine segments are then combined with fine segments from other

stations. Finally, a refit is done using all hits of such a combination, resulting in a crude global muon track, also known as a road.

Not all stations are equipped with ϕ measuring chambers though, so not all ρ-z segments are assigned to a road this way. The final step in the reconstruction is therefore to assign the remaining segments in the ρ-z plane to a road.

Once this is complete, a full refit is done taking into account the energy loss and Coulomb scattering effects. Finally, hits with high residuals are considered to be wrongfully assigned to the track and are removed.

An example of such a track is displayed in figure 3.2.

3.1.3 Converting events to XML

The JiveXML package provides the link between the Atlantis event viewer and the main offline software for ATLAS. Because Atlantis is a standalone program which cannot be run directly from Athena, all data necessary for visualizing an event needs to be converted from C++ objects into a format Atlantis can deal with.

To accomplish this, JiveXML is composed of objects called data retrievers. A data retriever is a C++ class that retrieves one specific type of data for Atlantis. It retrieves the needed objects from the transient data store, does any necessary calculations and returns the results in a standard form. At the moment there are about thirty different data types in Atlantis, this translates into thirty data retrievers in JiveXML.

The output from the data retrievers is then converted to XML[1] format and either written to a file or sent to Atlantis directly over a network connection.

Within this XML file, subsequent elements represent the different data types. Such elements contain, for example, the MDT hits, the simulated tracks or the muon segments. Within these data type elements are parameter elements, each containing an array of values separated by spaces.

In XML format three hits in an MDT chamber would produce a data type element that looks like the following fragment[2]:

```
<MDT count="3">
  <barcode>
    93 93 93
  </barcode>
  <driftR>
    1.185520 1.004480 0.817600
  </driftR>
  <id>
```

[1] eXtensible Markup Language; a widely used text based data format which allows users and developers to see and manipulate the raw data easily.

[2] Indentation was added for aesthetic reasons.

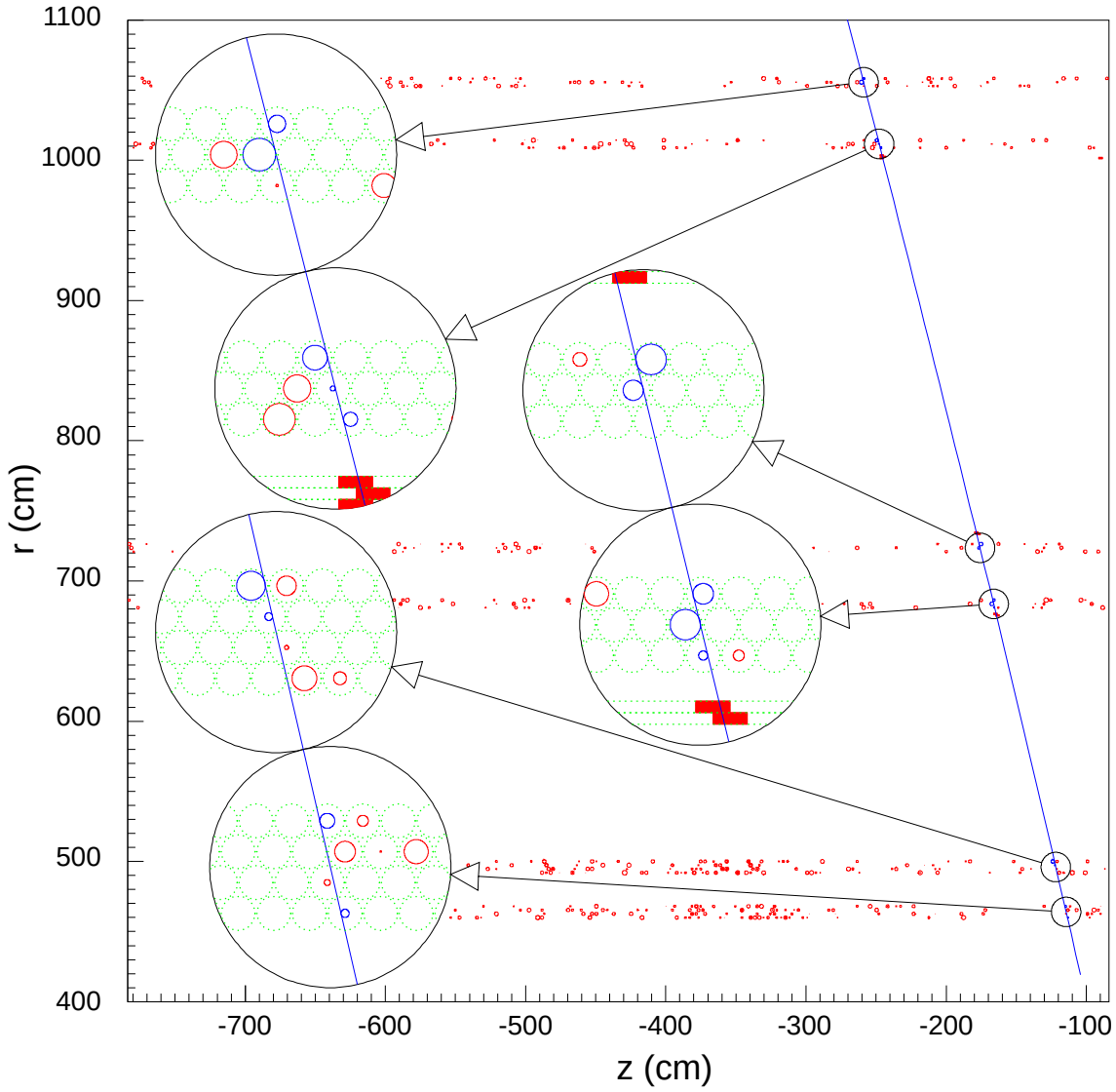

Figure 3.2: A reconstructed muon track in a 4-3-3 multilayer setup [3].

```
    -1 -1 -1
  </id>
  <identifier>
    7/BIS/3/2/MDT/1/1/5
    7/BIS/3/2/MDT/1/2/6
    7/BIS/3/2/MDT/1/3/6
  </identifier>
  <kine>
    6 6 6
  </kine>
  <length>
    153.149994 153.149994 153.149994
  </length>
  <x>
    169.479390 170.473602 171.467813
  </x>
  <y>
    409.159446 411.559686 413.959924
  </y>
  <z>
    218.000000 219.500000 221.000000
  </z>
</MDT>
```

Apart from the obvious information such as the coordinates and the drift radius, it also shows that the length and the full identifier of the tube are sent to Atlantis. The `barcode` and `kine` elements are used to associate simulated hits to a specific particle. They are not used in this case, neither is the `id` element.

3.1.4 An example Athena session

Now that some of the packages have been discussed, it's instructive to look at a (somewhat simplified) Athena session involving these algorithms. In figure 3.3 a schematic overview of a possible session is given. In this case, the JiveXML and the MOORE algorithm MuonTBSegmentMaker are used to generate an XML file containing MDT hits and fitted track segments.

On the left side in the figure is the Python script controlling the execution order and settings for the algorithms in the middle. As can be seen from this figure all communication between the algorithms is done via the transient data store. Only services have direct communication. For simplicity the persistent data store is not shown in the figure, as it is not part of the event data flowing through the system. It functions similarly to the transient data store.

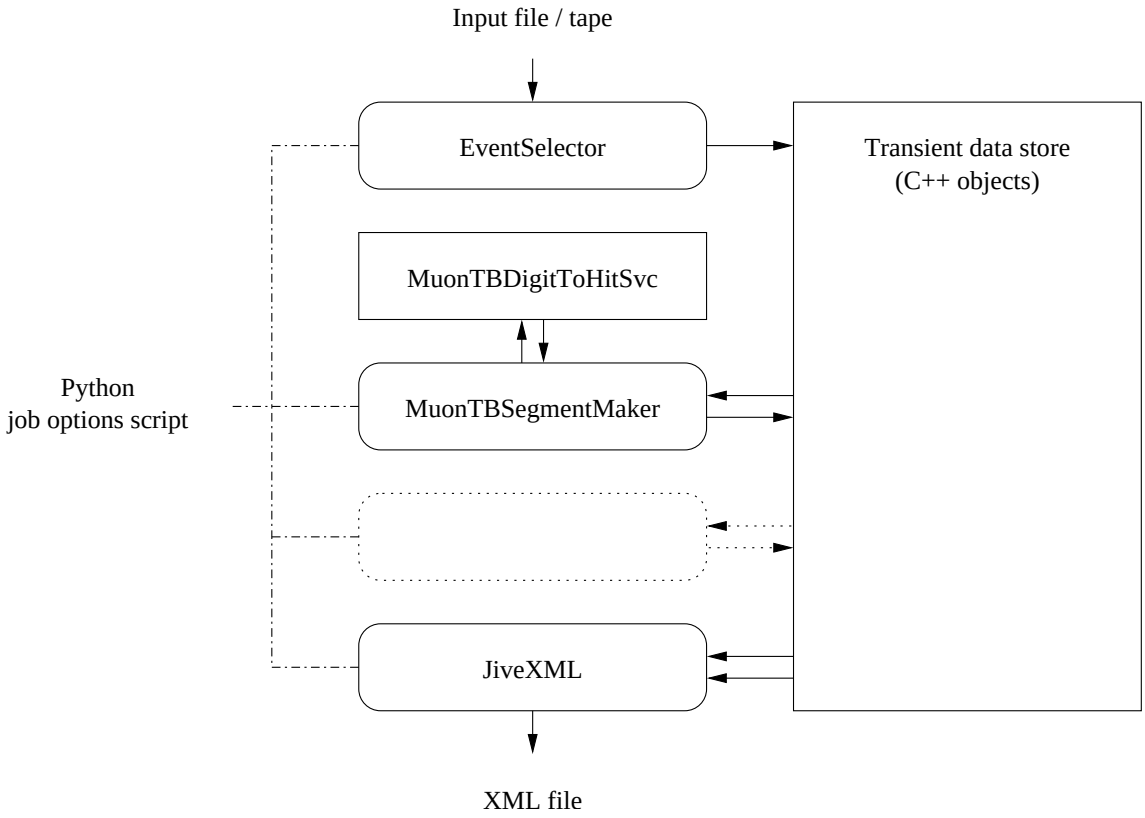

Figure 3.3: A schematic overview of an Athena session.

When the session is started, all required algorithms and services are loaded and their runtime options are set. Among others, it would need to tell the EventSelector which input file to use and how many events to process, tell the MuonTBSegmentMaker about the cuts we want to use and tell JiveXML where to write the XML file. Once this is complete, all components are initialized. During the initialization, algorithms can set up data structures and open any files they might need throughout the entire session.

Now the framework is ready to process the first event and this is where the sequence pictured in figure 3.3 starts. The EventSelector reads one event from its input file and converts the data into C++ objects. In this case, we are interested only in the MDT digits, so the only object recorded in the transient data store is a collection of MDT digits.

The next algorithm to be run is the MuonTBSegmentMaker. As its input, this algorithm needs the MDT digit collection, so it retrieves a pointer to this object in the transient data store. To be able to fit a segment, the drift radii need to be calculated first. This conversion is done by the MuonTBDigitToHit service, which the algorithm can access via the persistent data store. When the algorithm has finished fitting the segments, the created segment objects are once again recorded in the transient data store.

Last in this chain of algorithms is JiveXML. As discussed previously, JiveXML is used to convert the collection of C++ objects in the transient data store to XML files. In this example we would like to have both the MDT digits and the track segments visualized, so JiveXML retrieves the pointers to both objects in the data store. It then simply opens a file like any ordinary C++ program would do and writes the data to it.

After the final algorithm in the chain has been run the processing of the event has finished. At this point, all objects recorded to the transient data store are deleted and the system returns to its initial state. The EventSelector reads the next event and the chain of algorithms is run again.

Once a specified number of events has been reached or an error occurred somewhere, the system will jump to the finalization stage. Algorithms can utilize this opportunity to close any open files properly, free any memory they had in use and maybe print some summary information. When this is finished, Athena exits.

3.2 Atlantis event display

Understanding the enormous amount of data in a collision is a very complex task. A good visualization tool is absolutely necessary. Not only for knowing what goes on in terms of particles, but also for understanding the characteristics of our own hard- and software.

Development of Atlantis started about fifteen years ago at the ALEPH experiment at CERN. It started as the event display DALI [7], written in Fortran. It was used intensively at ALEPH for many years and eventually the program was rewritten in Java and adapted to ATLAS. This Java version is called Atlantis and is now the main event display for ATLAS.

The primary goal of the Atlantis event display is to help physicists investigate and understand complete events. For this investigation, several data-oriented projections are provided. These are two-dimensional projections chosen to best suit the geometrical properties of the detector elements. Because the chambers in the testbeam have been rotated, only the ρZ projection is useful for this setup.

Because it is written in Java, Atlantis cannot be run directly from the framework. Atlantis reads its event data from XML files that are generated by the JiveXML Athena algorithm. Once an event is converted to XML format it can be viewed with Atlantis independently from the framework. Nevertheless, it is also possible to receive events directly from Athena via a network connection and efforts for a two-way communication are underway.

Chapter 4

Visualization

4.1 Geometry

In addition to the event data, Atlantis also reads its detector geometry information from XML files. One of the visualization aspects of the testbeam involved retrieving the geometry from GeoModel and making the Atlantis displays with it.

Atlantis uses the geometry merely as a visual aid. Global coordinates for all hits are calculated by JiveXML and distributed with the event itself, so there is no need for a geometry down to the scale of single detection elements. The only geometry information needed by Atlantis are the outlines of the different detector chambers. This speeds up the drawing of the event and greatly reduces the size of the geometry files.

The geometry image Atlantis uses is defined using a basic set of geometrical shapes. The symmetry of the ATLAS detector is used as much as possible to reduce the size of the geometry definition.

4.1.1 Testbeam in 2003

In the first half of the year 2004 several key components of the software for the 2004 combined testbeam were not yet available. Therefore the visualization of the testbeam data started with the data taken in 2003.

The detector geometry description for the 2003 setup was not handled by GeoModel but by the MuonDetDescr package. However, to an end user the way of accessing the geometry is largely the same.

In 2003, the testbeam consisted of the same barrel and endcap setup as in 2004, only the CSC and TGC chambers were not present at that time. Because the chambers were nicely lined up with the Cartesian axes, the entire setup could be described in terms of rectangles. A picture of this setup is shown in figure 4.1.

