Muon Segment Reconstruction – Linked List Algorithm

Author: O. Peters (opeters@fnal.gov)

Introduction

Segment reconstruction is the part of the muon reconstruction code in which the pattern recognition is done. Hits in the muon wire chambers are combined, and a straight line, called a segment, is fit through the hits. After this, the found segment is combined with scintillator hit information for timing information on the segment. This document describes the Linked List algorithm, its code implementation and the reconstruction results. Appended are instructions on how to run the different software packages.

Algorithm

For the segment reconstruction algorithm, the muon detector is split in two parts, each of which is again split in two parts:

- Central system (WAMUS)
    - Octants 0, 3, 4 and 7 (vertical arrows in Figure 1)
    - Octants 1, 2, 5 and 6 (horizontal arrows in Figure 1)
- Forward system (FAMUS)
    - Octants 0, 3, 4 and 7 (vertical arrows in Figure 2)
    - Octants 1, 2, 5 and 6 (vertical arrows in Figure 2)

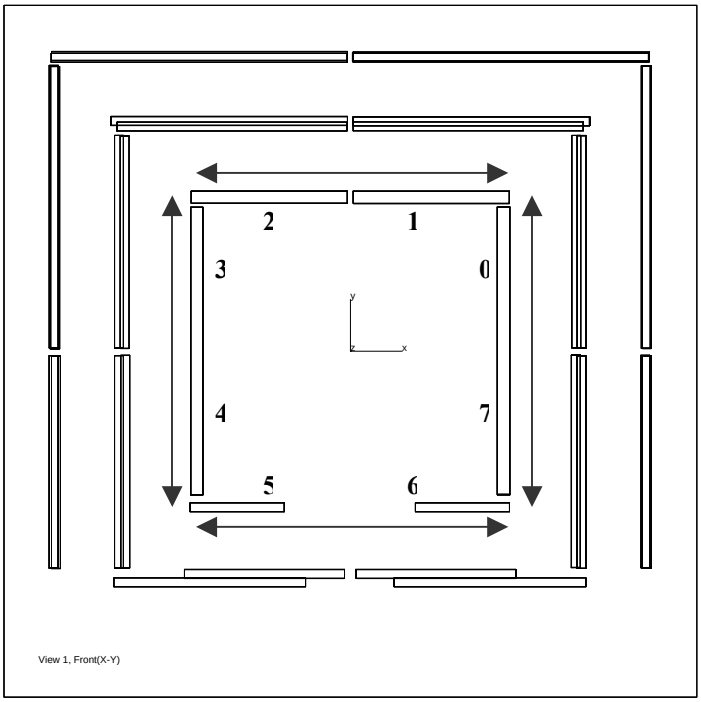

Figure 1: Wire direction in central system (numbers are octants)

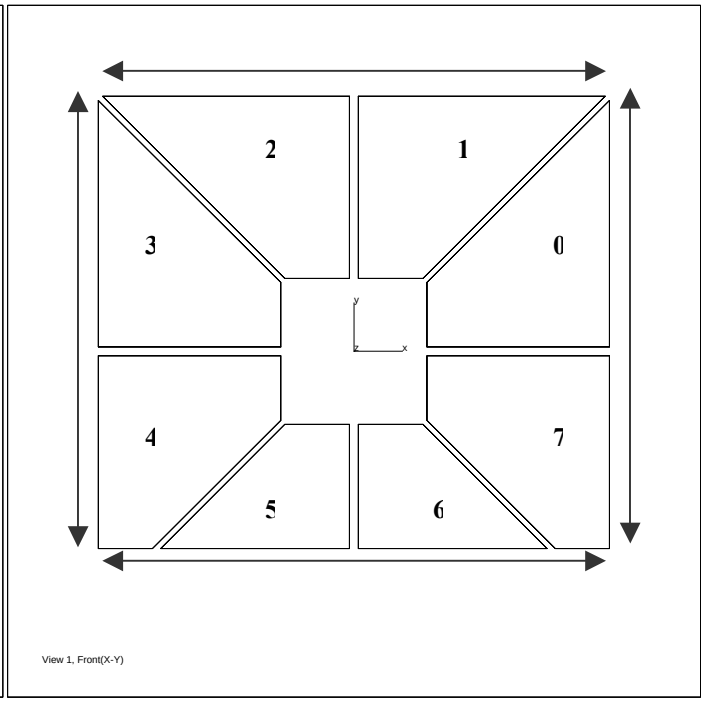

Figure 2: Wire direction in forward system (numbers are octants)

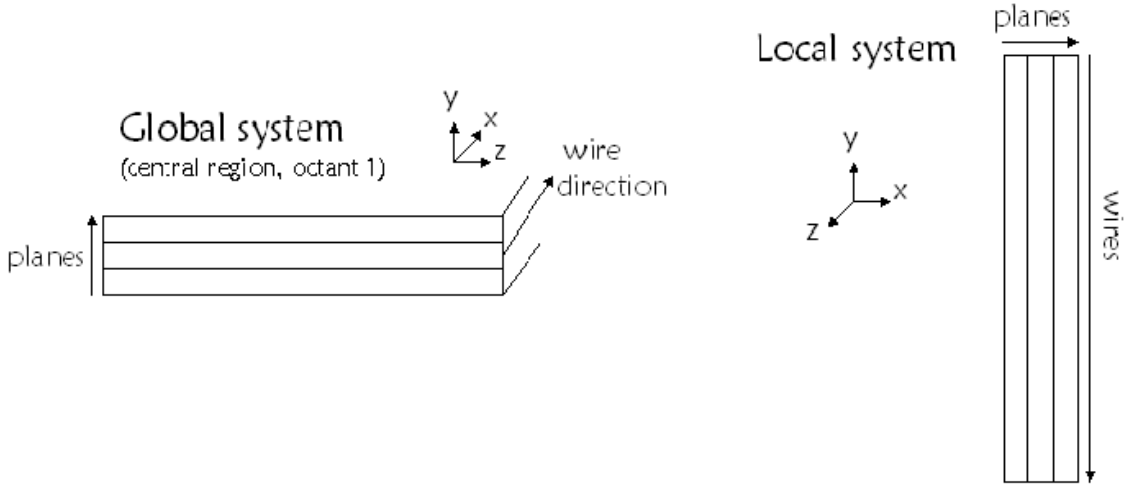

Figure 3: Geometry transformation between local and global system

This division is based on the difference in geometry between these parts. Both in the WAMUS and FAMUS, the wires are oriented along the y-axis (octants 0, 3, 4 and 7) or along the x-axis (octants 1, 2, 5 and 6). However, in the WAMUS, the wire plane is in the x-z or y-z plane (the planes are stacked in the y or x direction respectively), while in the FAMUS this is the x-y plane (where the planes are stacked in the z direction). To overcome this division, and still be able to have a common algorithm for all different parts, the pattern recognition is not done in the global system but in a local system. This requires a transformation of the reconstructed hits into the local system.

Another problem arises from the fact that the wire hits in the central system (PDT hits) and the wire hits in the forward system (MDT hits) are treated differently. The PDT hits need to be adjusted with the angle of the track that generated the hit to calculate the correct drift distance from the drift time, since the time-to-drift relationship is dependant on the angle of the track. The MDT hits need to be updated with a pixel hit for the axial[1] position (and better drift time) information, since the MDT hit is read out on only one side of the wire and has no axial position information. All this is reflected in the algorithm.

The algorithm is divided in 7 steps:

1. Transformation of 'global' hits to 'local' hits
2. Creation of links between hits
3. Matching of links into local segments
4. Fitting of local segments
5. Using vertex constraint for A-layer segment (if applicable)
6. Matching of local segments
7. Filtering of local segments; transformation back to global system

[1] The axial direction of a hit is the direction along the wire; the drift direction is perpendicular to the wire

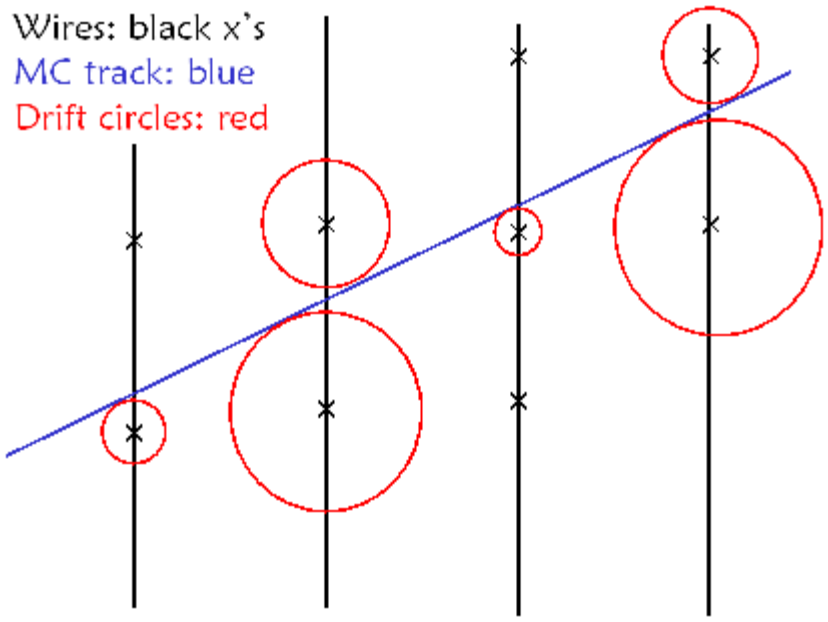

Figure 4: Segment reconstruction in drift plane

Step 1: In the first step, we move the hits from their global coordinate system to a local coordinate system. This is done so we can work in a coordinate space where the drift 'circles' are in the x-y plane, and the wire direction is in the z direction (see Figure 1). The plane in which the wires are located (the wire plane) is transformed such that it is parallel to the y-axis. Both the global and the local coordinate systems are right-handed; the transformation does not change the handedness. Also, the transformation does not involve a translation, only a rotation. The position of the origin remains the same in the local system as it was in the global system. For example, a position (x, y, z) in the central region, octant 1, will be transformed to (y, z, x).

