LHCb 2004-094
Tracking - Reconstruction
26 Octobre 2004

Online Pattern Recognition

Olivier Callot

Laboratoire de l'Accélérateur Linéaire, Orsay, France

Abstract

This note describes the current status of the online pattern recognition, used in the implementation of the L1 and HLT triggers for the DC'04 data challenge. A detailed description of the pattern recognition method and some discussion of the algorithm's parameters are given.

1. Introduction

The pattern recognition is the code which produces tracks from the “measured” information in the tracking detectors. In this context, a track is a collection of “measurements” and a representation of the “state” (position and direction, momentum and covariance matrix) at one or several locations.

The algorithms described in this note are the result of a historical process. A first implementation of the pattern recognition was released in summer 2002, with Velo tracking and “Forward” tracking. An independent implementation for L1 of the track search in the Velo, and the Velo-TT matching, was released in spring 2003 for the trigger TDR data production. At the same time, a second generation of the Velo and Forward tracking codes was released as a first attempt to implement the HLT. After the successful completion of the TDR, it was decided to re-implement completely the two trigger codes (L1 and HLT) using a common track representation, and essentially the same code for both applications. An effort was also made to be more realistic in the data access, starting from the L1 or the Raw Buffer, a simulation as close as possible of the buffer that will be available online. An accurate speed measurement was also implemented, as one of the main goals of the online pattern recognition is to be fast while as efficient as possible. The target figure is to produce the L1 decision in 6 ms (on a 1 GHz Pentium-III processor) for minimum bias events accepted by L0, so that the same code could run in 1 ms on an expected 2007 processor. For HLT, the situation is a bit more complex, but on average the whole decision should take less than 60 ms on the same processor.

This note will describe the data handling implemented to minimize memory management, the code for the Velo 2D tracking, for the 3D tracking, the Velo-TT matching and the Forward tracking. This software is available in the `Trg` hat of the LHCb cvs repository, and can be run easily in DaVinci from version v12r0.

1.1. Notation

C++ objects and methods are indicated in bold fixed font like this: `TestMethod`

Job Options are indicated in bold, indicating the default value: **algorithm.parameter = default**.

Time measurements are given on a 1 GHz Pentium III processor, the standard lxplus interactive node.

2. Environment and tools

Several tools are common to the whole package, and are described in this section

2.1. Data handling

The usual way to store objects in the Gaudi framework used in LHCb is to create them by `new` and store their pointer in a container on the Transient Event Store (TES). Not only creating the objects takes time, but adding them to the container is also not infinitely fast, and registering the container in the TES takes another fraction of a millisecond. The overhead is small for usual offline analysis algorithms, taking tens of milliseconds, but in the L1 context every fraction of a millisecond is relevant. A dedicated approach was then implemented.

As first step of the pattern recognition, one has to convert the L1 or Raw buffers to coordinates in the relevant detector(s). Creating an object for each coordinate will use a large fraction of the available CPU, something unacceptable for L1. Instead, the code uses a `GaudiTool` that owns a vector of objects, not created at each event. Each object is set to represent the values for the current event. Provided the vector is large enough, there is no memory management involved in this case. An automatic increase of the vector's size is also implemented, which costs some CPU cycles on the first large event to be processed.

Similarly, a `GaudiTool` owns a reasonably large number of tracks, and returns a new object with almost no time, when a new track has to be created. This avoids the overhead of creating objects by `new`, and all the handling of the TES. However, the resulting tracks need to be converted eventually to TES objects, for monitoring. This is done after the chronometer has been stopped.

A generic container implementation is available in the `Trg/TrgTools` package as a templated class `TrgContainer`. The code for the track and vertex holder is available in the same package as `TrgDataProvider`. Similarly, `TrgL0MuonProvider` holds the L0Muon representation. Similar implementations are available to store coordinates; they will be listed when describing each package.