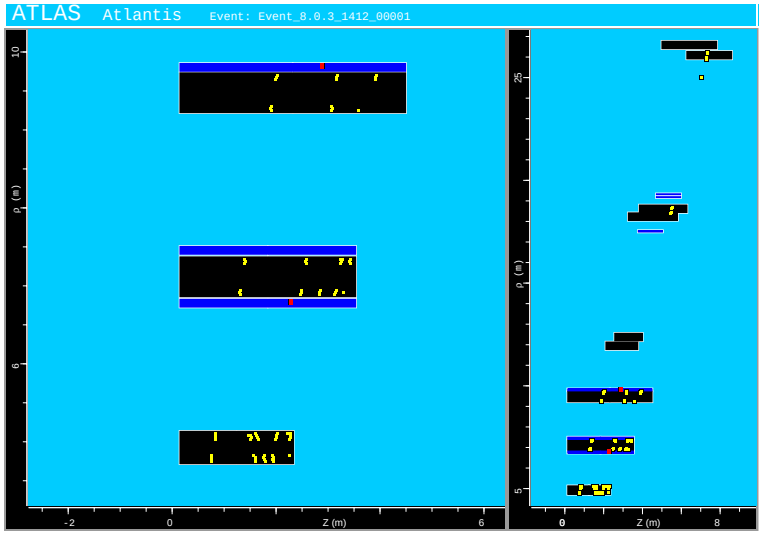

Figure 4.1: The geometry of the 2003 testbeam in Atlantis. This is the ρZ view, which basically provides a top view of the setup. Note that some of the MDT hits in the endcaps are displaced in ρ, this is due to a bug in Atlantis which was later resolved.

4.1.2 Testbeam in 2004

For the 2004 testbeam, the geometry becomes somewhat more complicated. Due to a change of the coordinate system, the barrel and endcap setup are now at a 15° angle with respect to the z-axis. Also, they are tilted forward by 1°. Even though the setup is still more or less the same slice of ATLAS, none of the projections in Atlantis provides an exactly perpendicular view of the chambers anymore.

As a consequence, the detector geometry has to be described in terms of polygons now. To generate these polygons we use the fact that the muon chambers are all convex shapes, so their projections should be convex too. The outline of a chamber with an arbitrary rotation can be created by projecting all corners onto the plane and then finding their convex hull. This results in the picture as displayed in figure 4.2.

Note that with the switch to GeoModel the definition of a detector element changed. Where the old detector description considered an MDT chamber as one element, GeoModel considers the two multilayers in the chamber as separate elements. A similar change affected the RPCs. As JiveXML displays the elements from the geometry package, this changed the way the chamber outlines are displayed in Atlantis.

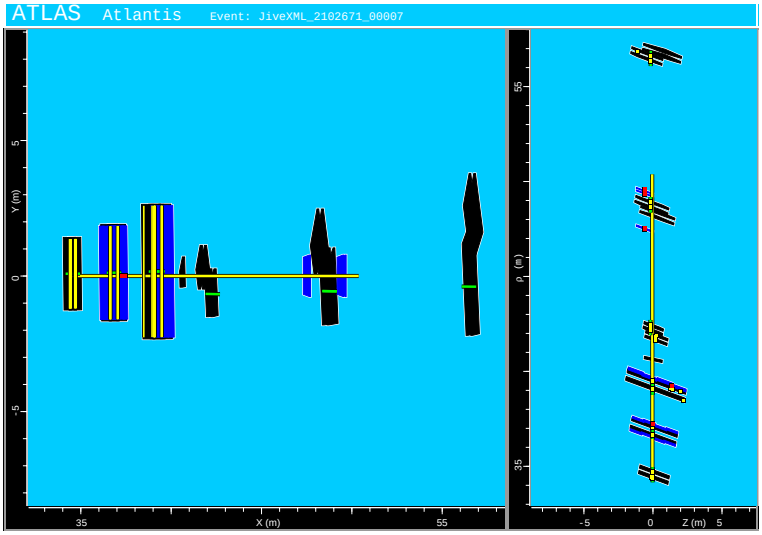

Figure 4.2: An event from the combined testbeam in 2004. On the left is the YX projection, looking at the setup from the side. On the right is the ρZ projection, which is approximately a top view in the testbeam.

4.2 r-t relation of the MDT tubes

Until now, JiveXML has used a linear relation between the drift radius and time for the MDT tubes. For the testbeam MDTs, this resulted in a drift radius spectrum as the one displayed in figure 4.3.

Obviously, this is not a very good approximation. The maximum drift radii are far beyond the radius of the tubes themselves, which is of course not possible. Also the distribution is not uniform, yet there is no reason for the tubes to be hit at a certain radius more often than others.

To properly display the MDT hits in Atlantis, the data retriever was modified to use the new Athena services to do the conversion. In case of the testbeam, the r-t relation is obtained by converting the time spectrum of a separate, rotating, MDT chamber into a uniform drift radius spectrum. This results in a spectrum as shown in figure 4.4.

4.3 Muon truth tracks

For studying the behavior of the pattern recognition software, it is useful to visualize the reconstructed tracks as well as the simulated ones, the so-called truth tracks. This was already done for the simulated tracks in the inner detector, but not for the tracks in the muon spectrometer.

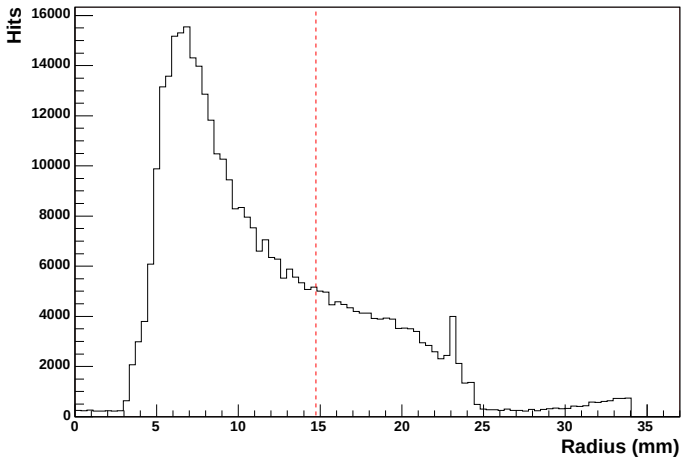

Figure 4.3: Drift radius spectrum for the testbeam chambers when using a linear relation with the drift time. The red line indicates the radius of the tube itself.

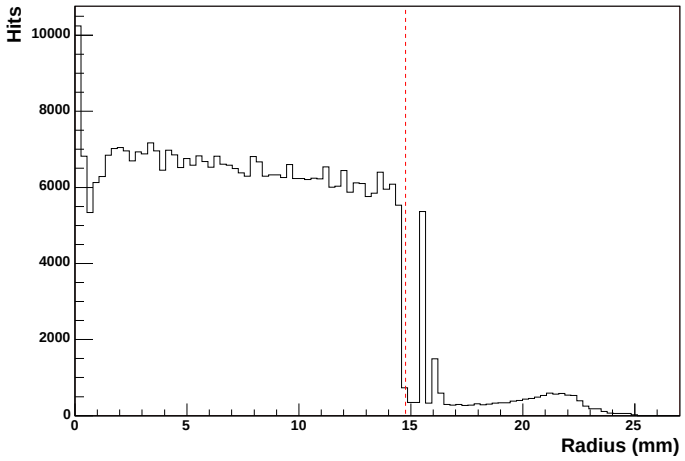

Figure 4.4: Drift radius spectrum for the testbeam chambers when using the MuonTB-DigitToHitSvc for the r-t calibration. The red line indicates the radius of the tube. The peak to the right of this line is caused by an anomaly in the time spectrum, it can also be seen in figure 4.3.

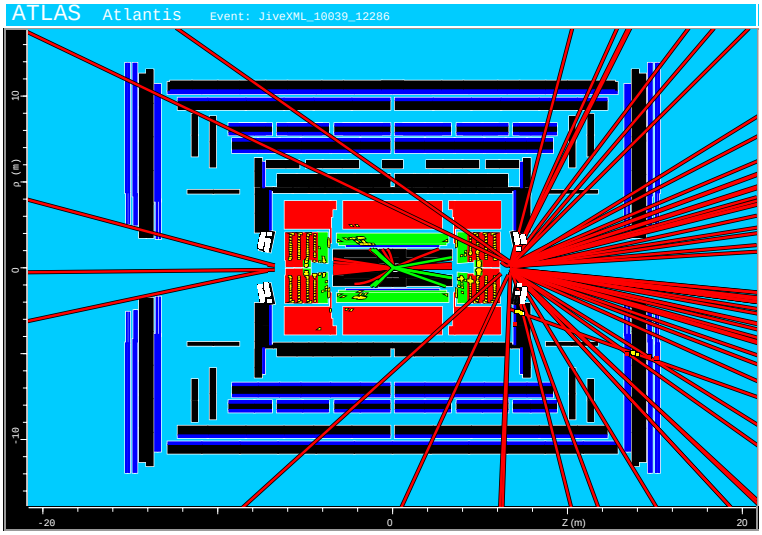

Figure 4.5: Simulated charged particles from the MuonEntryRecord, mostly low p_T electrons, shown in the ρZ projection of ATLAS.

In the software framework, information about the simulated particles traversing the muon detector can be retrieved in the form of the MuonEntryRecord. This object provides particle type, energy, position and momentum of all particles exiting the calorimeter. A data retriever was created to write this information to XML. The resulting picture is displayed in figure 4.5.

4.4 MDT segments

Another type of object that needed to be visualized were the fine MDT segments created by the MOORE package (section 3.1.2). These straight local tracks, produced as an intermediate object in the reconstruction, can be used for debugging the reconstruction software or for studying the alignment of the chambers. In both cases a proper visualization of the segments is a very helpful tool.

For this visualization a new data retriever had to be added to JiveXML. Figure 4.6 shows the segments in the testbeam setup, as displayed by Atlantis.

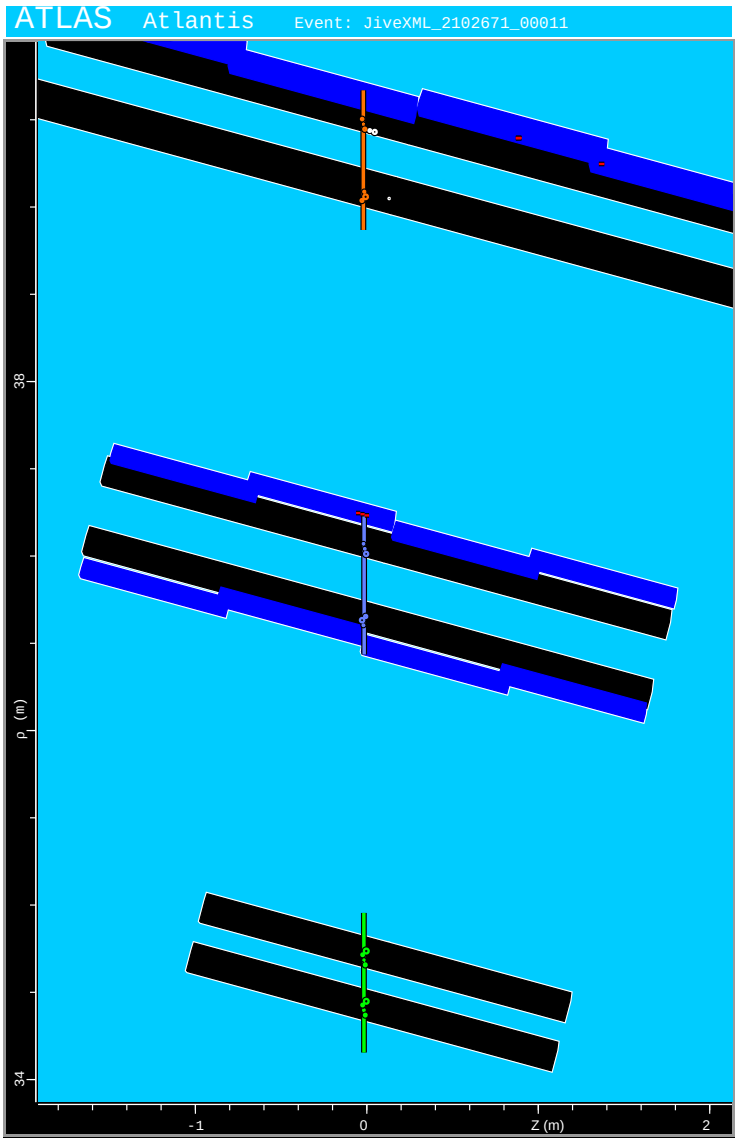

Figure 4.6: Track segments in the barrel chambers. The MDT hits are colored by the segment that they are associated to, unassociated hits are shown in white. The RPC hits are shown in red.

Chapter 5

Alignment with segments

In the ρ-z plane, track segments are created from the information of a single MDT chamber. These are not affected by misalignments of the chambers, so they provide a way to check, and further improve, the alignment. This is particularly useful in a setup without a magnetic field. In ATLAS the ρ-z plane is the bending plane of the magnetic field, but without a magnetic field the tracks in this plane are straight lines. This is the case for the testbeam setup. The muon trajectories are all straight lines, so the track segments from the different chambers should all line up nicely. Information about the relative alignment of the chambers can thus be obtained by simply extrapolating the segments and comparing them with each other.

Track segments in the ρ-z plane are defined by a ρ coordinate, a z coordinate and θ, the angle between the segment and the z-axis. In this chapter the differences in z and θ between the chambers will be discussed.

5.1 Theoretical expectations

To uniquely identify segments that belong to the same track, we need to concentrate on the events where the chambers under consideration contain exactly one track segment. Now there are two possibilities:

- The two segments do indeed belong to the same track. In this case the coordinate difference of the segments will be Gaussian distributed around the real coordinate difference between the chambers. The width of the distributions will be related to the resolution of the MDTs.
- One or both of the segments are incorrectly reconstructed. This means that there is no relation between the two segments and that their coordinate difference is uniformly distributed, at least in the area around the peak of the Gaussian.

The sample distribution for Δz and for $\Delta\theta$ will be the sum of these two distributions.

5.2 Implementation

The code for determining the misalignment of the chambers consists of two parts. The first part is an Athena algorithm that collects the information from the events and writes it to a file. The second part is a program that analyzes the data from this file.

5.2.1 Athena algorithm

The main functionality of the Athena algorithm is to extract the usable data from the transient data store and store it for further analysis. This algorithm groups the track segments it finds by chamber, selects the chambers that have only one segment and then writes the parameters of these segments to file.

5.2.2 Analysis program

The second part, the analysis program, is written using the ROOT framework. ROOT provides the program with the necessary methods for reading files and creating and fitting histograms.

The analysis of the data is a fairly straightforward process. First, we look for events with one segment in the reference chamber and one in the chamber we want to align. Then we extrapolate the segment in the reference chamber and calculate the difference in z and θ with the second segment. This data is then stored in a histogram to which a Gaussian plus uniform distribution is fitted.