The hits in this local coordinate system are called PDTSegmentHits for the PDT system, and MDTSegmentHits for the MDT system. These SegmentHits are hits lying on the (to be) reconstructed segment; they encapsulate the underlying reconstructed hit to add additional functionality which is needed to fit a segment with these hits. Also, because of the left-right ambiguity of the wire hits, two local hits, one on each side of the wire, are created for each wire hit. This is shown in Figure 3. Here, the black lines are the wire planes (the wires themselves are running perpendicular to the plane of the picture), and the wires are depicted by the black crosses. The MC track is represented by the blue line, and the red circles are the drift circles. In reality, the equidrifttime lines of the PDT's are not circles but look more like ovals (this is taken into account in the real algorithm). Each segment hit is initially placed on top or on the bottom of each of these circles.

The hits presented to the segment reconstruction need to be divided in collections that are suspected to be generated by one muon track. This is implemented by grouping the wire hits by MuoSectionIndex, where each MuoSectionIndex represents one wire chamber. Other schemes that are not dependent on the physical layout of the muon detector system can be envisioned, but are not implemented.

Step 2: After the hits are transformed to the local system, the links between the segment hits are made. These links are called LocalSegments, and consist of one or more segment hits. At first, these LocalSegments are just connections between two hits. As the algorithm progresses, the LocalSegments are matched with other LocalSegments to form new LocalSegments to form new LocalSegments which contain 3 or more hits. Thus, at the start, each LocalSegment is created with two segment hits, taking all possible combinations of segment hits which satisfy the following conditions:

1. Both segment hits are not coming from the same underlying wire hit (i.e. they are not positioned on the same drift circle);

2. The separation between both hits along the y-direction is smaller than 20 centimeters[2];

3. Both hits are not on the same plane, except if they are coming from two neighboring hits, and one is on the top of a drift circle, while the other is on the bottom; this condition is needed to find segments when less than 3 planes are hit.

LocalSegments are also made between hits which are not on neighboring planes, thus taking into account inefficiencies when the track passes a plane and not creating a hit.

The position and direction of the LocalSegment are calculated using the position of the two SegmentHits from which the LocalSegment is created. For LocalSegments in the central system, where the hits on the LocalSegment have a dependence on the angle of the LocalSegment, the direction is calculated once with the position of the hits, after which the position of the hits is updated with that calculated direction. Using those updated positions, the direction of the LocalSegment is recalculated. The created LocalSegment is assigned to the segment hit that has the lowest x-coordinate.

Step 3: Each LocalSegment is assigned to the segment hit that it contains that is at the lowest x-position, which is called the 'left' hit of the LocalSegment. The hit on the other side of the segment, on the end, is called the 'right' hit of the segment. At this stage, there is a unique correspondence between hits and LocalSegments. Every hit contains a collection of LocalSegments that are made with that hit and one other hit that is located at a higher x position. Each LocalSegment contains two hits, which can be shared with other LocalSegments. Note, that when the algorithm progresses, LocalSegments can be formed which contain more than two hits.

Now, to link all LocalSegments which have been made in the previous step, a loop is performed over all the segment hits with the goal of creating a 'tree' of linked LocalSegments for each segment hit (see Figure 5). A segment hit is called to have a tree if either it does not contain any LocalSegments (these are typically the hits on the right most plane – since there are no other hits to the right of these hits to which a LocalSegment can be made, these hits do not have any LocalSegments assigned to them), or each LocalSegment that the hit contains has already been tried to match with LocalSegments to the right. Starting with one, a loop is made over all the LocalSegments that are assigned to that hit. Each of these LocalSegments holds a segment hit at the end (the right hit). If this right hit already has a tree, the LocalSegment is matched with the LocalSegments assigned to that right hit, creating the tree for the left hit. If this right hit does not have a tree yet, the tree is recursively created for this hit before proceeding. Matching two LocalSegments involves comparing the angle and the position of both LocalSegments. If these are compatible with a straight line, a new LocalSegment, which contains all hits of the two other

[2] One could envision an algorithm here that relates the direction of the line segment between the hits with the position of the hit. If the direction of the line segment between the hits would not correspond to the track of a particle coming from the origin, the LocalSegment can be discarded.

LocalSegments, is created and assigned to the leftmost hit. The parameters which determine the compatibility of the segment match with a straight line are defined in the RCP file.

This algorithm is depicted in figures 4-1 through 4-8.

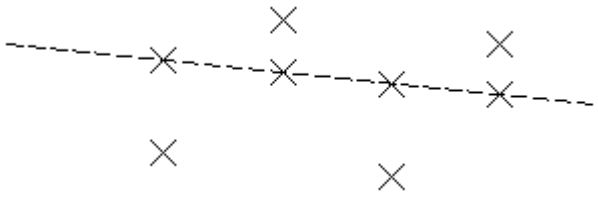

Figure 4-1: Track (dotted line) passing through chamber creating hits

Figure 4-1 shows the track (dashed line) passing through a chamber, generating hits. The crosses are the points where the drift circle of the hit passes through the plane of the wires; these are the SegmentHits.

Figure 4-2: SegmentHits with all LocalSegments that can be created between them

After the hits are created, all the LocalSegments that can be made using the restrictions as explained above are made, as is shown in Figure 4-2. Each LocalSegment drawn here is assigned to the left hit. Note, that none of these hits have a tree; that is, none of them are processed yet. This will be explained in more detail below.

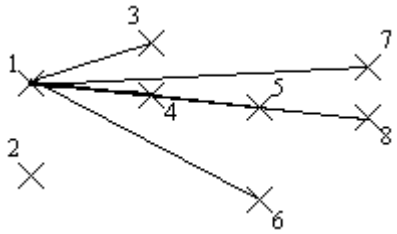

Figure 4-3: All LocalSegments belonging to SegmentHit 1

For convenience, the SegmentHits are numbered and will be called (in this example) H1 through H8. The LocalSegments belonging to H1 are called S13, S14, S15, S16, S17 and S18. The algorithm now starts with H1 (it could also have started with H2, it does not matter for the algorithm). A loop is made over every LocalSegment assigned to H1, that is, all the drawn LocalSegments in Figure 4-3, starting with S13. For S13, we take a look at the end (right) hit of this LocalSegment, which is H3.

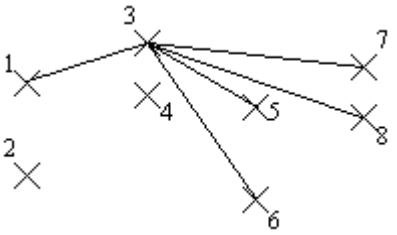

Figure 4-4: All LocalSegments belonging to H3, together with S13 (which belongs to H1)

Had H3 been assigned a tree somewhere earlier in the algorithm, we could now match S13 with the LocalSegments assigned to H3, and possibly create new LocalSegments. However, at this point, H3 has not yet been assigned a tree, so we have to loop over all LocalSegments (S35, S36, S37 and S38) that are assigned to H3. Starting with S35, we now look at H5, which also have not been processed yet.

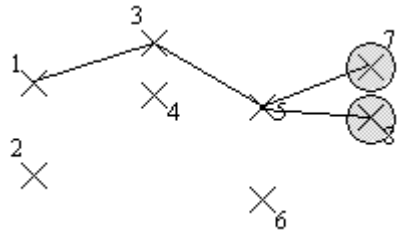

Figure 4-5: LocalSegments belonging to H5, together with S13 and S35. H7 an H8 are processed, which is shown by the gray circle.

H5 has two LocalSegments assigned to it, S57 and S58. The endpoints of both LocalSegments don't have any LocalSegments assigned to them. Thus, we set H7 and H8 to have a tree, i.e. they are so-called 'processed', which is denoted by the gray circles around the hits. Looking back at H5, we have processed all the LocalSegments assigned to it, so we set H5 to have a tree too.