2.2. Timing measurement

The time measurement is performed using the `GaudiSequencer` built-in time facility. By setting the option '`measureTime`' of a `GaudiSequencer`, one obtains at the end of the output a table with average time, minimum and maximum time. Two ways of measuring the time are used:

- **User** time which measures the CPU cycles used by the process, but only with a 10 ms sampling, not quite adapted to monitoring algorithms taking around or below 1 millisecond per event.

- **Elapsed** time which measures the change in the system clock between begin and end of the algorithm. This one is accurate to microsecond levels, but is affected by the load of the machine on which the job runs. On a not-too-loaded system, the two measurements agree quite well.

2.3. Efficiency measurement

This is one of the important tools to understand and tune any tracking algorithm. The measurement is based on the `MCTrackInfo` information built in **Boole**, which indicates for each `MCParticle` if it has enough measurements in the various tracking detector to be reconstructed. By measurement, one considers the Digits, available in the Raw buffer. The content of the L1 buffer may be somewhat different, as some cluster may be absent due to different thresholds for keeping or not the data. This is ignored in computing the efficiency, i.e. if not enough clusters are left, this is counted as an algorithm inefficiency.

Each reconstructed track has a list of associated measurements, in the form of pairs (channelID, size) for each measurement. The Boole output contains also Linker objects [1] associating channelID to MCParticle(s). It is then easy to compute how many measurements (counting a cluster for one measurement) are associated to a given MCParticle. If more than 70% of the measurements in the Velo are associated to the same MCParticle, the Velo part of the track is associated to it. The same applies, independently, to the T station measurements. For the TT measurements, the criteria is not 70%, but “all but one” as there are usually only 3 or 4 measurements.

The other ingredient is to select which MCParticle are to be reconstructed. Several categories are used: Enough Velo measurements, enough Velo+TT, enough Velo + T stations (so called long tracks), and long tracks that are daughters of a B. This last category is the one of interest for the physics.

The whole efficiency computation is in the `Trg/TrgChecker` package, and the `TrgChecker` algorithm prints efficiency numbers at the end of every job.

3. Velo tracking

The Velo tracking can be decomposed into 3 steps: Data preparation, 2D tracking in the R-Z projection, and space tracking. A description of a first implementation of the Velo tracking method has been published in [2].

3.1. Data preparation

The data preparation differs for L1 and HLT implementation, as the input is different. However, the decoded information is in both cases `VeloCoord` objects, so that the tracking itself can ignore if it runs in the L1 or HLT context. These objects are owned by a `VeloSector`, an intelligent container handling data of a 'sector' of the Velo, either a R sector of 45° with 512 strips, or one of the two Phi regions, inner or outer. Coordinates are sorted inside a region, allowing fast access to the measurement close to an expected value. All these `VeloSector` are owned by the `TrgVeloDataHolder` tool, whose main function is to process the Velo geometry information, from the Detector Element, to cache the useful parameters of each sector, this means mainly the Phi sensor geometry. The R geometry, this means the radius of each strip, is in fact produced in the decoding of the data. These objects and tools are all in the `Trg/TrgTool` package.

This decoding is done by 3 algorithms from the `Trg/TrgTool` package, `DecodeL1VeloR`, `DecodeL1VeloPhi` and `DecodeRawVelo`. The splitting of the L1 decoding in two algorithms was done in the hope that some events can be rejected by the selection of tracks with enough impact parameter after the 2D tracking and primary vertex search. In fact, there are very few events without interesting tracks, and one could merge the two algorithms.

R measurements are converted immediately to radial coordinate. It should be noted that in the described implementation, like in the rest of the LHCb software, alignment is not yet taken into account. This also means clearly that this code doesn't work when the Velo is in its "opened" position, off axis by 3 cm, as it will be at beginning of fill. Phi measurements are the strip number times the pitch, which is signed according to the normal or inverted Phi sensors.