5.3 Results

The inclination of the endcap chambers poses a problem for the track reconstruction. In ATLAS the endcaps are positioned such that the tubes of the MDT chambers are oriented perpendicular to the ρ-z plane. In the testbeam setup though, this is not the case. The resulting uncertainty in the z coordinate of the segments is of the order of tens of centimeters. Without a precise second coordinate measurement in these stations, alignment of the endcaps becomes very difficult. Hence we will focus on the alignment of the barrel chambers.

For the barrel setup, the total number of segments found in each of the chambers is listed in table 5.1. This indicates that most hits are in the in the left[1] series of barrel chambers. A logical choice for the reference chamber is therefore the BIL1 chamber, which is the first chamber of the left series. The number of coincidences between BIL1 and each of the other chambers is shown in the second column of the table. Based on these numbers, we will further narrow down our alignment to the left series of barrel chambers.

[1] When looking in the direction of the beam; these are the chambers lowest in z.

MDT chamber	Segments	Coincidences	Identifier
BIL1	206328	-	1627389952
BML1	94661	59041	1694498816
BOL1	248749	89402	1728053248
BIL2	4796	1079	1619001344
BML2	3344	655	1686110208
BOL2	121	21	1719664640

Table 5.1: Number of segments per chamber and number of times a coincidence with the BIL1 chamber occurred. This is based on the approximately 300,000 events in run 2102671.

Chamber	Run	$\Delta\theta$ (rad)	$\sigma_{\Delta\theta}$ (rad)
BML1	2102671	$-2.007 \pm 0.019 \times 10^{-4}$	$4.445 \pm 0.016 \times 10^{-4}$
	2102675	$-1.886 \pm 0.019 \times 10^{-4}$	$4.380 \pm 0.015 \times 10^{-4}$
BOL1	2102671	$-9.55 \pm 0.17 \times 10^{-5}$	$4.907 \pm 0.014 \times 10^{-4}$
	2102675	$-5.36 \pm 0.17 \times 10^{-5}$	$4.766 \pm 0.014 \times 10^{-4}$

Table 5.2: Displacements in θ for the BML1 and BOL1 chamber.

5.3.1 θ misalignment

Figures 5.1 to 5.4 show the distribution of the difference between the θ coordinate of the BML1 and BOL1 chambers and the BIL1 chamber. A summary of the fit parameters from these plots is given in table 5.2.

From the width of the $\Delta\theta$ distribution for the BML1 chamber in run 2102671 we obtain for the angular resolution of the chamber, σ_θ:

$$\sigma_\theta = \frac{1}{\sqrt{2}}\sigma_{\Delta\theta} = 3.143 \pm 0.011 \times 10^{-4}\,\text{rad}$$

this is in good agreement with the design angular resolution of 3×10^{-4} rad.

5.3.2 z misalignment

Figures 5.5 to 5.8 show the distribution of the difference between the z coordinate of the BML1 and BOL1 chambers and the BIL1 chamber. The fit parameters are summarized in table 5.3.

Based on this information, we can conclude that for adjacent stations the displacement in z can be determined with an uncertainty of less than 30µm when using around 1,000 events. This is comparable to the accuracy of the optical alignment system ATLAS will have.

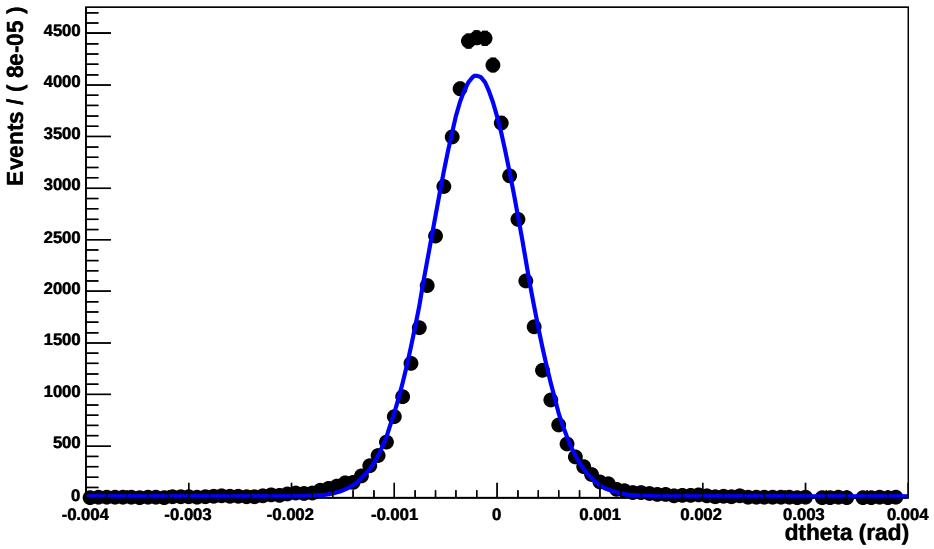

Figure 5.1: $\Delta\theta$ distribution for the BML1 chamber, data from run 2102671.

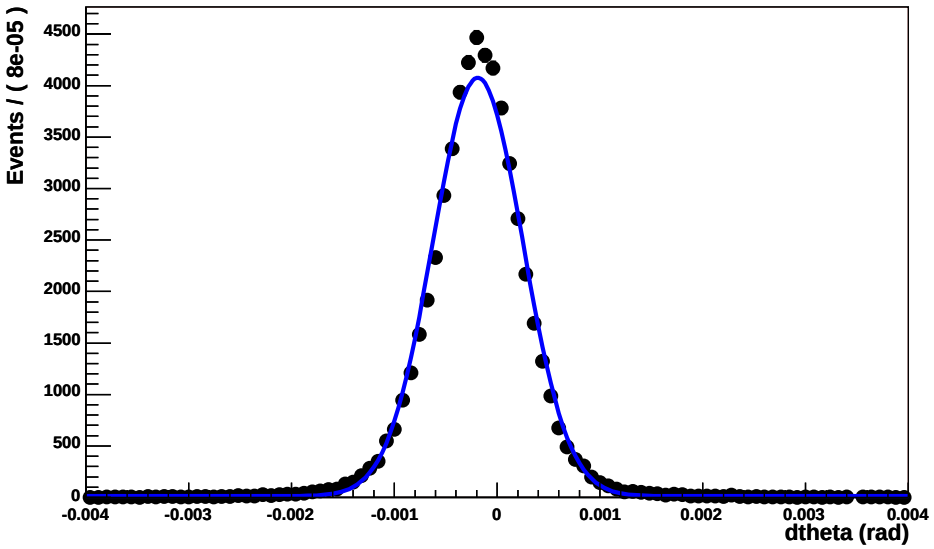

Figure 5.2: $\Delta\theta$ distribution for the BML1 chamber, data from run 2102675.

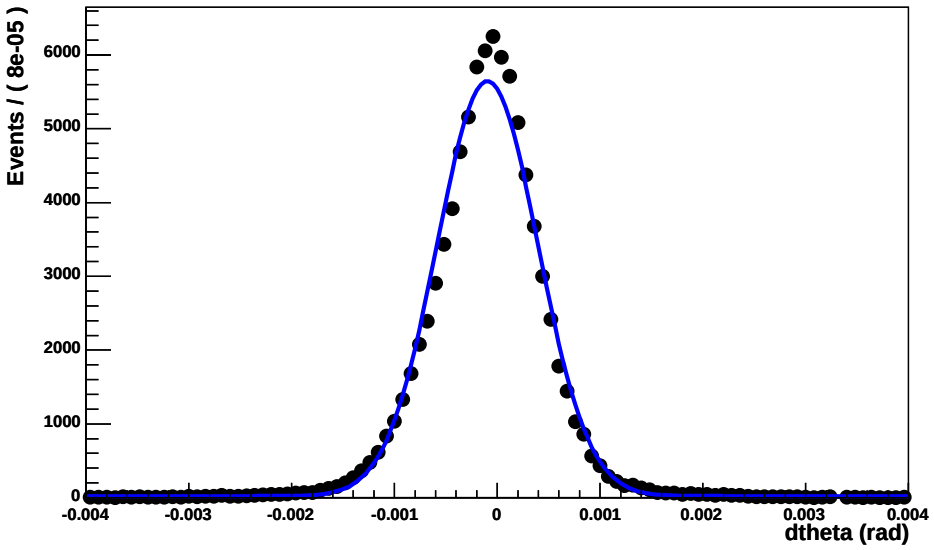

Figure 5.3: $\Delta\theta$ distribution for the BOL1 chamber, data from run 2102671.

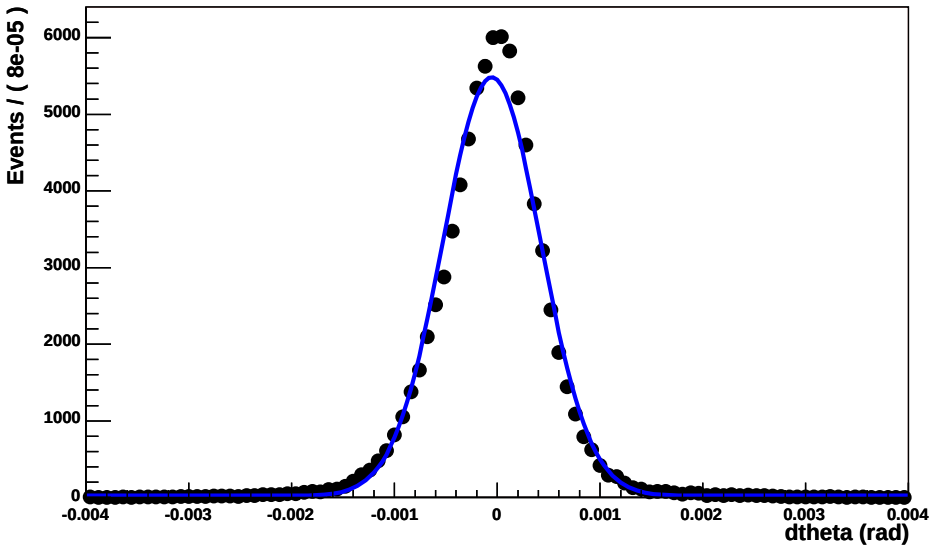

Figure 5.4: $\Delta\theta$ distribution for the BOL1 chamber, data from run 2102675.

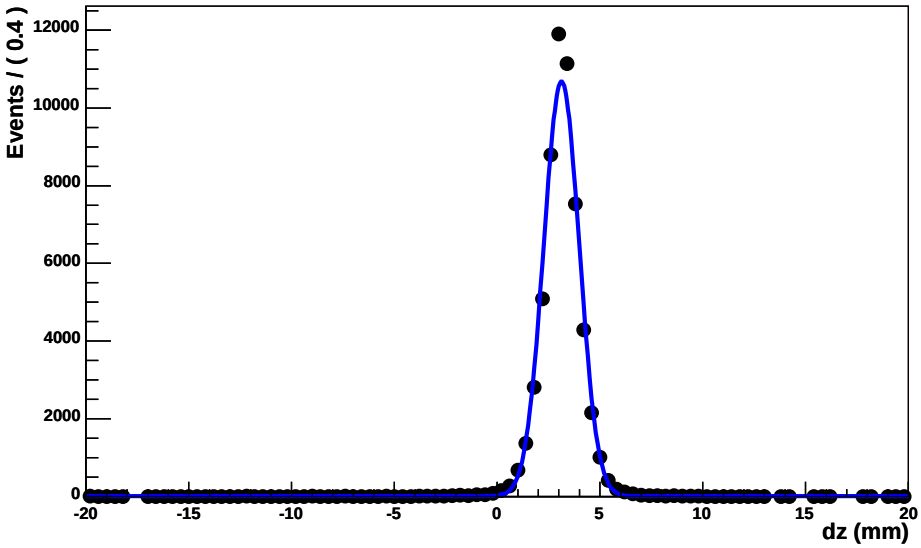

Figure 5.5: Δz distribution for the BML1 chamber, data from run 2102671.

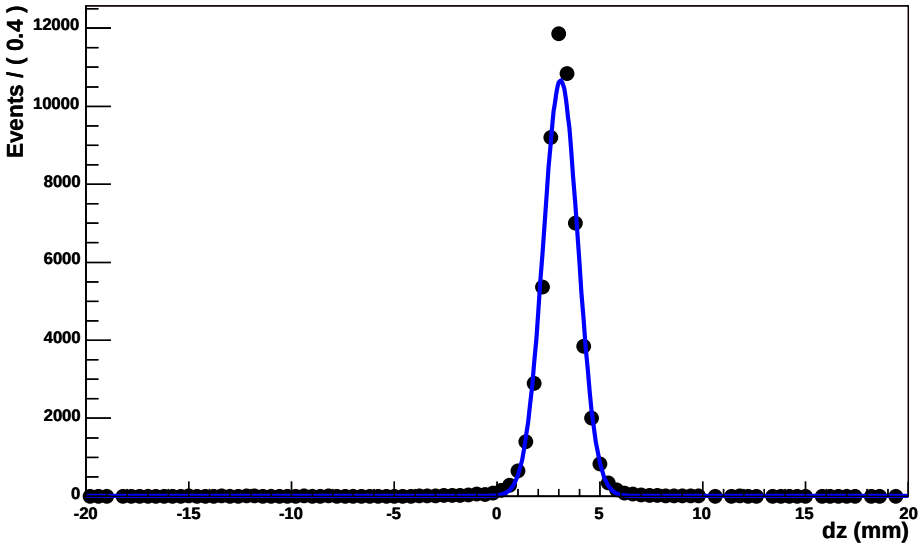

Figure 5.6: Δz distribution for the BML1 chamber, data from run 2102675.

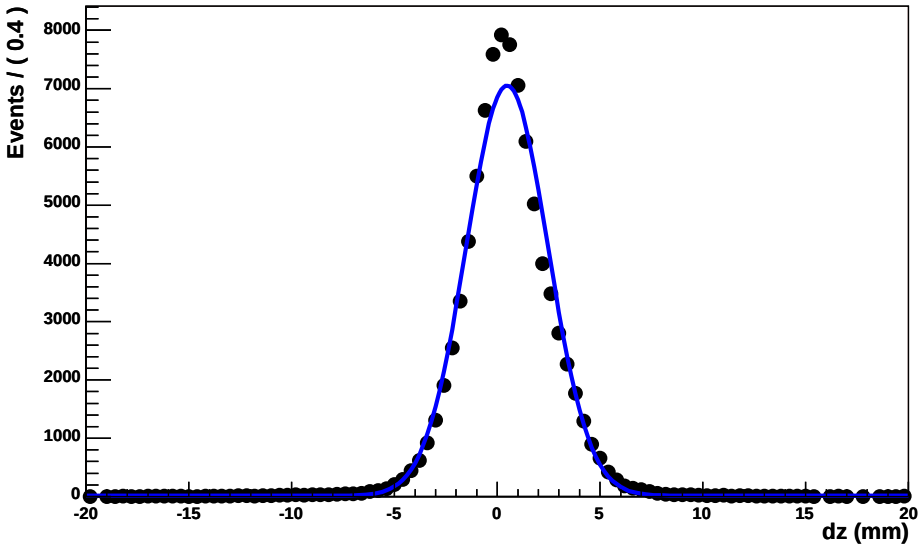

Figure 5.7: Δz distribution for the BOL1 chamber, data from run 2102671.