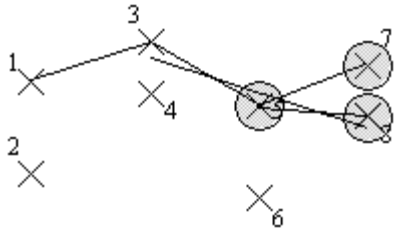

Figure 4-6: H5 is processed and a new LocalSegment S358 is created

Now that the endpoint of S35 has been processed, we can try to match S35 with any other LocalSegments which belong to the (processed) hit H5. The angle between S35 and S57 is too large to come from a straight line, but the angle between S35 and S58 is not. Since S35 and S38 are compatible, a new LocalSegment is created which is fitted through hit H3, H5 and H8. This new LocalSegment (S358) is then assigned to H3.

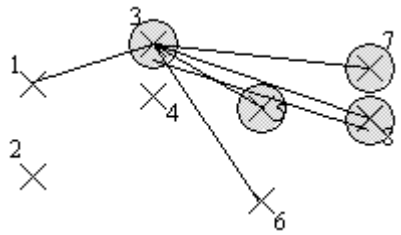

Figure 4-7: H3 is processed, and all LocalSegments assigned to H3 are shown, together with S13

After having looped over all LocalSegments assigned to H3, and made all possible new LocalSegments, H3 is set to be processed. Since no more new LocalSegments are possible in this configuration, H3 only has one more LocalSegment (S358). We can now step back to our starting point, H1.

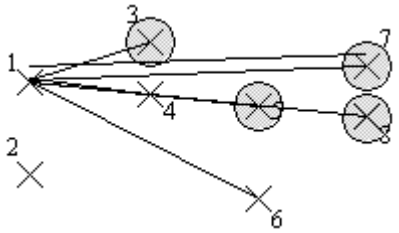

Figure 4-8: All LocalSegments, assigned to H1, after processing S13

We started from S13, and processed the tree starting from H3. Now we try to match the LocalSegment S13 with the LocalSegments assigned to H3. Only S37 is compatible with S13, so a new LocalSegment S137 is created and assigned to H1. The algorithm now proceeds by looking at the LocalSegment S14, and starts processing H4. Note, that when the algorithm encounters a hit which has previously been assigned a tree, it does not process that hit again. Also, after H1 is completely processed, the algorithm steps to the next hit which has not been processed yet, in this case H2.

Step 4: After matching all the local segments, each of them is fitted. The fit is done in two dimensions (x and y). The fit in z (i.e. the direction along the wire) is a separate fit. The former fit provides a χ^2 for the goodness of the fit. To reduce combinatorics, this χ^2 plus the number of hits on the segment is now used to filter out the best four segments, where a better segment has more hits assigned to it, or, if two segments have the same number of hits, the better segment has lower χ^2. After this, the remaining segments are fitted with the scintillator hits, to determine the time of these segments, and in case of the forward system, better position. This matching is done by extrapolating the segment to the x position of the scintillator hit, and determining whether the difference in the y position of the segment and the scintillator is smaller than δ*(sum of the errors in the y position of the segment and the y position of the hit), where δ is a cut-off parameter defined in the RCP file. The closest scintillator found this way is matched to the segment. If a match with a scintillator is found, the segment is refitted in the case the segment is located in the forward region. In the refit, the position of the scintillator itself is not taken into account but is only used to update the MDT hits on the segment with a position along the wire and to give time information.

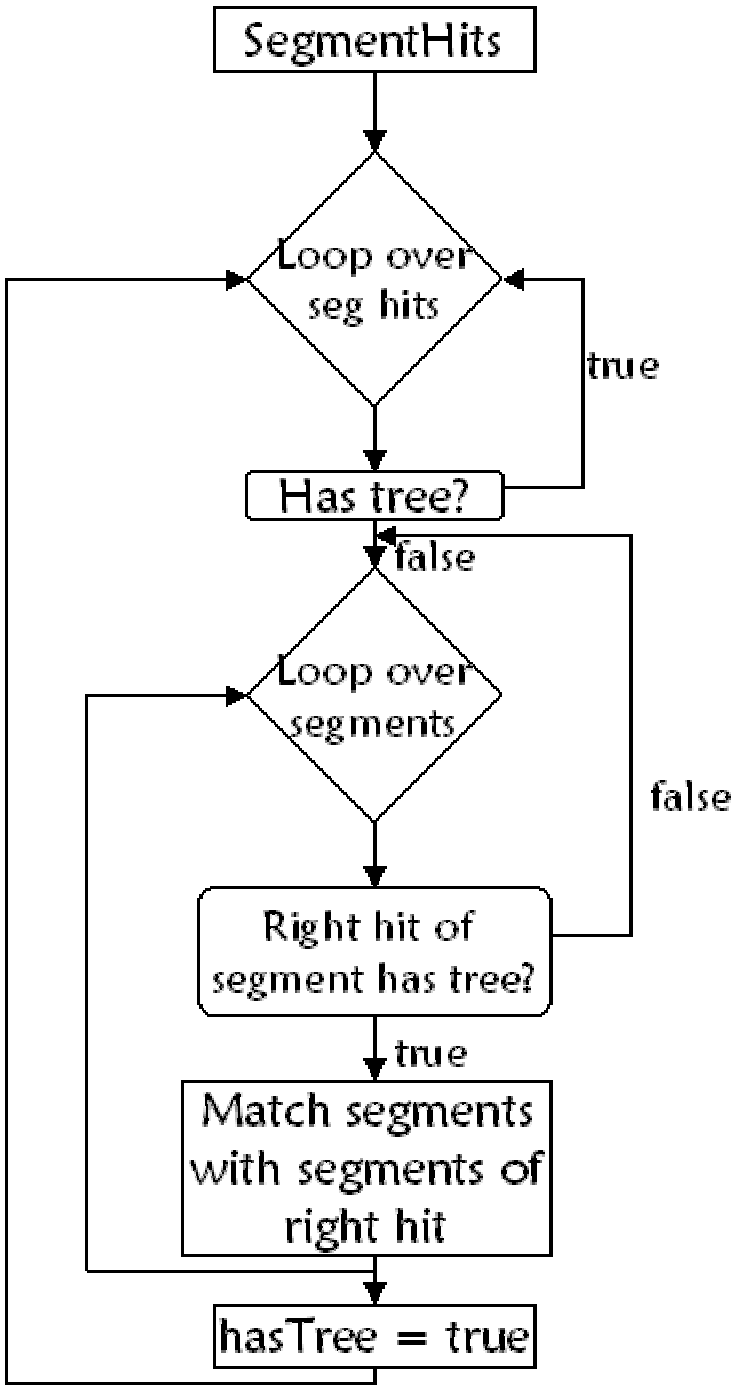

Figure 5: Flow diagram of tree building

Step 5: For a better angle resolution of the segment, the vertex position can be used to update the direction of the segment. Whether this is done depends on a switch in the RCP file, called 'useVertexA'. If this switch is set to true, the direction in the drift plane of each A-segment is taken to be the same as the direction of a line from the origin to the position of the segment.

Step 6: Because there is no a magnetic field between the B and C layers, it is possible to match the local segments in the B layer with the local segments in the C layer and do a better fit with a longer lever arm. In theory, it should also be possible to match segments in the same layer, but in a different system (e.g. PDT-A segments with MDT-A segments). However, because of the transformation step in Step 1, it is not directly possible to extrapolate a local segment in one subsystem to a local segment in another subsystem. Also, matching segments between octants could be done here, but is not implemented. Only B-C matching in the same region and octant is done. This is implemented by looping over all the segments in the B layer in one region, and match them with any other segment in the same region in the C layer. A match is found when the difference in the extrapolated y-position and the position of the C layer segment is smaller than δ*(error on extrapolated y-position + error on C layer segments y-position), where δ is a cut-off parameter defined in the RCP file. If this is the case, a new local segment is created, and refitted (with errors). The segments used to make this BC segment are still matched with other segments, but are discarded after that. Local segments that could not be matched with a partner in the other layer are kept.

Step 7: After fitting all local segments, a filter is applied which keeps only those LocalSegments that fulfill certain requirements. These requirements are set by an RCP parameter, and include the following options:

- All: keep all local segments;
- Bestn: keep the n best segments, best meaning those with lowest χ^2;
- Cut: keep all segments with a χ^2 lower than a given cut-off value.

The default value is to keep only the one best local segment. However, it often happens that there is a left-right ambiguity, which causes two segments to be found with the same χ^2. One of those is the correct one, while the other one is incorrect. If only the 'best' segment is taken, the wrong one might be selected. Keeping the best two segments solves some of this ambiguity, but the track reconstruction has to be able to cope with this.

The remaining segments are then transformed back to the global system by extracting the final Segment object. This transformation is the exact inverse from the transformation done in step 1.

Software implementation

This algorithm is also intended to run in the L3 framework, and therefore it has to adhere to a certain time budget, which is 50 ms for the muon reconstruction. Therefore, the aim of the segment reconstruction is to take no more than 10 ms.

To make the code as fast as possible, a couple of measures were taken. For example, no dynamic memory allocation takes place (i.e. no new and delete calls), and copying of objects is tried to be kept to a minimum. However, the code in the current state is not fully optimized. For example, there is no reuse of objects. Currently, the average time consumption of the Linked List algorithm is 12 ms/muon. This is measured with an optimized build on Unix (d0mino). The optimized NT (Linux) version is expected to be a factor faster, which would meet the timing demands for L3.