The decoding doesn't use the algorithm provided by the Velo group, which is just for the Raw buffer, because this algorithm produces `VeloCluster` objects stored in a `KeyedContainer`, an operation far too slow.

3.1.1. L1 Buffer decoding

The data in the L1 buffer are grouped by banks, one type for R sensors and one type for Phi sensors. For each sensor one gets the number of "clusters", each cluster being one or two strips wide. The information is the strip number, a bit to indicate that the strip as a fired neighbour, and a bit to indicate if the fired strip was over or below a high threshold. As some clusters in the R sectors can be wider than 2 strips, neighbouring clusters are merged. This is done during the decoding, assuming that the clusters are ordered in the TELL1 board, so that when many consecutive strips are fired, producing several L1 clusters, these clusters are consecutive in the L1 Buffer.

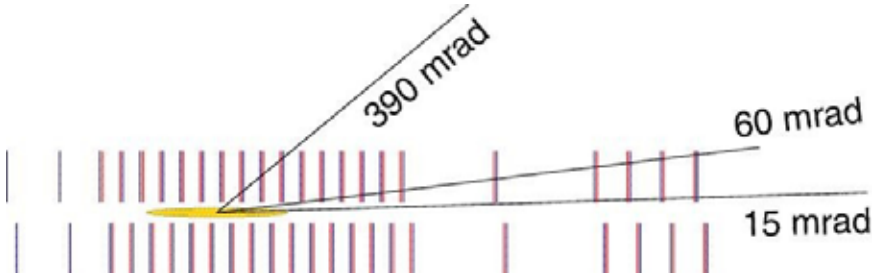
Clusters without the 'high threshold' bit are tagged. They tend to come from spill-over tracks. Tracks with too many such clusters are rejected, see later.

3.1.2. Raw Buffer decoding

This is very similar, except that the data in the buffer is made of clusters, with the first strip number, the size and the ADC content of each strip. For each such cluster, the sum and the barycentre are computed. Cluster that are too wide are rejected, this is controlled by the jobOption **DecodeRawVelo.MaxClusterSize = 3**.

3.2. R-Z tracking

The basic step in the R-Z tracking is to find a triplet of aligned clusters in three consecutive R sensors, in the same 45° sector. We don't look at tracks crossing a sector boundary. The range of slope in the R-Z plane is limited: the track should have a radius increasing with Z, and a maximal angle **VeloRTracking.MaxRSlope = 0.400** radians. This first assumption could become wrong for tracks originating far from the beam line, e.g. K_S^0 decay products. This limitation is not important for L1, and not (yet) relevant for HLT.



3.2.1. Finding triplets

Let's call S0, S1 and S2 the three sensors in the triplet. The most downstream sensor is the first sensor tried for S0, S1 is the previous one, and S2 is just before S1. This means that the search goes opposite to the track's direction, starting from the sensor where the tracks are most separated.

One iterates on the clusters in S0, from inside to outside. For each cluster, the minimum slope is defined by joining the cluster to a point on the axis, jobOption **VeloRTracking.rVertexMin = -180 mm**. One loops on the clusters in S2, asking the slope to be greater than the previously computed minimal slope, and smaller than **MaxRSlope**. To save time, the first valid cluster in S2 is memorized, and the search restarts from that one for the next cluster in S0, as previous clusters will be outside the angular range. For each pair, the position in S1 is predicted, and the closest cluster searched for. The search tolerance is defined as a fraction of the strip pitch at this radius, **VeloRTracking.rMatchTol = 0.90**. The `VeloSector` method `bestCoordinate` keeps track internally of the first good cluster in the search range, and restarts from it the next search, with automatic detection when this assumption becomes invalid. Loops are aborted as soon as the rest of the clusters cannot satisfy the condition. Having sorted cluster lists is the key point of the speed of the algorithm.

In order to avoid finding again the same piece of track when starting from the next sensor, the first cluster should be unused, i.e. not part of a previously found good track.