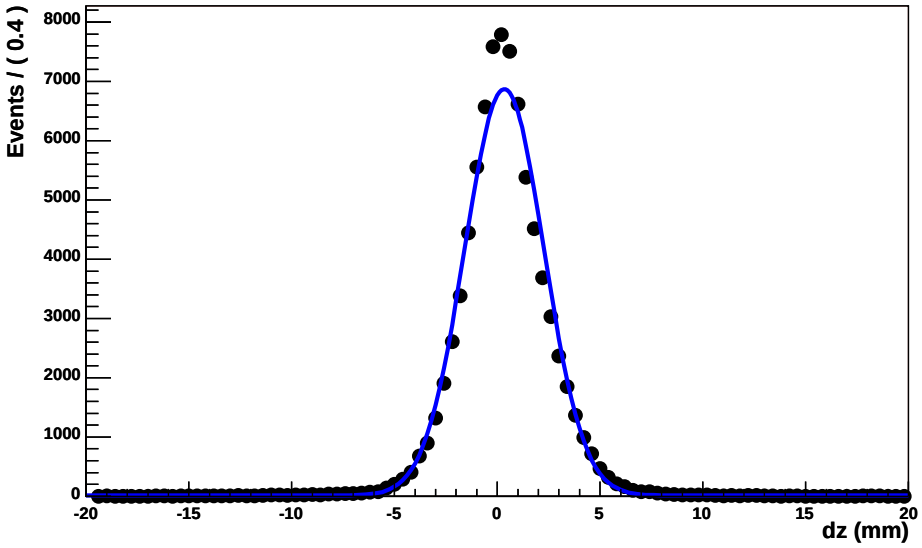

Figure 5.8: Δz distribution for the BOL1 chamber, data from run 2102675.

Chamber	Run	Δz (mm)	$\sigma_{\Delta z}$ (mm)
BML1	2102671	3.127 ± 0.004	$8.64 \pm 0.03 \times 10^{-1}$
	2102675	3.083 ± 0.004	$8.51 \pm 0.03 \times 10^{-1}$
BOL1	2102671	$4.77 \pm 0.07 \times 10^{-1}$	1.991 ± 0.005
	2102675	$3.41 \pm 0.07 \times 10^{-1}$	1.928 ± 0.005

Table 5.3: Displacements in z for the BML1 and BOL1 chamber.

The extrapolation of the segment in the reference chamber introduces a dependence on the angular resolution of the MDTs. To determine the positional resolution, we need to remove this dependence. This can be done using the events where a segment is present in all three chambers. By comparing the coordinate in the middle chamber with a line drawn from the inner to the outer chamber, we obtain a distribution for Δz without using the angle θ. This distribution is shown in figure 5.9. The corrections to the chamber positions found above have been applied, and the resulting estimates for Δz and $\sigma_{\Delta z}$ are:

$$\begin{aligned} \Delta z &= 1.203 \pm 0.004 \times 10^{-1}\,\text{mm} \\ \sigma_{\Delta z} &= 8.60 \pm 0.03 \times 10^{-2}\,\text{mm} \end{aligned}$$

The width of the distribution, $\sigma_{\Delta z}$, is decreased by a factor of 10.

If we denote the positional resolution of the chambers by σ_z and consider the three chambers to have the same resolution, we have:

$$\sigma_{\Delta z} = \sqrt{\sigma_{\text{line}}^2 + \sigma_z^2} = \sqrt{\left(\frac{\sigma_z}{2}\right)^2 + \left(\frac{\sigma_z}{2}\right)^2 + \sigma_z^2} = \sqrt{\frac{3}{2}}\sigma_z$$

Using the data from figure 5.9 this leads to a resolution of $\sigma_z = 70.2 \pm 0.3\,\mu\text{m}$. This is in reasonable agreement with the design resolution of $40\,\mu\text{m}$.

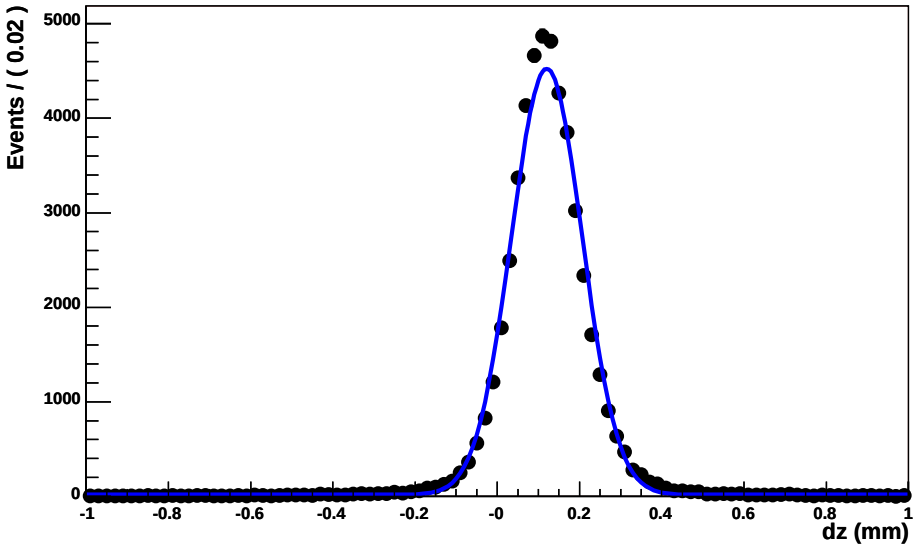

Figure 5.9: Δz distribution for the BML1 chamber when comparing the z coordinate with a line from the BIL1 to the BOL1 chamber. Run 2102671, 51,627 entries.

Chapter 6

Conclusions

The 2004 combined testbeam was the final testbeam before the start of ATLAS. The next stage will be the commissioning of the detector itself. This will be done by using cosmic particles traversing the ATLAS cavern.

Alignment of the different subdetectors will be one of the first issues that needs to be addressed. For the muon spectrometer, this will likely be done using the track segments and a procedure similar to the one described in this thesis.

Using the results obtained from the combined testbeam data, an estimate can be made for the amount of cosmic muons required to align the chambers. If we consider the accuracy of the optical alignment system as the minimal requirement, about 1,000 cosmic muons traversing a tower would suffice.

Also, the spatial and angular resolutions of the MDT chambers were obtained from the testbeam data. Considering the circumstances of the combined testbeam, where a lot of the conditions were far from fully optimized, the results presented here are remarkably close to the design values and show that the MDTs are likely to meet their requirements.

Appendix A

Glossary

Athena	The ATLAS software framework (see section 3.1)
ATLAS	A Toroidal LHC ApparatuS (see section 1.2)
BIL	Barrel Inner Large MDT chamber
BML	Barrel Middle Large MDT chamber
BOL	Barrel Outer Large MDT chamber
CSC	Cathode Strip Chamber (see section 2.4)
LHC	Large Hadron Collider (see section 1.1)
MDT	Monitored Drift Tube chamber (see section 2.1)
MOORE	Muon Object Oriented REconstruction (see section 3.1.2)
Python	A scripting language
ROOT	The analysis framework used by some of the programs
RPC	Resistive Plate Chamber (see section 2.2)
TGC	Thin Gap Chamber (see section 2.3)
XML	eXtensible Markup Language (see section 3.1.3)

Appendix B

Visualization sources

B.1 Geometry

B.1.1 MuonGeometryWriter.h

```
#ifndef JIVEXML_MUONGEOMETRYWRITER
#define JIVEXML_MUONGEOMETRYWRITER

#include <string>
#include <fstream>
#include "GaudiKernel/MsgStream.h"

#include "CLHEP/Geometry/Point3D.h"

class StoreGateSvc;

namespace MuonGM {
  class MuonDetectorManager;
  class MuonReadoutElement;
}

namespace JiveXML
{

  class MuonGeometryWriter {

    public:

      MuonGeometryWriter(StoreGateSvc *detStore, MsgStream &log) :
        m_detStore(detStore), m_log(log) {}

      void writeGeometry();

    private:

      const void writeHeader(std::ofstream &);
      const void writeMDTGeometry(std::ofstream &);
      const void writeCSCGeometry(std::ofstream &);
      const void writeRPCGeometry(std::ofstream &);
      const void writeTGCGeometry(std::ofstream &);
      const void writeFooter(std::ofstream &);

      const void writeElement(std::ofstream &, std::string, std::string, const MuonGM::MuonReadoutElement *);
      const void writeProjections(std::ofstream &, std::string, std::string, const std::vector<HepPoint3D> *);
      const double angle(HepPoint3D, HepPoint3D, std::string);
      const double distance2(HepPoint3D, HepPoint3D, std::string);

      StoreGateSvc *m_detStore;
      MsgStream m_log;
      const MuonGM::MuonDetectorManager *m_muon_manager;
  };
}
```

```
#endif
```