To make the code as comprehensible as possible, it was tried to use a modular design, which reflects the steps as outlined in the algorithm above. There is one main segment builder, called LinkedListSegmentBuilder. This is the object that is given the collections of hits (PDTHitCollection,

MDTHitCollection and MSCHitCollection)and returns the collection of found segments. It inherits from the SegmentBuilder interface, as it is defined in the package muo_segment, for compatibility with other segment reconstruction algorithms. When the LinkedListSegmentBuilder is called with the collections of hits, it first builds the segment hits for the local system (*step 1*). The segment hits are held in maps from MuoSectionIndices to segment hit collections, one map for the PDT segment hits, one map for the MDT segment hits. The MSC segment hits are stored in a list. Note, that these maps (and the list) are the places where the segment hits are stored; in the rest of the algorithm, only pointers to these segment hits are used to avoid copying of these objects. For each set of PDT segment hits, the PDTLinkedListSegmentBuilder is called which creates segments in the central region, for the MDT segment hits the MDTSegmentBuilder is called, which creates segments in the forward region (*step 2 through 4*). After the segments are found in each chamber (or MuoSectionIndex), the A segments are updated with the vertex (*step 5)*, and the B and C segments are matched by the SegmentMatcher to form BC segments (*step 6*). The SegmentFilter can then filter out the best segment(s) for each layer and transform the local segments to global segments, which are returned by the LinkedListSegmentBuilder (*step 7*).

Both the PDT- and the MDTLinkedListSegmentBuilder hold two lists of LocalSegments, which are used in the segment reconstruction process. When the builders are called with the wire segment hits, scintillator segment hits and an empty list of LocalSegments, they execute the algorithm as outlined above. First they loop over all the wire segment hits, to make LocalSegments of two hits, and store them in the _segment list. After this, a loop is made over all the wire segment hits, to make the trees for each hit, and to create new LocalSegments from two other LocalSegments which are stored in the _matchedsegment list. All the segments in this list are then refitted, matched with scintillators, and returned.

Each LocalSegment holds lists of pointers to each different type of segment hits that are lying on the segment. LocalSegments can be created in four different ways: from two PDTSegmentHits, two MDTSegmentHits, two other LocalSegments or by using the copy constructor. When they are created from two hits, a simple fit without errors is done to get a first estimate of the angle of the segment. For these two constructors, it is possible to also provide an angle of the segment, to make this initial fit better by using the updated positions of the segment hits (and make the first estimate a 'second' estimate). When the LocalSegments are matched in the PDT- and MDTLinkedListSegmentBuilder, new LocalSegments are created by calling the constructor with the two LocalSegments that are being matched. This constructor takes all the hits from both LocalSegments, inserts them in the correct lists (making sure no hits are listed double) and sorts the lists on ascending x-position of the hits. Then a fit is done, taking the angle of one of the LocalSegments as an initial angle to get better positions of the segment hits. The last method of interest in this class is the method getSegment(), which makes the transformation from a LocalSegment to a Segment.

Each SegmentHit is a wrapper around the reconstructed hit so that it can be used for the reconstruction in the local system. Each of the SegmentHits holds a list of pointers to LocalSegments that are to the right of the hit and start from that particular hit. These segment hits are created from a hit pointer, and internally use a GeometryHit (see package muo_hit) to access positions, drift distances and axial distances of the hit.

Results

The efficiency and resolution of the algorithm is tested on a single muon file, generated with $|\eta|<2.5$ and p_T between 5 and 100 GeV. Figure 6 shows the reconstruction efficiency as a function of η, φ and p_T of the muon.

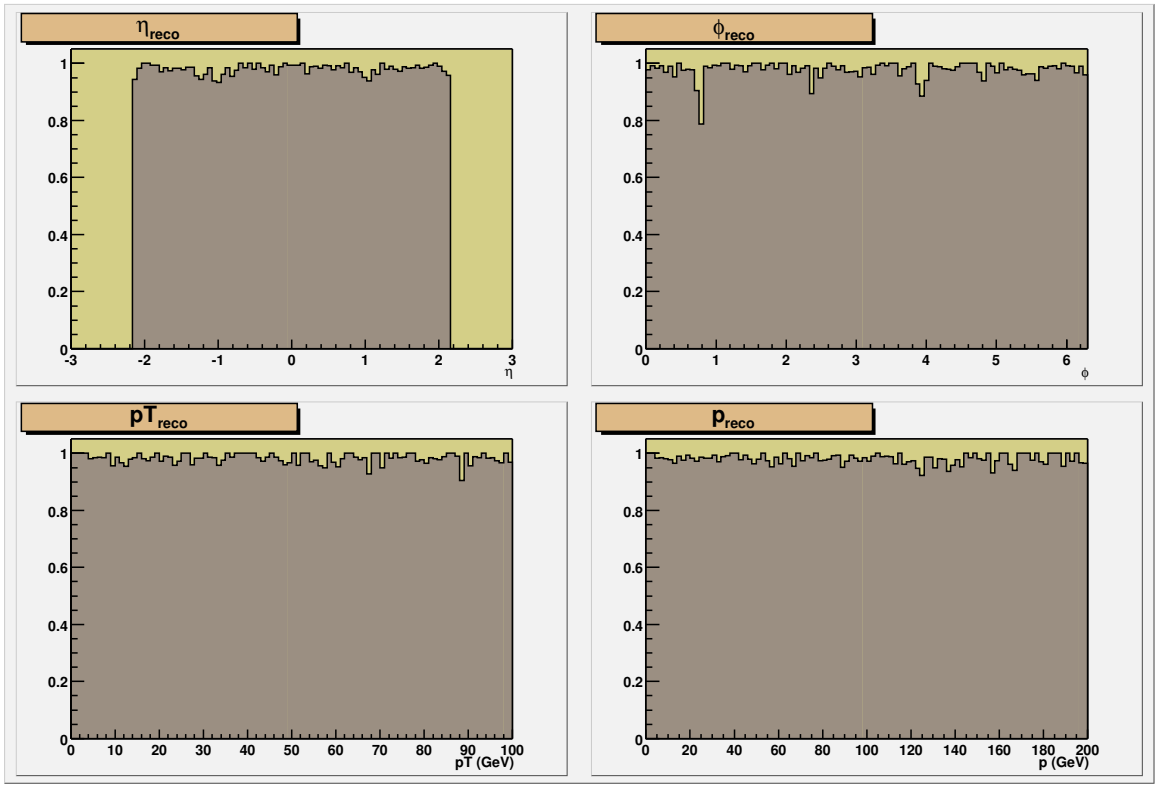

Figure 6: Segment reconstruction efficiency as function of η, φ, p_T and p of the MC muon

Overall, the reconstruction efficiency is 98.1%. The reconstruction efficiency is defined as:

$$\frac{\#reconstructed\ segments}{\#MC\ segments}$$

where a MC segment is defined as any part of the MC track that passes a muon module and generates at least two hits (wire hits + scintillator hits). This effectively cancels out the geometric acceptance. The 1.9% inefficiency is caused by hit reconstruction inefficiency and MC tracks generating one wire hit and one scintillator hit. The latter situation is not reconstructed in the algorithm.

The resolution of the found segments in the central region in η, θ, φ and the drift angle α is shown in Figures 7 and 8 for the central region, and Figures 10 and 11 for the forward regions. For comparison, the resolution of segments found by the combinatorial algorithm is shown in Figure 9 for the central region and Figure 12 for the forward region.