3.2.2. Track extension

Once a triplet is found, it is extended as much as possible, the radius in the next sensor is predicted and the cluster closest to the prediction searched for. The search window is now **VeloRTracking.RExtraTol = 3.5** times the pitch. This large value is needed to accept non

pointing tracks, for which the R-Z projection is not exactly a straight line. It also helps to reconstruct entirely low momentum tracks with large multiple scattering. The predicted radius is computed using a simple linear fit of all previously found R coordinates. The search is continued until no cluster are found in **VeloRTracking.MaxMissed = 3** consecutive sensors or until the extrapolation is outside the sensor. These tracks starting inside the Velo can be good tracks, like converted photons, interaction in the material of decay in flight of neutral particles.

3.2.3. Clones at vertical boundary

The two halves of the Velo have some overlap. Tracks in this region are then measured twice, on the right and on the left sensors. To decrease this number of clones (they are the same track, but don't share any cluster), a dedicated search is performed if the track has already more than 4 clusters: When in one of the sectors in this overlap region, clusters in the corresponding zone of the other sensors are searched for, with a search window of only **VeloRTracking.rOverlapTol = 0.6** times the pitch. If at least N-1 (N is the number of cluster on the original track) such clusters are found, they are added to the track, and tagged as used. Tracks produced this way will certainly be very useful for aligning the detector. The tolerance will probably have to be increased in the first attempts, when the misalignment can be large compared to this tolerance.

3.2.4. Tagging and storage

If the triplet is not extended, all hits have to be unused, as the probability to have a ghost in this case is quite high. For longer tracks, the track should have at least 2 clusters not already used on other tracks. If the track has **VeloRTracking.MinToTag = 4** or more clusters, they are tagged as used.

As sensors can be inefficient, we search also triplets with a missing sensor, namely (S0, S1, S3) and (S0, S2, S3) but only for end cluster not used in a previously found track. This avoids finding three times the same track. The cost in time of this recovery of inefficiencies is quite low, as long as the sensors are over 95% efficient efficient.

After having exhausted all combinations, the starting sensor S0 is changed, going towards the interaction point, until sensor number **VeloRTracking.MinFirstSensor = 30**.

A second pass is performed in the opposite direction; this means starting from the first sensor, looking for so called backward tracks, useful for vertexing. Exactly the same algorithm is used, except that clones at the vertical boundary are not searched for.

Before the final storage in the `TrgTrack` format, a filter is applied (only relevant for the HLT version) on the average charge of the track, to remove spill-over tracks: The fraction of clusters with charge over **VeloRTracking.ChargeThreshold = 10** is counted, and the track is rejected if this fraction is less than **VeloRTracking.HighChargeFraction = 0.5** of the number of clusters.

The track is stored, with reference to the Velo channel ID used, using the `type` to indicate the R sector the track was in, and also if it was a backwards track: type = sector + 10.

3.3. Space tracking

The space tracking is somewhat more difficult, as is the geometry of the Phi sensors. The idea is to collect hits in Phi sensors which, when associated to an R-Z track, make a straight line in space. Of course, it is difficult to select immediately which Phi cluster to associate. The trick is to build lists of compatible clusters, and once all clusters in all sensors have been processed, to select the best list. Reducing the number of combinations is the issue. Fast decision is also important.

3.3.1. Prepare the tracks

The first operation is to convert the valid `TrgTrack` to working tracks that have to be extended in space. The list of `VeloCord` used is reconstructed from the stored Velo channel ID. All valid R-Z tracks are processed, one after the other. Tracks are sorted by length, i.e. by number of R measurements. This allows to search first for the best tracks, and to remove their clusters for future searches.

3.3.2. Select sensors

First, the first and last Phi sensors that can be crossed by the track are computed, simply by taking one more station on each side, and testing the radius at the sensor's position. Iteration on the sensors starts again by the one most distant from the production point. The two halves of the Velo are handled simultaneously, as tracks close to the vertical boundary between the two halves will have part of their points on one side, part in the other. Until the track is found there is no method to avoid searching on both sides, except of course for tracks in the two central quarters of each R sensor: In this case, the only possible Phi hits are in the sensor of the same side.