B.1.2 MuonGeometryWriter.cxx

```
#include "JiveXML/MuonGeometryWriter.h"

#include "StoreGate/StoreGateSvc.h"
#include "GaudiKernel/IToolSvc.h"
#include "GaudiKernel/MsgStream.h"

#include "MuonIdHelpers/MdtIdHelper.h"
#include "MuonIdHelpers/RpcIdHelper.h"
#include "MuonIdHelpers/TgcIdHelper.h"
#include "MuonIdHelpers/CscIdHelper.h"

#include "MuonGeoModel/MuonDetectorManager.h"
#include "MuonGeoModel/MdtReadoutElement.h"
#include "MuonGeoModel/RpcReadoutElement.h"
#include "MuonGeoModel/TgcReadoutElement.h"
#include "MuonGeoModel/CscReadoutElement.h"
#include "MuonGeoModel/RpcReadoutSet.h"

#include <fstream>
#include <sstream>

namespace JiveXML {

  void MuonGeometryWriter::writeGeometry() {

    std::ofstream outputFile("AMuonGeometryTB.xml");

    StatusCode sc = m_detStore->retrieve(m_muon_manager);
    if (sc.isFailure()) {
      m_log << MSG::ERROR << "Could not retrieve MuonGM::MuonDetectorManager" << endreq;
      m_muon_manager = 0;
    } else {
      writeHeader(outputFile);
      writeMDTGeometry(outputFile);
      writeCSCGeometry(outputFile);
      writeRPCGeometry(outputFile);
      writeTGCGeometry(outputFile);
      writeFooter(outputFile);
    }

    outputFile.close();
  }

  const void MuonGeometryWriter::writeHeader(std::ofstream &out) {

    std::string version = m_muon_manager->geometryVersion();

    out << "<?xml version=\"1.0\"?>"                                << std::endl
        << "<!DOCTYPE AMuonGeometry ["                              << std::endl
        << "<!ELEMENT AMuonGeometry (section)>"                     << std::endl
        << "<!ELEMENT section (AGeneralPath*)>"                     << std::endl
        << "<!ATTLIST section"                                      << std::endl
        << "          name        CDATA \" \""                      << std::endl
        << "          version     CDATA \" \""                      << std::endl
        << "          date        CDATA \" \""                      << std::endl
        << "          author      CDATA \" \""                      << std::endl
        << "          top_volume  CDATA \" \""                      << std::endl
        << "          DTD_version CDATA \" \">"                     << std::endl
        << "<!ELEMENT AGeneralPath (APoint*)>"                      << std::endl
        << "<!ATTLIST AGeneralPath"                                 << std::endl
        << "          p           CDATA     #REQUIRED"            << std::endl
        << "          c           CDATA     #REQUIRED"            << std::endl
        << "          n           CDATA     #REQUIRED"            << std::endl
        << "          numPoints   CDATA     #REQUIRED"            << std::endl
        << "          xR          (YES|NO)  \"NO\""               << std::endl
        << "          yR          (YES|NO)  \"NO\">"              << std::endl
        << "<!ELEMENT APoint EMPTY >"                               << std::endl
        << "<!ATTLIST APoint"                                       << std::endl
        << "          x           CDATA     #REQUIRED"            << std::endl
        << "          y           CDATA     #REQUIRED>"           << std::endl
        << "]>"                                                     << std::endl
        << "<AMuonGeometry>"                                        << std::endl
```

```
      << "<section name        = \"AGMU\""                      << std::endl
      << "         version     = \"" << version << "\""         << std::endl
      << "         date        = \"\""                          << std::endl
      << "         author      = \"Eric Jansen, Frans Crijns\"" << std::endl
      << "         top_volume  = \"\""                          << std::endl
      << "         DTD_version = \"v4\">"                       << std::endl;
}

const void MuonGeometryWriter::writeMDTGeometry(std::ofstream &out) {

  const MdtIdHelper *mdtIdHelper = m_muon_manager->mdtIdHelper();
  MdtIdHelper::const_id_iterator itr;

  for (itr=mdtIdHelper->module_begin(); itr!=mdtIdHelper->module_end(); ++itr) {

    /* MdtReadoutElements correspond to a set of multilayers. */
    Identifier parentId = *itr;
    std::string stationName = mdtIdHelper->stationNameString(mdtIdHelper->stationName(parentId));
    int mlMin = mdtIdHelper->multilayerMin(parentId);
    int mlMax = mdtIdHelper->multilayerMax(parentId);

    for (int ml=mlMin; ml<=mlMax; ml++) {
      Identifier channelId = mdtIdHelper->channelID(parentId, ml, 1, 1);
      const MuonGM::MdtReadoutElement *element = m_muon_manager->getMdtReadoutElement(channelId);
      writeElement(out, stationName, std::string("MDT"), element);
    }
  }
}

const void MuonGeometryWriter::writeCSCGeometry(std::ofstream &out) {

  const CscIdHelper *cscIdHelper = m_muon_manager->cscIdHelper();
  CscIdHelper::const_id_iterator itr;

  for (itr=cscIdHelper->module_begin(); itr!=cscIdHelper->module_end(); ++itr) {

    /* CscReadoutElements correspond to a set of chamberlayers. */
    Identifier parentId = *itr;
    std::string stationName = cscIdHelper->stationNameString(cscIdHelper->stationName(parentId));
    int clMin = cscIdHelper->chamberLayerMin(parentId);
    int clMax = cscIdHelper->chamberLayerMax(parentId);

    for (int cl=clMin; cl<=clMax; cl++) {
      Identifier channelId = cscIdHelper->channelID(parentId, cl, 1, 1, 1);
      const MuonGM::CscReadoutElement *element = m_muon_manager->getCscReadoutElement(channelId);
      writeElement(out, stationName, std::string("CSC"), element);
    }
  }
}

const void MuonGeometryWriter::writeRPCGeometry(std::ofstream &out) {

  const RpcIdHelper *rpcIdHelper = m_muon_manager->rpcIdHelper();
  RpcIdHelper::const_id_iterator itr;

  for (itr=rpcIdHelper->module_begin(); itr!=rpcIdHelper->module_end(); ++itr) {

    /* RpcReadoutElements correspond to a set of doubletZs, each of them consisting
     * of one or two doubletPhis.
     */
    Identifier parentId = *itr;
    std::string stationName = rpcIdHelper->stationNameString(rpcIdHelper->stationName(parentId));
    MuonGM::RpcReadoutSet set(m_muon_manager, parentId);

    for (int dbz=1; dbz<=set.NdoubletZ(); dbz++) {
      for (int dbp=1; dbp<=set.NPhimodules(dbz); dbp++) {
        const MuonGM::RpcReadoutElement *element = set.readoutElement(dbz, dbp);

        writeElement(out, stationName, std::string("RPC"), element);
      }
    }
  }
}

const void MuonGeometryWriter::writeTGCGeometry(std::ofstream &out) {

  const TgcIdHelper *tgcIdHelper = m_muon_manager->tgcIdHelper();
  TgcIdHelper::const_id_iterator itr;
```

```
  for (itr=tgcIdHelper->module_begin(); itr!=tgcIdHelper->module_end(); ++itr) {

    /* TgcReadoutElements correspond to the modules. */
    Identifier parentId = *itr;
    std::string stationName = tgcIdHelper->stationNameString(tgcIdHelper->stationName(parentId));
    const MuonGM::TgcReadoutElement *element = m_muon_manager->getTgcReadoutElement(parentId);

    writeElement(out, stationName, std::string("TGC"), element);
  }
}

const void MuonGeometryWriter::writeElement(std::ofstream &out, std::string stationName,
                                            std::string technology, const MuonGM::MuonReadoutElement *element) {

  if (!element) return;

  std::vector<HepPoint3D> *points = new std::vector<HepPoint3D>;
  HepTransform3D transform = element->transform();
  double halfShortWidth = element->getSsize()/2.;
  double halfLongWidth = element->getLongSsize()/2.;
  double halfThickness = element->barrel() ? element->getRsize()/2. : element->getZsize()/2.;
  double halfLength = element->barrel() ? element->getZsize()/2. : element->getRsize()/2.;

  /* The eight corners of this chamber in global coordinates: */
  points->push_back(transform * HepPoint3D( halfThickness, -halfLongWidth,   halfLength));
  points->push_back(transform * HepPoint3D( halfThickness,  halfLongWidth,   halfLength));
  points->push_back(transform * HepPoint3D( halfThickness, -halfShortWidth, -halfLength));
  points->push_back(transform * HepPoint3D( halfThickness,  halfShortWidth, -halfLength));

  points->push_back(transform * HepPoint3D(-halfThickness, -halfLongWidth,   halfLength));
  points->push_back(transform * HepPoint3D(-halfThickness,  halfLongWidth,   halfLength));
  points->push_back(transform * HepPoint3D(-halfThickness, -halfShortWidth, -halfLength));
  points->push_back(transform * HepPoint3D(-halfThickness,  halfShortWidth, -halfLength));

  /* And some points along the side to correct the RZ-projection: */
  points->push_back(transform * HepPoint3D( halfThickness,  0.0,                                  halfLength));
  points->push_back(transform * HepPoint3D( halfThickness,  0.0,                                 -halfLength));
  points->push_back(transform * HepPoint3D( halfThickness,  (halfLongWidth + halfShortWidth)/2., 0.0));
  points->push_back(transform * HepPoint3D( halfThickness, -(halfLongWidth + halfShortWidth)/2., 0.0));
  points->push_back(transform * HepPoint3D(-halfThickness,  0.0,                                  halfLength));
  points->push_back(transform * HepPoint3D(-halfThickness,  0.0,                                 -halfLength));
  points->push_back(transform * HepPoint3D(-halfThickness,  (halfLongWidth + halfShortWidth)/2., 0.0));
  points->push_back(transform * HepPoint3D(-halfThickness, -(halfLongWidth + halfShortWidth)/2., 0.0));

  writeProjections(out, technology, stationName, points);
}

/* This uses the package/gift wrapping algorithm to find the convex hull of this set of points */
const void MuonGeometryWriter::writeProjections(std::ofstream &out, std::string technology,
                                                std::string stationName, const std::vector<HepPoint3D> *points) {

  std::vector<HepPoint3D> *outline = new std::vector<HepPoint3D>;
  std::vector<HepPoint3D>::const_iterator itr;
  HepPoint3D minPoint;
  bool closedPath;
  double searchAngle;

  /* YX projection */
  outline->push_back(*points->begin());
  for (itr=points->begin()+1; itr!=points->end(); ++itr) {

    if (itr->y() < outline->begin()->y()
        || itr->y() == outline->begin()->y() && itr->x() < outline->begin()->x()) {
      outline->pop_back();
      outline->push_back(*itr);
    }
  }

  closedPath = false;
  searchAngle = 0.0;
  while (!closedPath) {

    double minAngle = 2.01*M_PI;
    double maxDistance = 0.0;
    for (itr=points->begin(); itr!=points->end(); ++itr) {

      double curDistance = distance2(*(outline->end()-1), *itr, "YX");
```

```
    if (curDistance > 0.0) {

      double curAngle = angle(*(outline->end()-1), *itr, "YX");
      if (curAngle >= searchAngle && curAngle < minAngle
          || curAngle == minAngle && curDistance > maxDistance) {
        minAngle = curAngle;
        maxDistance = curDistance;
        minPoint = *itr;
      }
    }
  }

  if (minAngle > 2.0*M_PI || distance2(*outline->begin(), minPoint, "YX") == 0.0) {
    closedPath = true;
  } else {
    outline->push_back(minPoint);
    searchAngle = minAngle;
  }
}

/* And finally output the AGeneralPath element. */
out << "<AGeneralPath p=\"YX\" c=\"" << technology << "\""
    << " n=\"" << technology << "_" << stationName << "\""
    << " numPoints=\"" << outline->size() << "\">" << std::endl;

for (itr=outline->begin(); itr!=outline->end(); ++itr) {

  out << "  <APoint x=\"" << itr->x()/10. << "\""
      << " y=\"" << itr->y()/10. << "\""
      << " />" << std::endl;
}

out << "</AGeneralPath>" << std::endl;

outline->clear();

/* RZ projection */
outline->push_back(*points->begin());
for (itr=points->begin()+1; itr!=points->end(); ++itr) {

  if (itr->perp() < outline->begin()->perp()
      || itr->perp() == outline->begin()->perp() && itr->z() < outline->begin()->z()) {
    outline->pop_back();
    outline->push_back(*itr);
  }
}

closedPath = false;
searchAngle = 0.0;
while (!closedPath) {

  double minAngle = 2.01*M_PI;
  double maxDistance = 0.0;
  for (itr=points->begin(); itr!=points->end(); ++itr) {

    double curDistance = distance2(*(outline->end()-1), *itr, "RZ");
    if (curDistance > 0.0) {

      double curAngle = angle(*(outline->end()-1), *itr, "RZ");
      if (curAngle >= searchAngle && curAngle < minAngle
          || curAngle == minAngle && curDistance > maxDistance) {
        minAngle = curAngle;
        maxDistance = curDistance;
        minPoint = *itr;
      }
    }
  }

  if (minAngle > 2.0*M_PI || distance2(*outline->begin(), minPoint, "RZ") == 0.0) {
    closedPath = true;
  } else {
    outline->push_back(minPoint);
    searchAngle = minAngle;
  }
}

/* And finally output the AGeneralPath element. */
out << "<AGeneralPath p=\"RZ\" c=\"" << technology << "\""
```

```
      << " n=\"" << technology << "_" << stationName << "\""
      << " numPoints=\"" << outline->size() << "\">" << std::endl;

  for (itr=outline->begin(); itr!=outline->end(); ++itr) {

    out << "  <APoint x=\"" << itr->z()/10. << "\""
        << " y=\"" << itr->perp()/10. << "\""
        << " />" << std::endl;
  }

  out << "</AGeneralPath>" << std::endl;

  delete outline;
}

const double MuonGeometryWriter::angle(HepPoint3D p1, HepPoint3D p2, std::string projection) {

  double dx, dy, angle;
  if (projection == "YX") {
    dx = p2.x() - p1.x();
    dy = p2.y() - p1.y();
  } else {
    dx = p2.z() - p1.z();
    dy = p2.perp() - p1.perp();
  }

  angle = atan(fabs(dy/dx));

  if (dx > 0.0 && dy == 0.0) {
    /* Positive x-axis */
    return 0.0;
  } else if (dx > 0.0 && dy > 0.0) {
    /* First quadrant */
    return angle;
  } else if (dx == 0.0 && dy > 0.0) {
    /* Positive y-axis */
    return 0.5*M_PI;
  } else if (dx < 0.0 && dy > 0.0) {
    /* Second quadrant */
    return M_PI - angle;
  } else if (dx < 0.0 && dy == 0.0) {
    /* Negative x-axis */
    return M_PI;
  } else if (dx < 0.0 && dy < 0.0) {
    /* Third quadrant */
    return M_PI + angle;
  } else if (dx == 0.0 && dy < 0.0) {
    /* Negative y-axis */
    return 1.5*M_PI;
  } else if (dx > 0.0 && dy < 0.0) {
    /* Fourth quadrant */
    return 2.0*M_PI - angle;
  } else {
    printf("This should not happen\n");
    exit(1);
  }
}

const double MuonGeometryWriter::distance2(HepPoint3D p1, HepPoint3D p2, std::string projection) {

  double dx, dy;
  if (projection == "YX") {
    dx = p2.x() - p1.x();
    dy = p2.y() - p1.y();
  } else {
    dx = p2.z() - p1.z();
    dy = p2.perp() - p1.perp();
  }

  return dx*dx + dy*dy;
}

const void MuonGeometryWriter::writeFooter(std::ofstream &out) {

  out << "</section>" << std::endl
      << "</AMuonGeometry>" << std::endl;

}
```

```
}
```

B.2 r-t relation of the MDT tubes

B.2.1 MDTDigitRetriever.cxx

```
#include "JiveXML/MDTDigitRetriever.h"
#include <string>
#include <vector>
#include <map>
#include "GaudiKernel/MsgStream.h"
#include "GaudiKernel/Bootstrap.h"
#include "GaudiKernel/ISvcLocator.h"
#include "StoreGate/StoreGateSvc.h"
#include "JiveXML/ReserveSpace.h"
#include "JiveXML/MuonFullIDHelper.h"

#include "MuonDigitContainer/MdtDigitContainer.h"
#include "MuonDigitContainer/MdtDigitCollection.h"
#include "MuonDigitContainer/G3MdtMCTruthInfo.h"
#include "CLHEP/Geometry/Point3D.h"

#include "MuonCalibEvent/MdtCalibHit.h"
#include "MdtCalibSvc/MdtCalibrationSvc.h"
#include "MuonTBConditions/MuonTBDigitToHitSvc.h"

#include "MuonGeoModel/MuonDetectorManager.h"
#include "MuonGeoModel/MdtReadoutElement.h"
#include "MuonIdHelpers/MdtIdHelper.h"
#include "HepMC/GenParticle.h"

namespace JiveXML {

  MDTDigitRetriever::MDTDigitRetriever(StoreGateSvc* storeGate,StoreGateSvc* detStore, std::string sgKey, bool testbeam) :
    DataRetriever("MDT"),m_storeGate(storeGate),m_detStore(detStore),m_sgKey(sgKey),m_testbeam(testbeam){
    m_detStore->retrieve( m_muon_mgr );
    m_muonFullIDHelper = new MuonFullIDHelper();
  }
  MDTDigitRetriever::~MDTDigitRetriever(){
    if(m_muonFullIDHelper){delete m_muonFullIDHelper;}
  }

    std::map<std::string,std::vector<WriteType> >* MDTDigitRetriever::retrieve(MsgStream& log,
                                                std::vector<HepMC::GenParticle*>& particles){
      log << MSG::DEBUG << "Retrieving " << m_type <<endreq;
      (*m_map)["x"] = std::vector<WriteType>();
      (*m_map)["y"] = std::vector<WriteType>();
      (*m_map)["z"] = std::vector<WriteType>();
      (*m_map)["driftR"] = std::vector<WriteType>();
      (*m_map)["length"] = std::vector<WriteType>();
      (*m_map)["identifier"] = std::vector<WriteType>();
      (*m_map)["id"] = std::vector<WriteType>();
      (*m_map)["barcode"] = std::vector<WriteType>();

      typedef MdtDigitContainer::const_iterator collection_iterator;
      typedef MdtDigitCollection::const_iterator digit_iterator;
      typedef std::map<Identifier, G3MdtMCTruthInfo*> mdtTruth;
      const mdtTruth* mdtIdMap;
      const MdtDigitContainer* container;
      const MdtIdHelper* mdtHelper;

      ISvcLocator *svcLocator = Gaudi::svcLocator();
      MuonTBDigitToHitSvc* mdtDigitToHitSvc;
      MdtCalibrationSvc* mdtCalibrationSvc;

      if (!m_storeGate->retrieve(container,m_sgKey)) {
          log << MSG::WARNING << "Cannot retrieve MDT Container" << endreq;

      } else if (!m_muon_mgr) {
          log << MSG::ERROR << "MuonDetectorManager not available" << endreq;

      } else if (!(mdtHelper = m_muon_mgr->mdtIdHelper())) {
          log << MSG::ERROR << "Cannot retrieve MdtIdHelper" << endreq;
```

```
} else if (!m_testbeam && !svcLocator->service("MdtCalibrationSvc", mdtCalibrationSvc).isSuccess()) {
    log << MSG::ERROR << "Cannot initialize MdtCalibrationSvc" << endreq;

} else if (m_testbeam && !svcLocator->service("MuonTBDigitToHitSvc", mdtDigitToHitSvc).isSuccess()) {
    log << MSG::ERROR << "Cannot initialize MuonTBDigitToHitSvc" << endreq;

} else {

    if (!m_storeGate->retrieve(mdtIdMap,"MdtMap")) {
      log << MSG::WARNING << "Cannot retrieve MdtMCTruthInfo" << endreq;
      mdtIdMap = 0;
    }

    collection_iterator itr;
    //Reserve some space in the vectors
    int count = 0;
    for (itr=container->begin(); itr!=container->end(); ++itr) {
      count += (*itr)->size();
    }
    ReserveSpace(count,m_map,log);

    for (itr=container->begin(); itr!=container->end(); ++itr) {

      const MdtDigitCollection* mdtCollection = *itr;
      Identifier moduleId = mdtCollection->identify();

      if (!mdtHelper->validElement(moduleId)) {
        log << MSG::ERROR << "Invalid MDT module identifier " << mdtHelper->show_to_string(moduleId) << endreq;

      } else {
        digit_iterator dig_itr;
        for (dig_itr=mdtCollection->begin(); dig_itr!=mdtCollection->end(); ++dig_itr) {
          const MdtDigit* mdtDigit = *dig_itr;
          const MuonGM::MdtReadoutElement* mdt_element;
          Identifier dig_id = mdtDigit->identify();

          if (!(mdt_element = m_muon_mgr->getMdtReadoutElement(dig_id))) {
               log << MSG::ERROR << "Cannot retrieve MdtReadoutElement for identifier "
                << mdtHelper->show_to_string(dig_id) << endreq;

          } else {

            /** @todo fix mdt kine */
            //int kine =  mdtDigit->getParticle();//This is naughty and should not be used.
            int kine = 0;
            int barcode = 0;
            HepPoint3D tube_pos;

            if (!mdtIdMap) {
              tube_pos = mdt_element->tubePos(dig_id);
            } else {
              G3MdtMCTruthInfo* mdtInfo = (mdtIdMap->find(dig_id))->second;
              tube_pos = mdtInfo->getGlobalPoint();
              kine = mdtInfo->getTrack();
            }

            float driftR;
            if (m_testbeam) {
              MooMdtHit mooMdtHit = MooMdtHit(mdtDigit, tube_pos, mdt_element, 0.0);
              mdtDigitToHitSvc->DriftDistanceAndError(&mooMdtHit, tube_pos.mag());
              driftR = fabs(mooMdtHit.SignedDrift());
            } else {
              MdtCalibHit mdtCalibHit = MdtCalibHit(mdtDigit, tube_pos, mdt_element);
              mdtCalibrationSvc->driftRadiusFromTime(mdtCalibHit, tube_pos.mag());
              driftR = fabs(mdtCalibHit.driftRadius());
            }

            //calculate active tube length
            float active_tube_length = mdt_element->getActiveTubeLength(mdtHelper->tubeLayer(dig_id),
                                                                        mdtHelper->tube(dig_id));

            (*m_map)["x"].push_back(WriteType(tube_pos.x() /10.));
            (*m_map)["y"].push_back(WriteType(tube_pos.y() /10.));
            (*m_map)["z"].push_back(WriteType(tube_pos.z() /10.));
            (*m_map)["driftR"].push_back(WriteType(driftR /10.));
            (*m_map)["length"].push_back(WriteType(active_tube_length /10.));

            if (kine > 0 && kine <= int(particles.size())){
```

```
                  barcode = particles[kine-1]->barcode();
                }
                (*m_map)["barcode"].push_back(WriteType(barcode));

                //Make identifer
                (*m_map)["identifier"].push_back(WriteType(m_muonFullIDHelper->getFullID(dig_id,mdtHelper)));

                (*m_map)["id"].push_back(WriteType(dig_id.get_compact()));

              }
            }
          }
        }
      }

      log << MSG::DEBUG << m_type<<": " << (*m_map)["x"].size() << endreq;

      return m_map;
    }
}
```