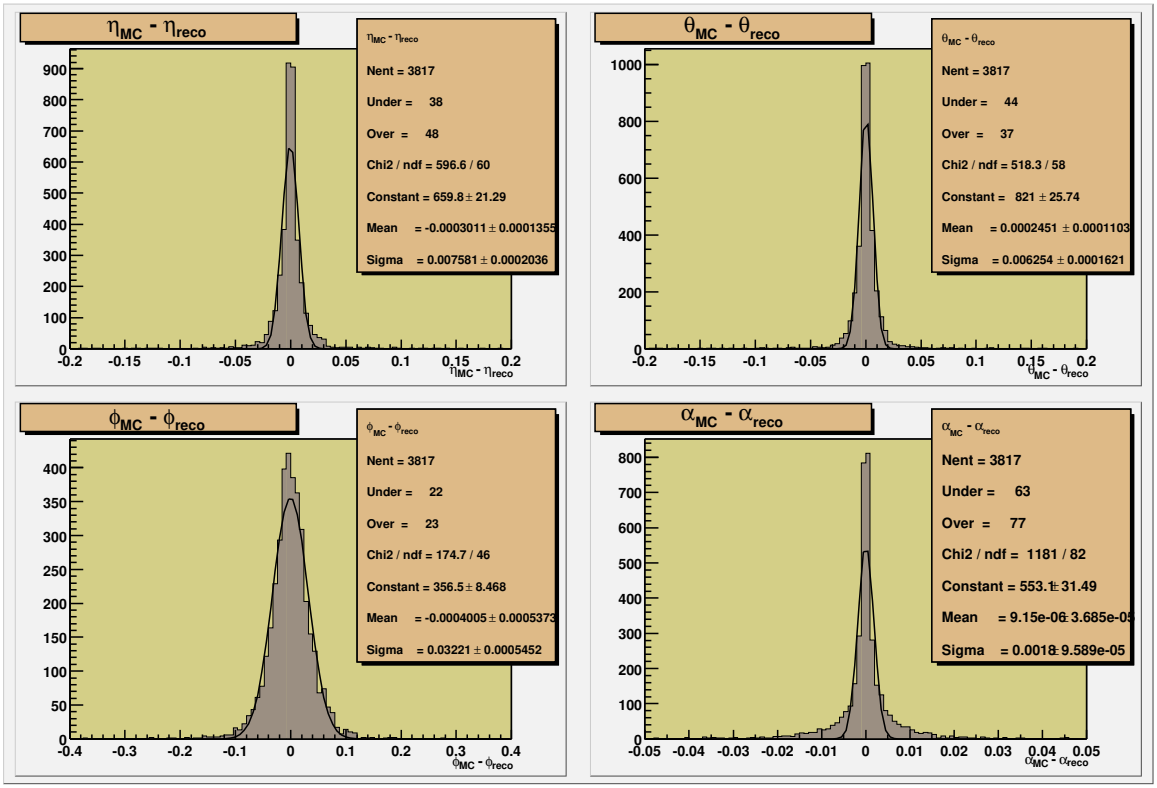

Figure 7: Reconstruction resolution of the Linked List algorithm in the central region using a vertex constraint on the A-layer segment

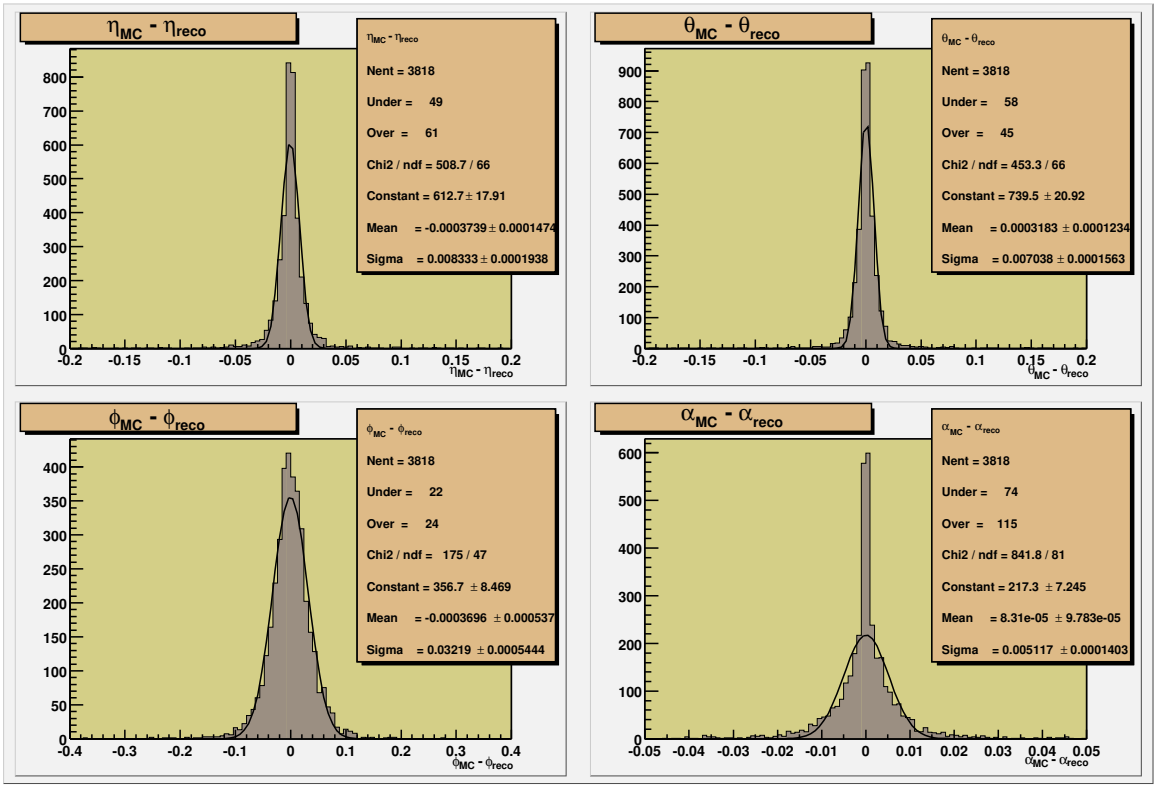

Figure 8: Reconstruction resolution of the Linked List algorithm in the central region without using the vertex constraint on the A-layer segments

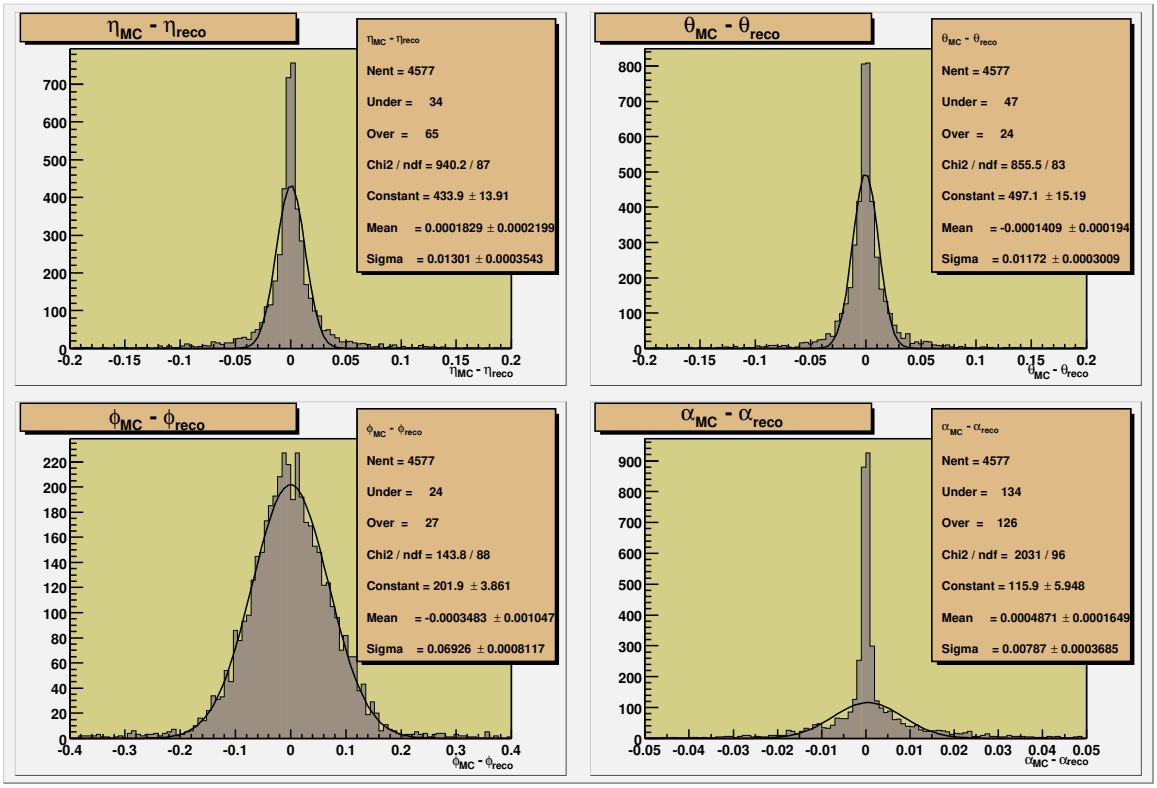

Figure-9: Reconstruction resolution of the Combinatorial algorithm in the central region

Without using any constraints like the vertex or the BC matching, the resolution on a segment in a single layer is around 5 mrad. With the BC matching and the vertex constraint, the resolution drops to around 2 mrad. How valid the vertex constraint is for low p_T tracks remains to seen. High p_T tracks have less multiple scattering in the calorimeter, so the vertex constraint does make sense for these tracks. Compared with the Combinatorial algorithm, the resolution with the Linked List algorithm is twice better in both η and φ.

In the forward region this difference is smaller and both algorithms perform almost the same. Using the vertex constraint gives a resolution of less then a millirad on the driftangle, but without this constraint the resolution jumps up to 9 mrad. Note that the Combinatorial algorithm uses this vertex constraint by default.