3.3.3. Building the list of Phi clusters

For each sensor, the coefficients to convert a Phi strip to a Phi coordinate, and the range of Phi coordinates values, are computed, from the boundary of the R sector in which the track sits, plus a tolerance **VeloSpaceTracking.PhiAngularTol = 0.005**. A quick test on an empty range allows skipping Phi sensors without overlap with the R sector.

For the first two sensors (one on each side), a new `VeloPhiList` is created for each Phi cluster.

For the next sensors, a match with all existing lists is tried for each cluster. If there is no match and the cluster was never used, a new `VeloPhiList` is created, but only for the first 3 pairs of sensors. After that, the track would have too many missed hits and it is not worth adding more lists to test.

For each cluster, the operation is first to convert the strip number to its angle, which gives then a point (x,y) in space. The matching with existing `VeloPhiList` is performed by comparing the distance in space between the predicted trajectory and the point..

Note that the radius at the point is computed from the R-Z projection of the track, using an interpolation of the R measurements, and the matching is performed using the projected x-z and y-z straight line parameterization of the `VeloPhiList`. The tolerance for this comparison is controlled with the parameter **VeloSpaceTracking.PhiMatchTol = 0.20 mm**, which indicates the error in position for a normal station spacing, and is scaled by the ratio of the distance between the last measurement and the current one over the normal separation of 60 mm. When there is a single measurement already in the list, the tolerance is simply 0.15 times the radius.

A difficulty is to allow multiple combinations: for each `VeloPhiList`, only the best cluster for a given sensor is kept. The same coordinate can be used in several `VeloPhiList`, and new list should be created if the coordinate is not used (when in the first tested sensors). When a new cluster is kept, the parameterization of the track is adjusted, using the (x,y) point corresponding to the cluster. R measurements are used only indirectly, because they are used in converting Phi to (x,y).

When all Phi sensors have been scanned, the best `VeloPhiList` is selected. This is the one with more clusters, the one with the best χ^2 if several candidates have the same number of clusters. Of course a minimal length is requested, **VeloSpaceTracking.FractionFound = 0.35** of the sensors, which means twice this number of stations, should have a cluster.

3.3.4. Final fit and storage

Once all the R and Phi hits are collected, the track is fitted as a straight line using all coordinates. In order to take into account multiple scattering, the weight of each point in the fit is damped by a damping factor. This means that the measurement error is increased by `MSError`, where `MSError` starts at zero and is incremented by **VeloSpaceTracking.stepError = 0.002** for each measurement. Last, the track is converted to the storage format of `TrgTrack`, using the results of the fit for defining the first state and its covariance matrix.

4. Velo – TT tracking

The second step of the online pattern recognition is to get a momentum estimate using TT. As there are only 4 planes in TT, and inefficiency must be accepted, there are not many constraints. The basic assumption is that the effect of the field can be replaced by a kick of the track at a fixed Z. The method is then to extrapolate each track as a straight line to this Z plane, and then project (using a straight line from this point) all TT measurements on a fixed plane. The correct solution is an accumulation in this projection plane.

As usual, the first step is to decode and prepare the data, with two versions, from the L1 buffer or from the Raw buffer. The choice is controlled by the option **VeloTTAlg.InputFromRawBuffer = false**. The whole code (decoding and pattern recognition) is in a single algorithm `TrgVeloTT` in the `Trg/TrgVeloTT` package.

In order to find hits efficiently in TT, data are grouped by horizontal bands, described by the object `TTPlane`. In this object, all measurements cover a similar vertical range, and they can be sorted in X. The bands are defined using the `RowID` method of the ST Detector Element object `TTLayer`. This is quite similar to the Velo sectors, an intelligent data holder allowing fast access to the data. Of course one should look at more than one band if the vertical extrapolation of the track is compatible with two bands.