B.3 Muon truth tracks

B.3.1 TruthMuonTrackRetriever.h

```
#ifndef JIVEXML_TRUTHMUONTRACKRETRIEVER_H
#define JIVEXML_TRUTHMUONTRACKRETRIEVER_H

#include <string>
#include <vector>
#include <map>
#include "JiveXML/IDataRetriever.h"
#include "JiveXML/WriteType.h"
#include "CLHEP/HepPDT/ParticleDataTable.hh"

class MsgStream;
class StoreGateSvc;
namespace HepMC{
  class GenParticle;
}
namespace JiveXML{


class TruthMuonTrackRetriever : public IDataRetriever {
 public:

  TruthMuonTrackRetriever(StoreGateSvc*);
  virtual std::map<std::string,std::vector<WriteType> >* retrieve(MsgStream&,std::vector<HepMC::GenParticle*>&);
  virtual ~TruthMuonTrackRetriever();
  virtual std::string type(){return m_type;}
  virtual void clear();
  private:
  std::string m_type;
  std::map<std::string,std::vector<WriteType> >* m_map;
  StoreGateSvc *m_storeGate;
  HepPDT::ParticleDataTable *m_pdt;
};

}
#endif
```

B.3.2 TruthMuonTrackRetriever.cxx

```
#include "JiveXML/TruthMuonTrackRetriever.h"
#include <string>
#include <vector>
#include <map>
#include "GaudiKernel/MsgStream.h"
#include "GaudiKernel/Bootstrap.h"
#include "GaudiKernel/ISvcLocator.h"
#include "StoreGate/StoreGateSvc.h"
```

```
#include "HepMC/GenParticle.h"
#include "TrackRecord/TrackRecord.h"
#include "TrackRecord/TrackRecordCollection.h"
#include "PartPropSvc/PartPropSvc.h"
#include "CLHEP/HepPDT/ParticleData.hh"
#include "CLHEP/Geometry/Point3D.h"
#include "CLHEP/Geometry/Vector3D.h"

namespace JiveXML {

  TruthMuonTrackRetriever::TruthMuonTrackRetriever(StoreGateSvc *storeGate) :
    IDataRetriever(),m_type("SMTr"),m_storeGate(storeGate){
    m_map = new std::map<std::string,std::vector<WriteType> >;

    // get the particle data table for determining the charge
    ISvcLocator *svcLocator = Gaudi::svcLocator();
    IPartPropSvc *partPropSvc;
    if (svcLocator->service("PartPropSvc", partPropSvc).isSuccess()) {
      m_pdt = partPropSvc->PDT();
    } else {
      m_pdt = 0;
    }
  }

  TruthMuonTrackRetriever::~TruthMuonTrackRetriever(){if(m_map)delete m_map;}

  void TruthMuonTrackRetriever::clear(){
    //get back all that vector capacity that has been used!
    std::map<std::string,std::vector<WriteType> >::iterator itr = m_map->begin();
    for(; itr != m_map->end() ; ++itr){
      (*itr).second.erase((*itr).second.begin(),(*itr).second.end());
      std::vector<WriteType>((*itr).second).swap((*itr).second);
    }
    m_map->clear();

  }

  std::map<std::string,std::vector<WriteType> >* TruthMuonTrackRetriever::retrieve(MsgStream& log,
                                                  std::vector<HepMC::GenParticle*>& particles){
    log << MSG::DEBUG << "Retrieving " << m_type <<endreq;

    (*m_map)["pt"] = std::vector<WriteType>();
    (*m_map)["phi"] = std::vector<WriteType>();
    (*m_map)["eta"] = std::vector<WriteType>();
    (*m_map)["rhoVertex"] = std::vector<WriteType>();
    (*m_map)["phiVertex"] = std::vector<WriteType>();
    (*m_map)["zVertex"] = std::vector<WriteType>();
    (*m_map)["code"] = std::vector<WriteType>();
    (*m_map)["id"] = std::vector<WriteType>();

    const TrackRecordCollection* recordCollection;
    if (m_storeGate->retrieve(recordCollection, "MuonEntryRecord") == StatusCode::SUCCESS) {
      TrackRecordCollection::const_iterator record;

      if (!m_pdt) {
        log << MSG::ERROR << "No PartPropSvc/ParticleDataTable available."
                          << " Simulated muon tracks will not be written for other charged particles."
                          << endreq;
      }

      for (record=recordCollection->begin(); record!=recordCollection->end(); record++) {

        int code = (*record)->GetPDGCode();
        log << MSG::DEBUG << "Particle " << code << endreq;
        if (m_pdt) {
          HepPDT::ParticleData *particle = m_pdt->particle(fabs(code));
          if (!particle || !particle->charge()) continue;
        } else {
          if (fabs(code) != 13) continue;
        }

        HepPoint3D vertex = (*record)->GetPosition();
        HepVector3D momentum = (*record)->GetMomentum();

        (*m_map)["pt"].push_back(WriteType( momentum.perp()/1000. ));
        (*m_map)["phi"].push_back(WriteType( momentum.phi() < 0 ? momentum.phi() + 2*M_PI : momentum.phi() ));
        (*m_map)["eta"].push_back(WriteType( momentum.pseudoRapidity() ));
```

```
        (*m_map)["rhoVertex"].push_back(WriteType( vertex.perp()/10. ));
        (*m_map)["phiVertex"].push_back(WriteType( vertex.phi() < 0 ? vertex.phi() + 2*M_PI : vertex.phi() ));
        (*m_map)["zVertex"].push_back(WriteType( vertex.z()/10. ));
        (*m_map)["code"].push_back(WriteType( code ));
        (*m_map)["id"].push_back(WriteType( 0 ));
      }

    } else {
      log << MSG::ERROR << "MuonEntryRecord not found" << endreq;
    }

    log << MSG::DEBUG
        << m_type << ": "<< (*m_map)["pt"].size()
        << endreq;

    return m_map;
  }
}
```

B.4 MDT Segments

B.4.1 MDTSegmentRetriever.h

```
#ifndef JIVEXML_MDTSEGMENTRETRIEVER_H
#define JIVEXML_MDTSEGMENTRETRIEVER_H

#include <string>
#include <vector>
#include <map>

#include "JiveXML/DataRetriever.h"
#include "JiveXML/WriteType.h"

class StoreGateSvc;
class MsgStream;
namespace HepMC{
  class GenParticle;
}
namespace JiveXML{

class MDTSegmentRetriever : public DataRetriever {
 public:

  MDTSegmentRetriever(StoreGateSvc* storeGate, std::string mdtSegments_key) :
    DataRetriever("MSeg"),m_storeGate(storeGate),m_mdtSegments_key(mdtSegments_key){}

  virtual std::map<std::string,std::vector<WriteType> >* retrieve(MsgStream&,std::vector<HepMC::GenParticle*>&);
  virtual ~MDTSegmentRetriever(){};
 private:
  StoreGateSvc* m_storeGate;
  std::string m_mdtSegments_key;
};

}
#endif
```

B.4.2 MDTSegmentRetriever.cxx

```
#include "JiveXML/MDTSegmentRetriever.h"
#include <string>
#include <vector>
#include <map>
#include "GaudiKernel/MsgStream.h"
#include "StoreGate/StoreGateSvc.h"
#include "JiveXML/ReserveSpace.h"
#include "MooEvent/MooMdtSegment.h"
#include "MooEvent/MooMdtHit.h"
#include "MooEvent/MdtSegmentContainer.h"
#include "HepMC/GenParticle.h"

namespace JiveXML {
```

```
  std::map<std::string,std::vector<WriteType> >* MDTSegmentRetriever::retrieve(MsgStream& log,
                                                 std::vector<HepMC::GenParticle*>& particles) {

    log << MSG::DEBUG << "Retrieving " << m_type <<endreq;

    (*m_map)["x"] = std::vector<WriteType>();
    (*m_map)["y"] = std::vector<WriteType>();
    (*m_map)["z"] = std::vector<WriteType>();
    (*m_map)["cotTheta"] = std::vector<WriteType>();
    (*m_map)["numHits"] = std::vector<WriteType>();
    (*m_map)["phi0"] = std::vector<WriteType>();

    const MdtSegmentContainer *container;
    StatusCode sc = m_storeGate->retrieve(container, m_mdtSegments_key);

    if (sc.isFailure()) {
      log << MSG::ERROR << "Cannot retrieve MDT segment container" << endreq;
    } else {
      MdtSegmentContainer::const_iterator seg_itr;
      int numSegs = container->size();
      int numHits = 0;

      for (seg_itr=container->begin(); seg_itr!=container->end(); ++seg_itr) {
        MooMdtSegment *segment = *seg_itr;
        numHits += segment->hits_count();
      }

      std::string hits = "hits multiple=\"" + numberToString((float) numHits/numSegs) + "\"";

      (*m_map)[hits] = std::vector<WriteType>();
      ReserveSpace(numSegs, m_map, log);
      std::map<std::string,std::vector<WriteType> >::iterator temp = m_map->find(hits);
      ReserveSpace(numHits, temp, log);

      for (seg_itr=container->begin(); seg_itr!=container->end(); ++seg_itr) {
        MooMdtSegment *segment = *seg_itr;
        std::vector<MooMdtHit*>::const_iterator hit_itr;
        double phi = 0.0, rho = 0.0, z = 0.0;

        for (hit_itr=segment->hits_begin(); hit_itr!=segment->hits_end(); ++hit_itr) {
          MooMdtHit *hit = *hit_itr;
          (*m_map)[hits].push_back(WriteType(hit->HitId().get_compact()));

          phi += hit->position().phi() / segment->hits_count();
          rho += hit->position().perp() / segment->hits_count();
          z += hit->position().z() / segment->hits_count();
        }

        /* Adjust the average of the hit positions to be on the segment. */
        if (fabs(segment->cot_theta()) > fabs(segment->tan_theta())) {
          rho = segment->rho() + segment->tan_theta() * (z - segment->z());
        } else {
          z = segment->z() + segment->cot_theta() * (rho - segment->rho());
        }

        (*m_map)["x"].push_back(WriteType(cos(phi)*rho/10.));
        (*m_map)["y"].push_back(WriteType(sin(phi)*rho/10.));
        (*m_map)["z"].push_back(WriteType(z/10.));
        (*m_map)["cotTheta"].push_back(WriteType(segment->cot_theta()));
        (*m_map)["numHits"].push_back(WriteType(segment->hits_count()));
        (*m_map)["phi0"].push_back(WriteType(phi));
      }
    }

    log << MSG::DEBUG << m_type << ": " << (*m_map)["cotTheta"].size() << endreq;

    return m_map;
  }
}
```

Appendix C

Alignment sources

C.1 Athena algorithm

C.1.1 MakeSegmentNTuple.h

```
#include "GaudiKernel/Algorithm.h"
#include "GaudiKernel/NTuple.h"

#include <string>

class StoreGateSvc;
class MdtIdHelper;
class MdtCalibrationSvc;

namespace MuonGM {
  class MuonDetectorManager;
}

class MakeSegmentNTuple : public Algorithm {

  public:
    MakeSegmentNTuple(const std::string &, ISvcLocator *);
    StatusCode initialize();
    StatusCode execute();
    StatusCode finalize();

  private:
    StoreGateSvc *p_storeGate, *p_detectorStore;
    MdtCalibrationSvc *p_calibrationSvc;
    const MuonGM::MuonDetectorManager *p_muonDetectorManager;
    const MdtIdHelper *p_mdtIdHelper;

    std::string m_nTupleName;
    bool m_writeRecords;

    NTuple::Tuple* p_nTuple;
    NTuple::Item<long> m_numSegs;
    NTuple::Item<long> m_event;
    NTuple::Array<long> m_chamber;
    NTuple::Array<long> m_numHitsL1;
    NTuple::Array<long> m_numHitsL2;
    NTuple::Array<double> m_z;
    NTuple::Array<double> m_rho;
    NTuple::Array<double> m_theta;
    NTuple::Array<double> m_thetaError;
    NTuple::Array<double> m_betaError;
    NTuple::Array<double> m_chi2;
};
```