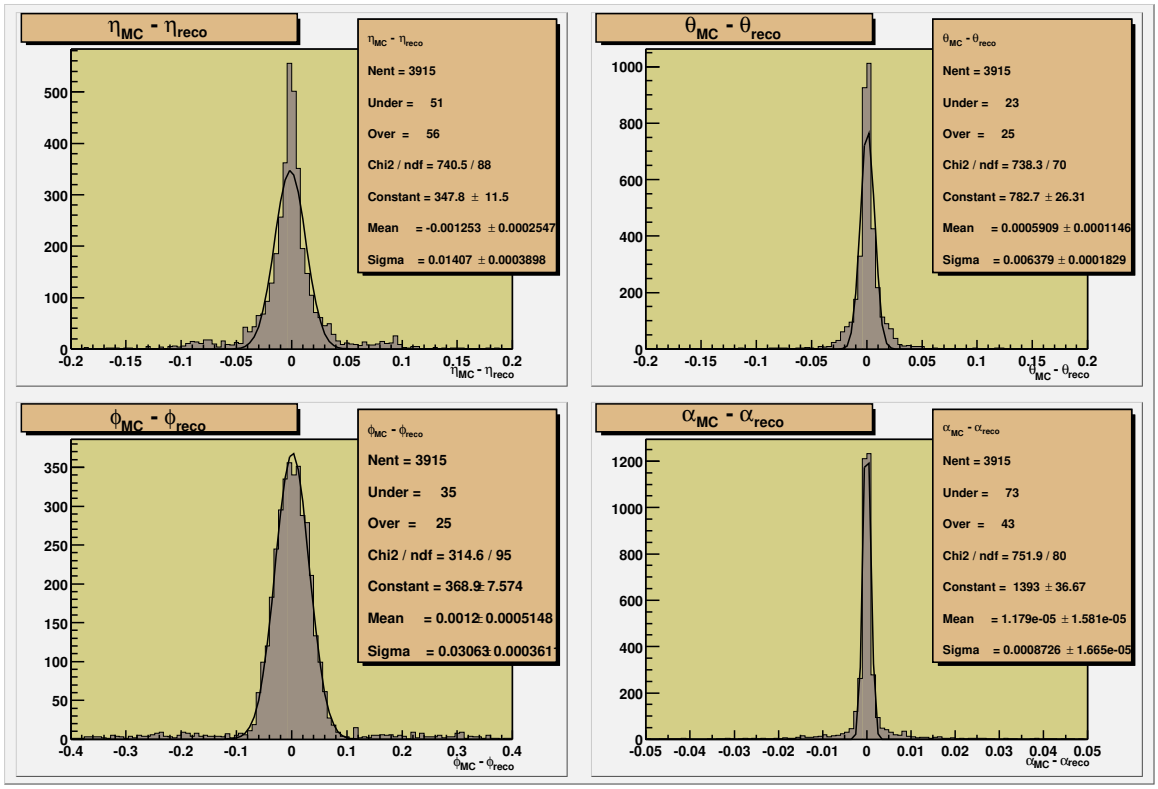

Figure-10: Reconstruction resolution of the Linked List algorithm in the forward region, using a vertex constraint on the A-layer segments

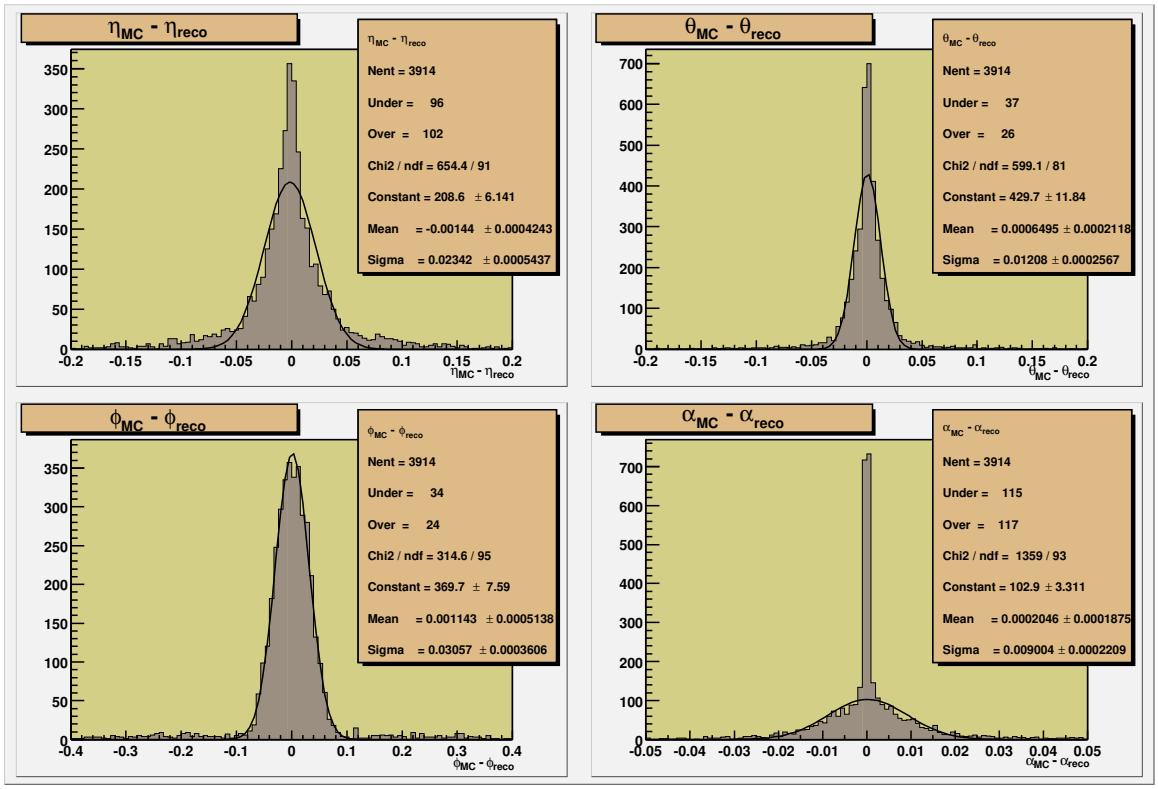

Figure-11: Reconstruction resolution of the Linked List algorithm in the forward region, not using the vertex constraint for the A-layer segments

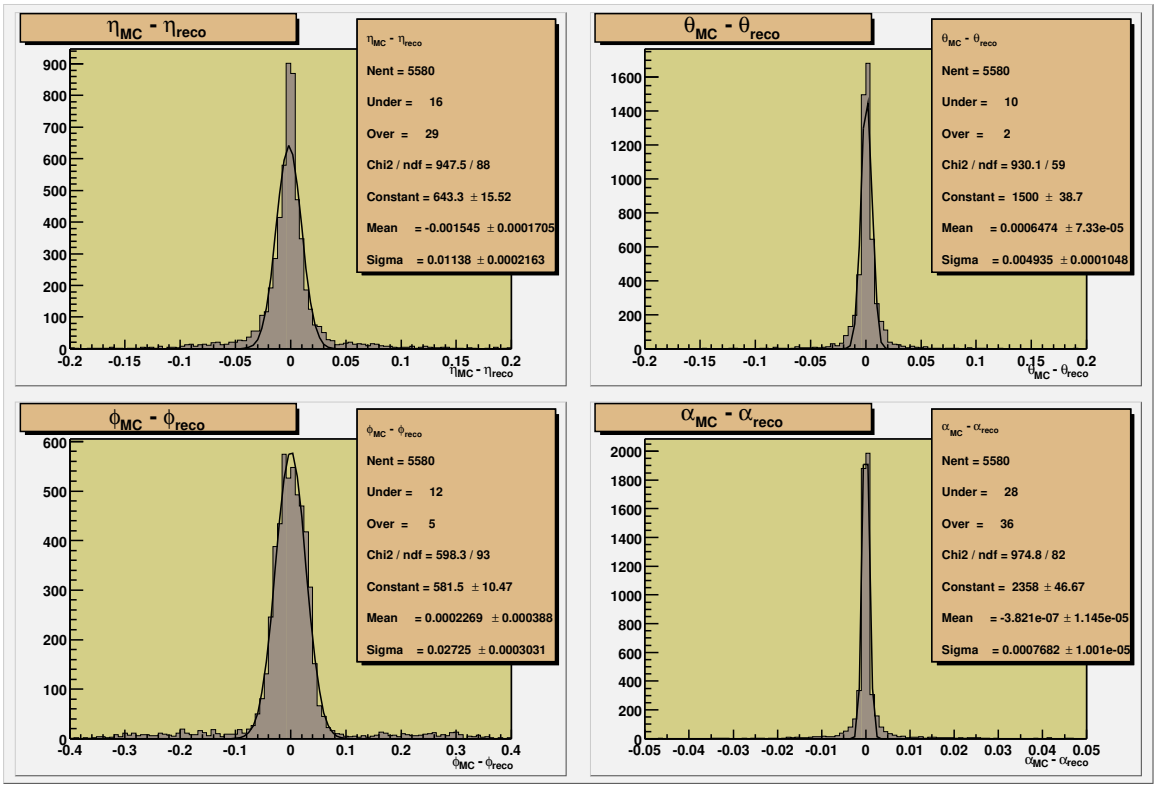

Figure-12: Reconstruction resolution of the Combinatorial algorithm in the forward region

Figures 13 through 15 show the pull distributions of the segments reconstructed by the LinkedList algorithm (the vertex constraint is not used in these plots). The 'true' error on the segment is calculated using the MC track that is passing through the chamber, and generates the hits that are used for the reconstructed segment. The position and direction of both MC and reconstructed segments are calculated at the entry point of the track in the chamber. Therefore, no pull distribution is shown for the x-position of the segment, since this position is the same by default for the MC segment and the reconstructed segment. The plots are divided in PDT segments and MDT segments, each of which is again divided into one plot for segments with a scintillator hit and one for segments without a scintillator hit.