4.1. Data preparation

Data is decoded from the L1 or the Raw buffer using the tools provided by the `IT/STDAQ` package. From the L1 buffer, one retrieves the channel ID, the size, and a flag for high threshold. From the Raw buffer, one retrieves the list of ADC content in place of the high threshold flag. This is very similar to the Velo information. As first filter, hits without the high threshold bit (L1) or with a total signal less than **VeloTTAlg.HighThreshold = 6** are ignored.

4.2. Pattern recognition

The search is performed independently for each valid Velo Space track. The various `TTPlane` are looked at, one after the other. If the track can have a hit in this region of the detector (**VeloTTAlg.yTolerance = 0.5 mm**), the track is first extrapolated to the average z of the `TTPlane`. This is used to check if the track is in the sensitive area of TT. This means outside a square hole of size **VeloTTAlg.centralHoleSize = 38 mm**, and inside the overall plane, **VeloTTAlg.maxXSize = 660 mm** and **VeloTTAlg.maxYSize = 580 mm**. The search window in X is defined as a constant term **VeloTTAlg.xTolerance = 0.35 mm** and a linear dependence with the angle with the beam axis **VeloTTAlg.xTolSlope = 350 mm**. This basically defines a minimal transverse momentum window, as the momentum is lower at larger angle with the beam axis. The width of the window is limited to the deflection on the minimal momentum track, defined by **VeloTTAlg.minMomentum = 1.5 Gev**.

The track is extrapolated to the z plane of the measurement, for each measurement, as the Z varies from wafer to wafer. The distance between the extrapolation and the measurement, measured perpendicularly to the strip, is compared to the tolerance, both projected to the nominal plane (average of the TT planes): they are scaled by the ratio of the z distances of the actual and nominal planes to **VeloTTAlg.zMidField = 1620 mm**. This value is determined by fitting MCHits in a stand-alone program. Hits are then stored on lists, merging compatible hits with a tolerance

VeloTTAlg.matchTol = 0.35 mm plus **VeloTTAlg.matchTolSlope = 0.02** times the absolute distance between the hit and the extrapolation. This is how the X and stereo hits are matched together. In fact, we maintain two independent lists of grouped hits, one for TTa and one for TTb. For each group, the presence of hits for both planes (x and stereo) is detected. This is not just counting the number of hits, as there may be two hits from the same track in the same plane, due to the overlap between adjacent wafers.

Once the two lists are built, they are matched to get the best candidate. First, only pairs with all 4 planes fired are searched for. They have to be compatible, this means the measured distance has to differ by less than **VeloTTAlg.finalMatchTol = 2 mm**. But there may be several pairs matching this criterion. The pair having the best χ^2 of compatibility is the one selected. If there is no solution with 4 planes, one tries with only 3 planes. Only one solution is kept.

Once the search is finished, the candidate is stored. If the track doesn't cross the 4 planes,it is also accepted according to the option **VeloTTAlg.AcceptInHole = false**. The momentum is estimated, q/p is estimated as the product of the distance in the reference plane times a coefficient **VeloTTAlg.distToMonetum = 0.00031** which is related to the integral of the field between the Velo and TT. The TrgTrack object is produced, and re-fit is performed, using lookup tables to get the field integral and the position of the centre of the field, depending on the Z of origin of the track and the Y slope. This eliminates some wrong momentum estimates, so called "p_T mistakes" that would be bad for L1 decision.

4.3. Improvements

This fast estimate of the track's momentum is a key component of the trigger. The current implementation has some limitations that should be addressed in the future

- A single track is found for each Velo track. This is a limitation, for example converted photons have two tracks, which are very frequently overlapping in the Velo. Finding two solutions of opposite charge in case the Velo charge is indicating a double ionisation may be useful.