C.1.2 MakeSegmentNTuple.cxx

```
#include "MuonTBAlignment/MakeSegmentNTuple.h"
#include "StoreGate/StoreGateSvc.h"
#include "StoreGate/DataHandle.h"
#include "DataModel/DataVector.h"
#include "DataModel/ElementLink.h"

#include "EventInfo/EventInfo.h"
#include "EventInfo/EventID.h"

#include "GaudiKernel/MsgStream.h"
#include "GaudiKernel/ISvcLocator.h"
#include "GaudiKernel/INTupleSvc.h"
#include "GaudiKernel/NTuple.h"
#include "GaudiKernel/SmartDataPtr.h"

#include "MuonDigitContainer/MdtDigitContainer.h"
#include "MuonDigitContainer/MdtDigitCollection.h"

#include "MooEvent/MooMdtSegment.h"
#include "MooEvent/MooMdtHit.h"
#include "MuonTBData/MdtSegmentContainer.h"

#include "MuonGeoModel/MuonDetectorManager.h"
#include "MuonGeoModel/MdtReadoutElement.h"

#include "CLHEP/Geometry/Point3D.h"

MakeSegmentNTuple::MakeSegmentNTuple(const std::string& name, ISvcLocator* pSvcLocator)
  : Algorithm(name, pSvcLocator), m_nTupleName("/NTUPLES/FILE1/TB/tree") {

  declareProperty("nTupleName", m_nTupleName);
}

StatusCode MakeSegmentNTuple::initialize() {
  MsgStream log(msgSvc(), name());
  StatusCode sc;

  if (!(p_nTuple = NTuplePtr(ntupleSvc(), m_nTupleName))) {
    p_nTuple = ntupleSvc()->book(m_nTupleName, CLID_ColumnWiseTuple, "MuonTBAlignment data");
    m_writeRecords = true;
  } else {
    m_writeRecords = false;
  }

  if (p_nTuple) {
    p_nTuple->addItem("numsegs", m_numSegs, 0, 15);
    p_nTuple->addItem("event", m_event, 0, 1);
    p_nTuple->addItem("chamber", m_numSegs, m_chamber);
    p_nTuple->addItem("numhitsl1", m_numSegs, m_numHitsL1);
    p_nTuple->addItem("numhitsl2", m_numSegs, m_numHitsL2);
    p_nTuple->addItem("z", m_numSegs, m_z);
    p_nTuple->addItem("rho", m_numSegs, m_rho);
    p_nTuple->addItem("theta", m_numSegs, m_theta);
    p_nTuple->addItem("thetaerror", m_numSegs, m_thetaError);
    p_nTuple->addItem("betaerror", m_numSegs, m_betaError);
    p_nTuple->addItem("chi2", m_numSegs, m_chi2);
  } else {
    log << MSG::ERROR << "Could not access NTuple " << m_nTupleName << endreq;
  }

  if (!service("StoreGateSvc", p_storeGate)) {
    log << MSG::ERROR << "Could not locate StoreGate service" << endreq;
    return StatusCode::FAILURE;
  }

  if (!service("DetectorStore", p_detectorStore)) {
    log << MSG::ERROR << "Could not locate DetectorStore service" << endreq;
    return StatusCode::FAILURE;
  }

  if (!p_detectorStore->retrieve(p_muonDetectorManager)) {
    log << MSG::ERROR << "Could not retrieve MuonGM::MuonDetectorManager" << endreq;
    return StatusCode::FAILURE;
  }

  if (!(p_mdtIdHelper = p_muonDetectorManager->mdtIdHelper())) {
```

```
    log << MSG::ERROR << "Could not retrieve MdtIdHelper" << endreq;
    return StatusCode::FAILURE;
  }

  return StatusCode::SUCCESS;
}

StatusCode MakeSegmentNTuple::execute() {
  MsgStream log(msgSvc(), name());

  const EventInfo* eventInfo;
  if (!p_storeGate->retrieve(eventInfo)) {
    log << MSG::ERROR << "Could not retrieve EventInfo" << endreq;
    return StatusCode::FAILURE;
  }

  const MdtSegmentContainer *segContainer;
  if (!p_storeGate->retrieve(segContainer)) {
    log << MSG::ERROR << "Could not retrieve MooMdtSegmentContainer" << endreq;
    return StatusCode::FAILURE;
  } else {
    log << MSG::DEBUG << "Retrieved MooMdtSegmentContainer with " << segContainer->size() << " segment(s)" << endreq;
    MdtSegmentContainer::const_iterator segContainerIterator;
    std::map< Identifier, std::vector<MooMdtSegment*> > segmentMap;
    std::map< Identifier, std::vector<MooMdtSegment*> >::const_iterator mapIterator;

    for (segContainerIterator=segContainer->begin(); segContainerIterator!=segContainer->end(); segContainerIterator++) {
      MooMdtSegment *segment = *segContainerIterator;

      Identifier stationId = segment->identify_module();
      segmentMap[stationId].push_back(segment);
    }

    for (m_numSegs=0, mapIterator=segmentMap.begin(); mapIterator!=segmentMap.end(); mapIterator++) {
      log << MSG::DEBUG << "Iterating over segmentMap" << endreq;

      if (mapIterator->second.size() == 1) {
        log << MSG::DEBUG << "Found usable segment" << endreq;
        MooMdtSegment *segment = *mapIterator->second.begin();

        std::vector<MooMdtHit*>::const_iterator hit_itr;
        double phi = 0.0, rho = 0.0, z = 0.0;
        int hitsL1 = 0, hitsL2 = 0;

        for (hit_itr=segment->hits_begin(); hit_itr!=segment->hits_end(); ++hit_itr) {
          MooMdtHit *hit = *hit_itr;

          if (p_mdtIdHelper->multilayer(hit->HitId()) == 1) {
            hitsL1++;
          } else {
            hitsL2++;
          }

          phi += hit->position().phi() / segment->hits_count();
          rho += hit->position().perp() / segment->hits_count();
          z += hit->position().z() / segment->hits_count();
        }

        /* Adjust the average of the hit positions to be on the segment. */
        if (fabs(segment->cot_theta()) > fabs(segment->tan_theta())) {
          rho = segment->rho() + segment->tan_theta() * (z - segment->z());
        } else {
          z = segment->z() + segment->cot_theta() * (rho - segment->rho());
        }

        m_event                 = eventInfo->event_ID()->event_number();
        m_chamber[m_numSegs]    = mapIterator->first;
        m_numHitsL1[m_numSegs]  = hitsL1;
        m_numHitsL2[m_numSegs]  = hitsL2;
        m_z[m_numSegs]          = z;
        m_rho[m_numSegs]        = rho;
        m_theta[m_numSegs]      = segment->theta();

        m_thetaError[m_numSegs] = segment->theta_error();
        m_betaError[m_numSegs]  = segment->beta_error();
        m_chi2[m_numSegs]       = segment->Chi2();

        m_numSegs++;
```

```
      }
    }
  }

  if (m_writeRecords) {
    ntupleSvc()->writeRecord(p_nTuple);
  }

  return StatusCode::SUCCESS;
}

StatusCode MakeSegmentNTuple::finalize() {
  MsgStream log(msgSvc(), name());

  return StatusCode::SUCCESS;
}
```

C.2 Analysis program

C.2.1 AlignmentData.h

```
#ifndef AlignmentData_h
#define AlignmentData_h

#ifndef __CINT__
#include <TTree.h>
#include <Rtypes.h>
#include <RooFitCore/RooRealVar.hh>
#include <RooFitCore/RooDataSet.hh>
#include <RooFitCore/RooGenericPdf.hh>
#include <RooFitCore/RooAddPdf.hh>
#include <RooFitCore/RooPlot.hh>
#include <RooFitModels/RooGaussian.hh>
#endif

/**
 * Class representing a dataset and its Gaussian+constant fit.
 */
class AlignmentData {

  public:
    /**
     * Perform a Gaussian+constant fit on a tree variable.
     * \param tree Tree containing the data
     * \param variable Name of the variable in the data
     * \param min Minimal value of the fitted parameter
     * \param max Maximal value of the fitted parameter
     */
    AlignmentData(TTree *tree, char *variable, Double_t min, Double_t max);

    /**
     * Plots the data and the fit.
     */
    void Plot() const;

    /**
     * \return Mean of the Gaussian fit.
     */
    Double_t Mean() const { return m_mean.getVal(); }

    /**
     * \return Error on the mean of the Gaussian fit.
     */
    Double_t MeanError() const { return m_mean.getError(); }

    /**
     * \return Sigma of the Gaussian fit.
     */
    Double_t Sigma() const { return m_sigma.getVal(); }

    /**
     * \return Error on the sigma of the Gaussian fit.
     */
    Double_t SigmaError() const { return m_sigma.getError(); }
```

```
  private:
    /** Fit variables. */
    RooRealVar m_value, m_mean, m_sigma;

    /** Minimum and maximum values of m_value. */
    Double_t m_min, m_max;

    /** Ratio between the constant and the Gaussian PDF. */
    RooRealVar m_fraction;

    /** Dataset containing misalignment data for a coordinate. */
    RooDataSet m_dataset;

    /** Constant background PDF. */
    RooGenericPdf m_background;

    /** Gaussian PDF. */
    RooGaussian m_gauss;

    /** Combined PDF. */
    RooAddPdf m_model;

    /** Fit result object. */
    RooFitResult *m_result;
};

#endif
```

C.2.2 AlignmentData.cxx

```
#include "AlignmentData.h"

AlignmentData::AlignmentData(TTree *tree, char *variable, Double_t min, Double_t max)
  : m_min( min ), m_max( max ),
    m_value( RooRealVar(variable, variable, min, max) ),
    m_dataset( RooDataSet(variable, variable, tree, m_value) ),
    m_mean( RooRealVar("mean", "mean", min, max) ),
    m_sigma( RooRealVar("sigma", "sigma", 0.0, max-min) ),
    m_gauss( RooGaussian("gauss", "gauss", m_value, m_mean, m_sigma) ),
    m_background( RooGenericPdf("background", "1", RooArgList()) ),
    m_fraction( RooRealVar("fraction", "fraction", 0.1, 0.0, 1.0) ),
    m_model( RooAddPdf("model", "model", RooArgList(m_background, m_gauss), m_fraction) ) {

  m_model.fitTo(m_dataset);
}

void AlignmentData::Plot() const {

  RooPlot *plot = m_value.frame();

  m_dataset.plotOn(plot);
  m_model.plotOn(plot);
  plot->Draw();
}
```

C.2.3 ChamberAlignment.h

```
#ifndef ChamberAlignment_h
#define ChamberAlignment_h

#ifndef __CINT__
#include <Rtypes.h>
#include <RooFitCore/RooRealVar.hh>
#endif

class AlignmentData;
class TTree;

/**
 * Class representing the alignment data of an MDT chamber.
 */
class ChamberAlignment {
```

```
  public:
    /**
     * \param identifier Athena identifier of the chamber.
     */
    ChamberAlignment(Int_t identifier);

    /**
     * Adds a misalignment to the dataset.
     * \param dz Z misalignment
     * \param dtheta Theta misalignment
     */
    void Add(Double_t dz, Double_t dtheta, Double_t, Double_t, Double_t, Int_t, Int_t, Double_t, Double_t);

    /**
     * Performs a fit on the misalignment data.
     */
    void Fit();

    /**
     * Creates a window and plots the z and theta alignment data and their fits.
     */
    void Plot() const;

    /**
     * Prints the alignment data fitted values.
     */
    void Print() const;

    /**
     * \return Tree created from the misalignment data
     */
    const TTree *Tree() const { return m_tree; }

    /**
     * \return Athena identifier of the chamber
     */
    Int_t Identify() const { return m_identifier; }

    /**
     * \return Misalignment data structure for z.
     */
    AlignmentData *Z() const { return m_alignment_z; }

    /**
     * \return Misalignment data structure for theta.
     */
    AlignmentData *Theta() const { return m_alignment_theta; }

  private:
    /** Identifier of the chamber this alignment object belongs to. */
    Int_t m_identifier;

    /** Data structure for the coordinates we want to align. */
    AlignmentData *m_alignment_z, *m_alignment_theta;

    /** Misalignment data tree */
    TTree *m_tree;

    /** Tree data. */
    Double_t t_dz, t_dtheta;

    /** Tree data for selection. */
    Int_t t_hitsl1, t_hitsl2;

    /** Tree data for debugging. */
    Double_t t_z, t_theta, t_chi2, t_thetaerror, t_betaerror;
};

#endif
```