All the pull distributions have a gaussian shape, except for the pull distribution for the z-position of MDT segments with no scintillator hit. This is caused by the fact that MDT segments without a scintillator hit have no position information along the wire at all.

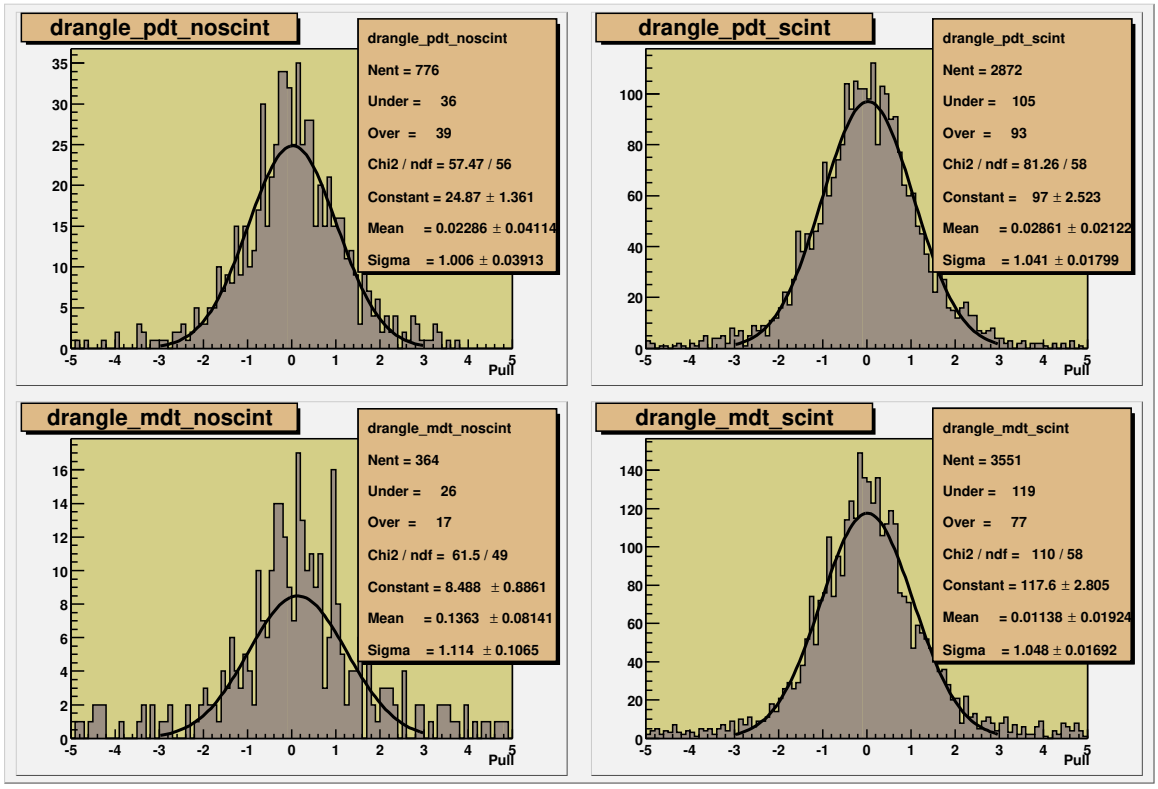

Figure-13: Pull distributions for the angle resolution

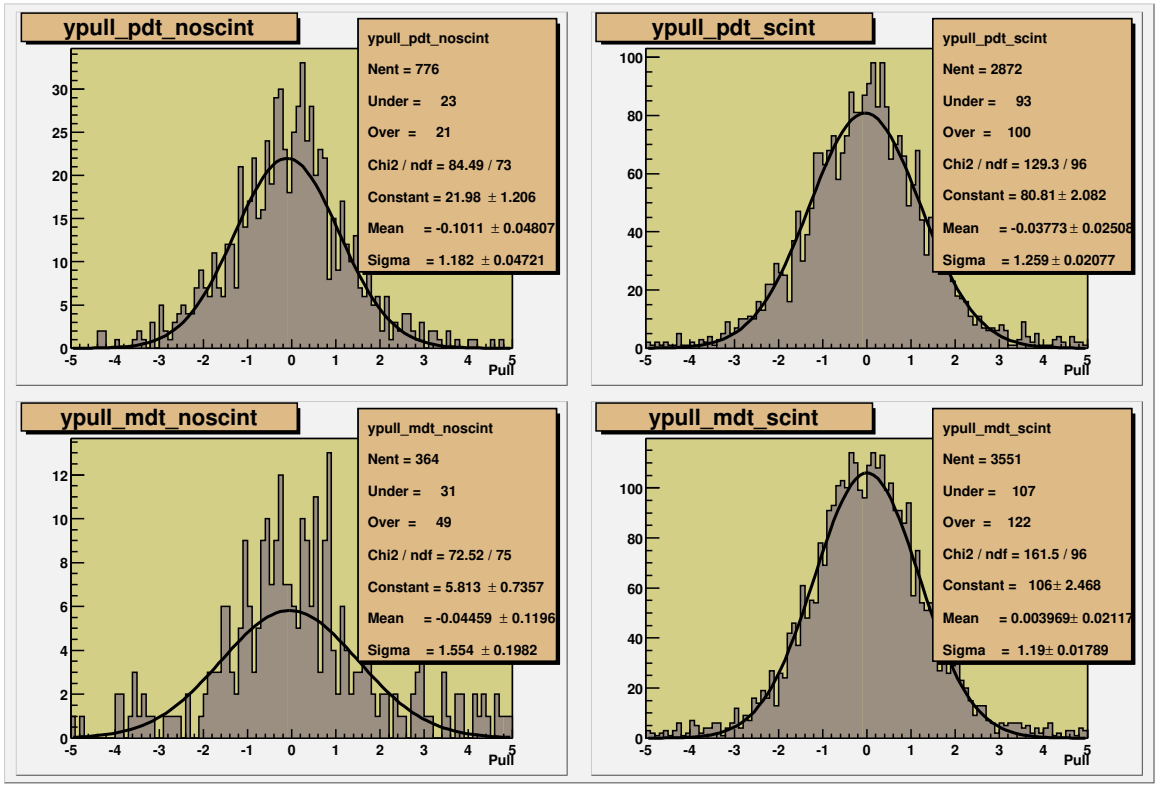

Figure-14: Pull distributions for the y-position resolution

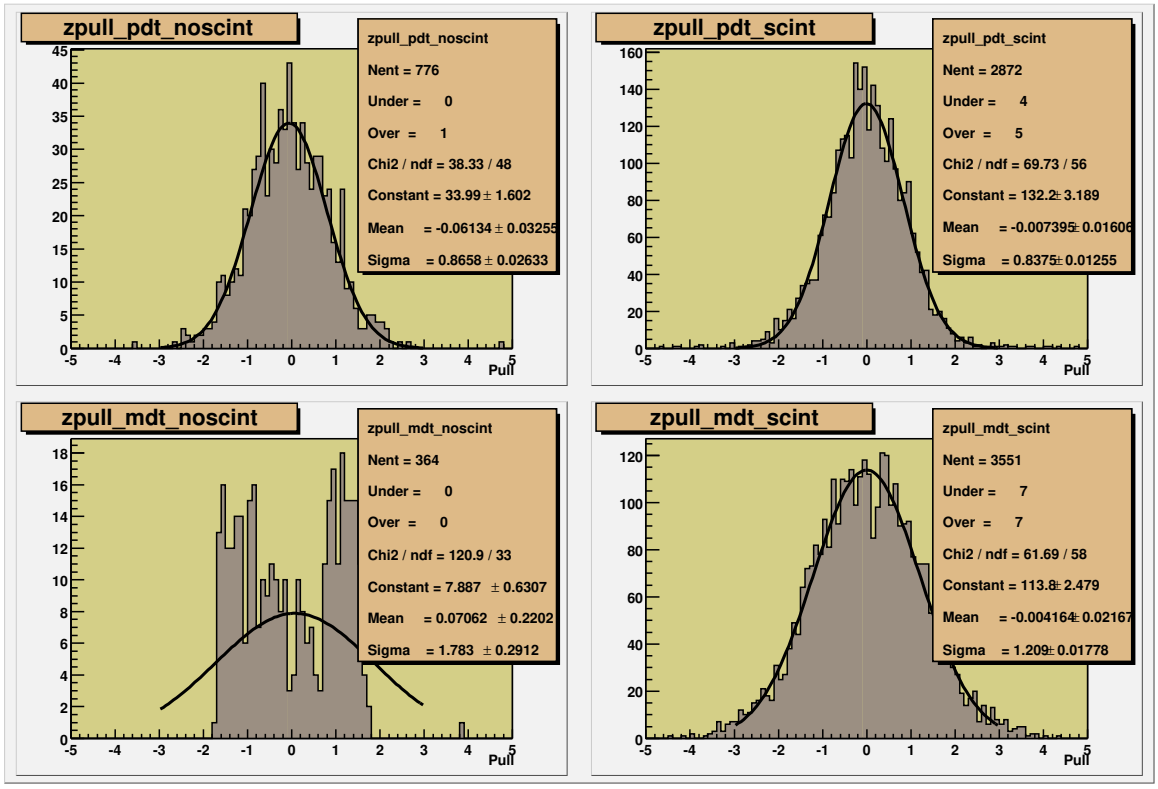

Figure-14: Pull distributions for the z-position resolution

How to run the Linked List algorithm

The algorithm used for the segment reconstruction is steered by the parameter 'SegmentAlgorithm' in the MuoSegmentReco.rcp file (in the package muo_segmentreco). To run the Linked List algorithm, this parameters should be set to "LinkedList" (default since t01.52.00).