- The tolerance in Y should probably depend on the estimated momentum, i.e. a butterfly search window may be better than a rectangle as it is now. The price to pay is probably more ghosts and a slower algorithm.

- Matching the two lists is done a bit blindly. There are inefficiencies that one should understand and fix: Velo-TT is the seed of the Forward tracking, which should find all physics tracks!

5. Forward tracking

This is the most complex algorithm, and the most time consuming one. A first version, used for offline tracking, has been already described in [3]. But since then the algorithm itself was largely changed, to take into account new ideas. In particular, the search is performed using a histogram and no longer trying to find local space points. The code is now packaged as a tool, doing the tracking for one track, and an algorithm looping on tracks. This code has been largely modified several times, and it is time to rewrite it from scratch, so that it can be maintained again for some time.

5.1. Principle of the method

The forward tracking starts from a Velo track, and tries to find hits in the T stations such that the momentum of the track can be estimated. Knowing the Velo track, and a point in the T stations, the whole trajectory is defined. This is true provided the field is homogeneous enough, which is the case. This means that one can project to a common plane all T hits, assuming they come from the tested Velo track. The first step of the pattern recognition is thus just a projection of all hits, histogramming the positioning the common plane, and trying to find a peak in this histogram. Once a peak is found, this defines an initial trajectory. All hits compatible with this trajectory are collected, and an iterative process fits the trajectory and removes the worst hits, until the fit is good or there are no longer enough hits.

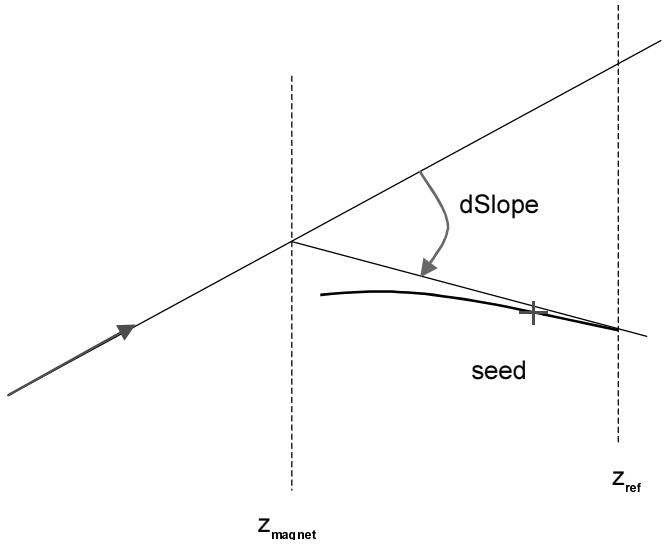


Figure 1 : Definition of the main parameters

Clearly, one key ingredient is the way to project a hit to this common plane, and (similar but not identical) the parameterisation of a trajectory starting from a point in the common plane. A dedicated program, `HltFitParams`, available in the same package `Trg/TrgForward` with options file to run it, is used for that purpose: The Velo track are reconstructed, the true hits in the T stations of the MCParticle associated to this Velo track are fitted (cubic in X, parabola in Y), and a parameterisation of these coefficients is obtained. With a magnet without fringe field in the T stations, the trajectory there would be a straight line, crossing the Velo extrapolation in the centre of the field. Two parameters would be needed: The position of the centre of the field z_{magnet}, and the deflection dSlope, inversely proportional to the momentum, as displayed on Figure 1.

As there is some field in the T stations, the trajectory is curved, and the curvature is basically proportional to the deflection. All the parameters have dependencies on the vertical and horizontal slopes in the Velo, and on dSlope. Using an Ntuple storing all track properties and the fitted polynomial coefficients, the dependencies of the coefficients are guessed. Then the program fit the value of the parameterisation of these polynomial coefficients. One difficulty is that dSlope depends on z_{magnet}, and z_{magnet} depends on dSlope. A small iterative computation is needed; it converges in a few iterations.