C.2.4 ChamberAlignment.cxx

```
#include "ChamberAlignment.h"
#include "AlignmentData.h"

#include <TCanvas.h>
```

```
#include <TTree.h>
#include <iostream>
#include <sstream>

ChamberAlignment::ChamberAlignment(Int_t identifier) : m_identifier(identifier) {

  std::ostringstream title_stream;
  title_stream << "MDT_" << m_identifier << std::ends;
  std::string title_string = title_stream.str();
  const char *title = title_string.c_str();

  m_tree = new TTree(title, "alignment data");
  m_tree->Branch("dz", &t_dz, "dz/D");
  m_tree->Branch("dtheta", &t_dtheta, "dtheta/D");

  m_tree->Branch("z", &t_z, "z/D");
  m_tree->Branch("theta", &t_theta, "theta/D");
  m_tree->Branch("chi2", &t_chi2, "chi2/D");
  m_tree->Branch("hitsl1", &t_hitsl1, "hitsl1/I");
  m_tree->Branch("hitsl2", &t_hitsl2, "hitsl2/I");
  m_tree->Branch("thetaerror", &t_thetaerror, "thetaerror/D");
  m_tree->Branch("betaerror", &t_betaerror, "betaerror/D");
}

void ChamberAlignment::Add(Double_t dz, Double_t dtheta, Double_t z, Double_t theta, Double_t chi2, Int_t hitsl1,
                           Int_t hitsl2, Double_t thetaerror, Double_t betaerror) {

  t_dz = dz;
  t_dtheta = dtheta;
  t_z = z;
  t_theta = theta;
  t_chi2 = chi2;
  t_hitsl1 = hitsl1;
  t_hitsl2 = hitsl2;
  t_thetaerror = thetaerror;
  t_betaerror = betaerror;
  m_tree->Fill();
}

void ChamberAlignment::Fit() {

  m_alignment_z = new AlignmentData(m_tree, "dz", -20.0, 20.0);
  m_alignment_theta = new AlignmentData(m_tree, "dtheta", -0.004, 0.004);
}

void ChamberAlignment::Plot() const {

  std::ostringstream title_stream;
  title_stream << "MDT_" << m_identifier << std::ends;
  std::string title_string = title_stream.str();
  const char *title = title_string.c_str();

  TCanvas *canvas = new TCanvas(title, title, 700, 875);
  canvas->Divide(1, 2);

  canvas->cd(1);
  m_alignment_z->Plot();
  canvas->cd(2);
  m_alignment_theta->Plot();
}

void ChamberAlignment::Print() const {

  std::cout << "Chamber " << Identify() << ":" << std::endl;
  std::cout << " z: mean " << m_alignment_z->Mean() << " +/- " << m_alignment_z->MeanError()
            << " sigma " << m_alignment_z->Sigma() << " +/- " << m_alignment_z->SigmaError()
            << std::endl;
  std::cout << " theta: mean " << m_alignment_theta->Mean() << " +/- " << m_alignment_theta->MeanError()
            << " sigma " << m_alignment_theta->Sigma() << " +/- " << m_alignment_theta->SigmaError()
            << std::endl;
}
```

C.2.5 MuonTBAlignment.h

```
#ifndef MuonTBAlignment_h
#define MuonTBAlignment_h
```

```
#include <iostream>
#include <vector>

#ifndef __CINT__
#include <Rtypes.h>
#endif

class ChamberAlignment;
class TTree;

/**
 * Class representing the alignment data of a series of MDT chambers.
 */
class MuonTBAlignment {

  public:
    /**
     * Constructor.
     * \param filename Name of an ntuple with alignment data
     */
    MuonTBAlignment(const char *filename = "alignmentdata.root");

    /**
     * Get a pointer to the chamber alignment object.
     * \param identifier Athena identifier for the chamber
     * \return Alignment object
     */
    ChamberAlignment *Get(Int_t identifier);

    /**
     * Loop over tree data and create alignment objects for encountered chambers.
     * \param refChamber Athena identifier of the chamber to use as reference
     */
    void Loop(Int_t refChamber = 1627389952);

    /**
     * Perform a fit on the misalignment data.
     * \param identifier Chamber to fit, 0 means all
     */
    void Fit(Int_t identifier = 0);

    /**
     * Print alignment data.
     * \param identifier Chamber to print, 0 means all
     */
    void Print(Int_t identifier = 0) const;

    /**
     * Plot alignment data.
     * \param identifier Chamber to plot, 0 means all
     */
    void Plot(Int_t identifier = 0) const;

  private:
    /** Vector of ChamberAlignment objects. */
    std::vector<ChamberAlignment *> m_alignment;

    /** Tree produced by the Athena algorithm. */
    TTree *m_tree;

    /** Data from the input tree. */
    Int_t t_numsegs, t_chamber[15], t_numhits1[15], t_numhits2[15];
    /** Data from the input tree. */
    Double_t t_z[15], t_rho[15], t_theta[15], t_betaerror[15], t_chi2[15], t_thetaerror[15];
};

#endif
```

C.2.6 MuonTBAlignment.cxx

```
#include "MuonTBAlignment.h"
#include "ChamberAlignment.h"
#include "AlignmentData.h"

MuonTBAlignment::MuonTBAlignment(const char *filename) {
```

```
  TFile *file = new TFile(filename);
  file->cd("/TB");
  m_tree = (TTree *)gDirectory->Get("tree");

  m_tree->SetBranchAddress("numsegs", &t_numsegs);
  m_tree->SetBranchAddress("chamber", t_chamber);
  m_tree->SetBranchAddress("numhitsl1", t_numhitsl1);
  m_tree->SetBranchAddress("numhitsl2", t_numhitsl2);
  m_tree->SetBranchAddress("z", t_z);
  m_tree->SetBranchAddress("rho", t_rho);
  m_tree->SetBranchAddress("theta", t_theta);
  m_tree->SetBranchAddress("betaerror", t_betaerror);
  m_tree->SetBranchAddress("chi2", t_chi2);
  m_tree->SetBranchAddress("thetaerror", t_thetaerror);

  m_tree->Print();
}

ChamberAlignment *MuonTBAlignment::Get(Int_t identifier) {

  /* Find the requested chamber. */
  std::vector<ChamberAlignment *>::iterator iterator;
  for (iterator=m_alignment.begin(); iterator<m_alignment.end(); iterator++) {

    if ((*iterator)->Identify() == identifier) {
      return *iterator;
    }
  }

  /* It wasn't found, so create a new one and return that. */
  ChamberAlignment *align = new ChamberAlignment(identifier);

  m_alignment.push_back(align);

  return align;
}

void MuonTBAlignment::Loop(Int_t refChamber) {

  if (!m_tree) return;

  Int_t numentries = (Int_t)m_tree->GetEntriesFast();
  for (Int_t i=0; i<numentries; i++) {
    m_tree->GetEntry(i);

    Int_t refIndex = -1;

    /* Loop over all segments and find the one in the reference chamber, if any. */
    for (Int_t j=0; j<t_numsegs; j++) {

      if (t_chamber[j] == refChamber) {
        refIndex = j;
        break;
      }
    }

    /* Skip event if there's no segment in the reference chamber. */
    if (refIndex < 0) continue;

    /* Demand at least 3 hits in each multilayer. */
    if (t_numhitsl1[refIndex] < 3 || t_numhitsl2[refIndex] < 3) continue;

    /* Loop over all segments and calculate the deviations w.r.t. the reference chamber. */
    for (Int_t j=0; j<t_numsegs; j++) {

      /* Don't align the reference chamber with respect to itself. */
      if (j == refIndex) continue;

      Double_t drho = t_rho[j] - t_rho[refIndex];
      Double_t dtheta = t_theta[j] - t_theta[refIndex];
      Double_t dz = t_z[j] - t_z[refIndex] - cos(t_theta[refIndex]) * drho;

      Get(t_chamber[j])->Add(dz, dtheta, t_z[refIndex], t_theta[refIndex], t_chi2[refIndex],
          t_numhitsl1[refIndex], t_numhitsl2[refIndex], t_thetaerror[refIndex], t_betaerror[refIndex]);
    }

    if (!(i % 1000)) std::cout << "Event: " << i << std::endl;
```

```
  }
}

void MuonTBAlignment::Fit(Int_t identifier) {

  std::vector<ChamberAlignment *>::iterator iterator;
  for (iterator=m_alignment.begin(); iterator<m_alignment.end(); iterator++) {
    ChamberAlignment *chamber = *iterator;

    if (!identifier || chamber->Identify() == identifier) {
      chamber->Fit();
    }
  }
}

void MuonTBAlignment::Print(Int_t identifier) const {

  std::vector<ChamberAlignment *>::const_iterator iterator;
  for (iterator=m_alignment.begin(); iterator<m_alignment.end(); iterator++) {
    const ChamberAlignment *chamber = *iterator;

    if (!identifier || chamber->Identify() == identifier) {
      chamber->Print();
    }
  }
}

void MuonTBAlignment::Plot(Int_t identifier) const {

  std::vector<ChamberAlignment *>::const_iterator iterator;
  for (iterator=m_alignment.begin(); iterator<m_alignment.end(); iterator++) {
    ChamberAlignment *chamber = *iterator;

    if (!identifier || chamber->Identify() == identifier) {
      chamber->Plot();
    }
  }
}
```

C.2.7 align.cxx

```
#include "MuonTBAlignment.h"
#include "ChamberAlignment.h"
#include "AlignmentData.h"

#include "TApplication.h"
#include "TCanvas.h"

int main(int argc, char **argv) {

  if (argc < 4) {
    std::cerr << "Usage: " << argv[0] << " <ntuple> <reference> <chamber1> [chamber2 ...]" << std::endl;
    exit(1);
  }

  /* TApplication should keep its hands off argc and argv, so give it null pointers instead. */
  TApplication theApp("App", 0, 0);

  MuonTBAlignment M(argv[1]);
  M.Loop(atoi(argv[2]));

  for (int i=3; i<argc; i++) {
    M.Fit(atoi(argv[i]));
    M.Plot(atoi(argv[i]));
  }

  for (int i=3; i<argc; i++) {
    M.Print(atoi(argv[i]));
  }

  theApp.Run();
}
```

C.2.8 MuonTBMiddleChamber.h

```
#ifndef MuonTBMiddleChamber_h
#define MuonTBMiddleChamber_h

#include <iostream>
#include <vector>

#ifndef __CINT__
#include <Rtypes.h>
#endif

#define BIL 1627389952
#define BML 1694498816
#define BOL 1728053248

class ChamberAlignment;
class TTree;

/**
 * Class representing the alignment data of a series of MDT chambers.
 */
class MuonTBMiddleChamber {

  public:
    /**
     * Constructor.
     * \param filename Name of an ntuple with alignment data
     */
    MuonTBMiddleChamber(const char *filename = "alignmentdata.root");

    /**
     * Get a pointer to the chamber alignment object.
     * \return Alignment object
     */
    ChamberAlignment *Get();

    /**
     * Loop over tree data and create alignment objects for encountered chambers.
     */
    void Loop();

    /**
     * Perform a fit on the misalignment data.
     */
    void Fit();

    /**
     * Print alignment data.
     */
    void Print() const;

    /**
     * Plot alignment data.
     */
    void Plot() const;

  private:
    /** ChamberAlignment object. */
    ChamberAlignment *m_alignment;

    /** Tree produced by the Athena algorithm. */
    TTree *m_tree;

    /** Data from the input tree. */
    Int_t t_numsegs, t_chamber[15], t_numhitsl1[15], t_numhitsl2[15];
    /** Data from the input tree. */
    Double_t t_z[15], t_rho[15], t_theta[15], t_betaerror[15], t_chi2[15], t_thetaerror[15];
};

#endif
```

C.2.9 MuonTBMiddleChamber.cxx

```
#include "MuonTBMiddleChamber.h"
#include "ChamberAlignment.h"
```

```
#include "AlignmentData.h"

MuonTBMiddleChamber::MuonTBMiddleChamber(const char *filename) {

  TFile *file = new TFile(filename);
  file->cd("/TB");
  m_tree = (TTree *)gDirectory->Get("tree");

  m_tree->SetBranchAddress("numsegs", &t_numsegs);
  m_tree->SetBranchAddress("chamber", t_chamber);
  m_tree->SetBranchAddress("numhitsl1", t_numhitsl1);
  m_tree->SetBranchAddress("numhitsl2", t_numhitsl2);
  m_tree->SetBranchAddress("z", t_z);
  m_tree->SetBranchAddress("rho", t_rho);
  m_tree->SetBranchAddress("theta", t_theta);
  m_tree->SetBranchAddress("betaerror", t_betaerror);
  m_tree->SetBranchAddress("chi2", t_chi2);
  m_tree->SetBranchAddress("thetaerror", t_thetaerror);

  m_tree->Print();
}

ChamberAlignment *MuonTBMiddleChamber::Get() {

  return m_alignment;
}

void MuonTBMiddleChamber::Loop() {

  if (!m_tree) return;

  m_alignment = new ChamberAlignment(BML);

  Int_t numentries = (Int_t)m_tree->GetEntriesFast();
  for (Int_t i=0; i<numentries; i++) {
    m_tree->GetEntry(i);

    Int_t bilIndex = -1, bmlIndex = -1, bolIndex = -1;

    for (Int_t j=0; j<t_numsegs; j++) {

      switch(t_chamber[j]) {
        case BIL:
          bilIndex = j;
          break;
        case BML:
          bmlIndex = j;
          break;
        case BOL:
          bolIndex = j;
          break;
      }
    }

    if (bilIndex < 0 || bmlIndex < 0 || bolIndex < 0) continue;

    /* Get corrected z position for the three chambers. */
    Double_t zi = t_z[bilIndex];
    Double_t zm = t_z[bmlIndex] - 3.127;
    Double_t zo = t_z[bolIndex] - 0.477;

    Double_t dz = zm - (zi + zo)/2.0;

    /* We are only using the ChamberAlignment class to make a
     * plot of dz this time, the other values are not used. */
    m_alignment->Add(dz, 0.0, 0.0, 0.0, 0.0, 0, 0, 0.0, 0.0);

    if (!(i % 1000)) std::cout << "Event: " << i << std::endl;
  }
}

void MuonTBMiddleChamber::Fit() {

  m_alignment->Fit();
}

void MuonTBMiddleChamber::Print() const {
```

```
  m_alignment->Print();
}

void MuonTBMiddleChamber::Plot() const {

  m_alignment->Plot();
}
```

C.2.10 middlechamber.cxx

```
#include "MuonTBMiddleChamber.h"
#include "ChamberAlignment.h"
#include "AlignmentData.h"

#include "TApplication.h"
#include "TCanvas.h"

int main(int argc, char **argv) {

  if (argc != 2) {
    std::cerr << "Usage: " << argv[0] << " <ntuple>" << std::endl;
    exit(1);
  }

  /* TApplication should keep its hands off argc and argv, so give it null pointers instead. */
  TApplication theApp("App", 0, 0);

  MuonTBMiddleChamber M(argv[1]);
  M.Loop();

  M.Fit();
  M.Plot();
  M.Print();

  theApp.Run();
}
```

Bibliography

[1] ATLAS Collaboration, *ATLAS Technical Proposal for a General-Purpose pp Experiment at the Large Hadron Collider at CERN*, CERN/LHCC/94-43, 15 December, 1994

[2] ATLAS Collaboration, *ATLAS Detector and Physics Performance Technical Design Report*, CERN/LHCC/99-14, 25 May 1999

[3] ATLAS Muon Collaboration, *Muon Spectrometer Technical Design Report*, CERN/LHCC/97-22, 31 May 1997

[4] Peter Kluit, *Twin Tubes*, presentation given at the Dutch Atlas Day, Amsterdam, 30 May 2005

[5] *Athena Developer Guide*, 8.0.0 (draft), 9 February 2004, `http://cern.ch/Atlas/GROUPS/SOFTWARE/OO/architecture/`

[6] MOORE Group, *Track reconstruction in the ATLAS Muon Spectrometer with MOORE*, ATL-COM-MUON-2003-012, 2 October 2003

[7] H. Drevermann, D. Kuhn, B.S. Nilsson, *Event Display: Can We See What We Want To See?*, CERN/ECP/95-25, 19 October 1995