To change the parameters that steer the Linked List reconstruction, the file muo_segmentlinkedlist/rcp/MuoSegmentLLReco.rcp has to be modified. This RCP holds the variables which are listed below.

The following are variables used when matching two local segments into one new local segment:

RCP Parameter	Function
double pdt_max_angle_diff = 0.2	Maximum angle between local pdt segments, in radians
double pdt_max_y_diff = 1	Maximum difference in position on matching plane, in cm
double mdt_max_angle_diff = 0.1	Maximum angle between local mdt segments, in radians
double mdt_max_y_diff = 1	Maximum difference in position on matching plane, in cm

These are the variables determining which segments are kept, based on some filter. There are three filters possible:

- "bestn" - keep the best n segment with the lowest chi2, n given by value of npdt or nmdt
- "all" - keep all the segments found
- "cut" - keep all segments with a χ^2 which is smaller than the cutoff value given by pdt_segment_cut_chi2 or mdt_segment_cut_chi2

All these options are possible for both the pdt and the mdt system.

Parameters for matching B and C segments into BC segments:

RCP Parameter	Function
double match_max_angle_diff = 0.05	Maximum angle between B&C segment, in radians
double match_max_r_diff = 1	Maximum difference in position on matching plane, in cm
double nsigma_segmatch = 5	Max number of sigma's when matching BC segments
double nsigma_scintmatch = 5	Max number of sigma's when matching segments with scintillators
bool doMatch = true	Regardless of above parameters, don't match BC segments if false

Parameters to bail out when the reconstruction cannot be done:

RCP Parameter	Function
int max_pdt_hits = 30	Maximum number of PDT hits per module before ignoring all hits in the module (and not doing reconstruction in that module)
int max_mdt_hits = 30	Maximum number of MDT hits per module before ignoring all hits in the module (and not doing reconstruction in that module)

Flag to determine if an A-layer segment has to be constrained with the vertex:

RCP Parameter	Function
bool useVertexA	If true, the algorithm uses the vertex constraint

How to run the D0VE display

A useful tool to display the results of the segment reconstruction is the d0ve display that is capable of displaying the muon geometry, Monte Carlo hits, reconstructed hits and reconstructed segments. To make the display executable, download the package d0ve_muon and gmake. To edit the data file that is being read in, edit the file d0ve_muon/rcp/ReadEvent.rcp. To run the display, run:

```
bin/$BFARCH/runMuonDisplay -rcp d0ve_muon/bin/runMuonDisplay.rcp
```

When the display pops up, press space to browse through the events. A typical display, zoomed in to an area of interest, is shown in Figure 12 on the next page. The little blue crosses are the Monte Carlo hits, generated by d0gstar. The red cross is a reconstructed scintillator hit, while the red circles are the drift circles of the reconstructed hits. These drift circles are merely an approximation for the equidrift lines; the real equidrift lines have a closer resemblance to ovals. The blue line is the reconstructed segment.

How to run the segment analysis package

The plots for the reconstruction efficiency and efficiency that are shown in this paper are made by the package muo_segmentlinkedlist. It consists of two parts, a binary that makes a ROOT ntuple and a ROOT macro which analyses the ntuple and creates the plots. To make the binary, take the following steps:

1. `addpkg muo_segmentlinkedlist`
2. `gmake`

To change the file that is read in, the muo_segmentlinkedlist/rcp/ReadEvent.rcp has to be edited. The ntuple is then created by giving the following command:

```
bin/$BFARCH/SegmentAnalyze -rcp muo_segmentlinkedlist/bin/runSegmentAnalyze.rcp
```

To analyze this ntuple, follow these steps:

1. `cd muo_segmentlinkedlist/macros`
2. `root`
3. `.x SegmentAnalyze.C`

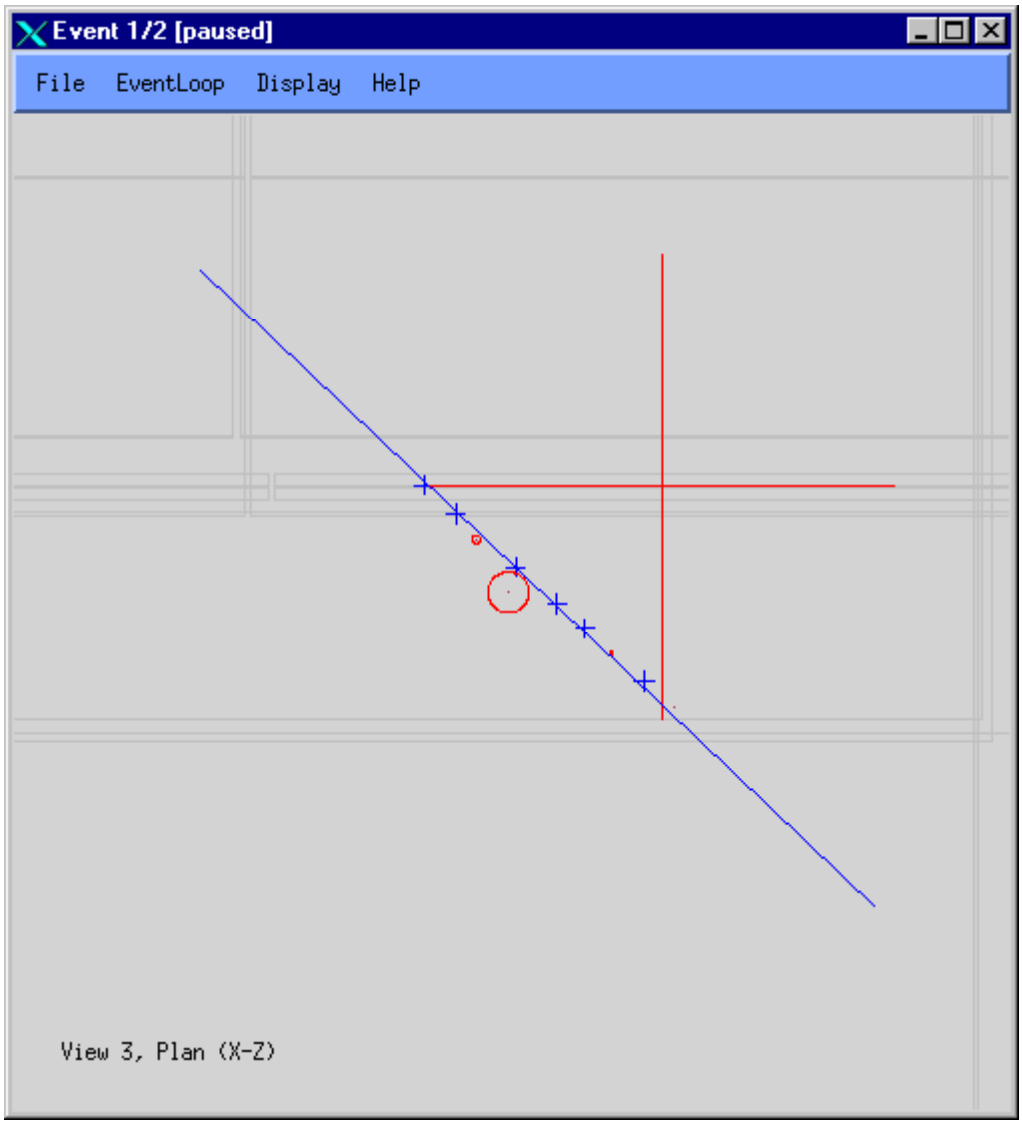

Figure 12: D0VE display of segment reconstruction. See text for explanation of display

How to measure the timing

To measure the time it takes the algorithm to run, the shell script Timing.sh, located in muo_segmentlinkedlist/bin should be run. This script will start the SegmentAnalyze program and runs over the number of events as it is designated by muo_segmentlinkedlist/rcp/ReadEvent.rcp. The output of the script is a file called SegmentAnalyze.prof. To get the timing information per event, open this file and look for the line that looks like this:

```
[41]      0.060   0.0%   89.4%     98.310  16.8%       3277
Muon::LinkedListSegmentBuilder::getSegments(const
Muon::PDTHitCollection&,const Muon::MDTHitCollection&,const
Muon::MSCHitCollection&) (SegmentAnalyze: LinkedListSegmentBuilder.cpp,
54; compiled in LinkedListSegmentBuilder.cpp00BB9D11000.int.c)
```

The fifth number (in this case 98.310) is the number of seconds it took the LinkedList algorithm to run.