5.2. Data preparation

As usual, the data should be retrieved from the Raw buffer (the forward tracking is not available for L1, for speed reasons), and put in the appropriate structure. For each of the 4 “planes” of the 3 stations, the measurements can be in the upper or lower part of OT, or in one of the 4 IT boxes. In each of these collections, the measurements can be sorted according to the X coordinate.

Each of these collections is a `FwdPlane` object, which has the geometry of the region: limits in X and Y, information from the Detector Element to convert a strip / straw number to a position. These collections are owned by the tool `FwdHitProvider`, whose initialisation process the Detector Elements and stores the geometry.

The decoding is performed by the algorithm `FwdHitFromRawBuffer`. It owns the `FwdHit` that are then passed to the `FwdHitProvider` tool, where they are made available to the reconstruction algorithm. The decoding of the IT hits uses the tools from the IT/STDAQ package to get the layer, wafer and strip. The decoding of the OT hits is done manually, as there is no generic tool for that. This is a place where improvement is needed. A foreseen change in the OT coding of the Raw buffer will require a serious re-implementation.

This algorithm has no options.

5.3. Pattern algorithm

The algorithm `TrgForwardAlg` is a simple interface to the tool `TrgFowardTool`, which does the real work. The algorithm is basically a loop on the Velo tracks, to try to reconstruct each of them. It also contains a global clean-up procedure, to remove clones: If two found tracks share too many T hits, even from a different Velo track, the worst one is removed. The comparison is based on

- The number of T station hits. The one with less hits is removed
- The χ^2 of the fit. The worst one is removed if the difference is larger than **TrgForwardAlg.DeltaKhi2 = 3**.
- The quality of the slope matching. If the difference of error on the slope is larger than **TrgForwardAlg.DeltaDiffSlope = 4**, the worst one is removed.

This cleanup reduces the number of ghosts, with a limited effect on the efficiency.

The algorithm starts in fact from the Velo-TT tracks, using the momentum estimate obtained there to restrict the zone where to search T hits. It also converts the found tracks to their TrgTrack representation, including the computation of a state after the T stations, for extrapolation in RICH2, calorimeter and muon systems. Basically, the estimated momentum is added to the Velo state. A possible improvement would be to re-fit the Velo track knowing the momentum, and to compute properly the covariance matrix, using estimates of the amount of material between the last Velo point and the production vertex.

The algorithm performs a two passes processing, first with stronger cuts to be fast and identify the easy tracks, then with relaxed cuts, ignoring already used hits, for tracks with some inefficiencies.

5.4. Pattern recognition tool

This is the core of the processing. This is also a really ugly piece of code, to be re-engineered as soon as possible. The tool has a lot of jobOptions, to define all the cuts, and also to pass the parameterisations computed as described in section 5.1. The processing is performed in 4 steps:

- Initialisations for the track
- Histogramming the T hits (X planes only)
- Searching for peaks in the histogram
- Collecting all hits for each candidate peak candidate

These steps will be described now in more detail.

5.4.1. Initialisation for the track

Internally, the patter recognition is done using a working track of type `TrgLongTrack`. This object is initialised with the input track's parameters (position, slope, error on slopes, momentum estimate) and the algorithm parameters, which are passed as a large array. This contains the parameterisation of the various polynomial coefficients.

5.4.2. Histogramming the T hits

First, a minimal momentum is computed, to restrict the search window. It is defined as **TrgForwardTool.SeedMinMomentum = 2 GeV** and **.SeedMinMomentumTrans = 80 MeV**. If the track has some momentum, the search window is restricted to ± a fraction **TrgForwardTool.TolQOverP = 0.2** of the deflection due to the estimated momentum, plus a constant corresponding to the displacement of a 10 GeV particle.

A loop is performed on all X layer `FwdPlane` compatible with the Y extrapolation. For each unused hit in the range previously computed, the compatibility between the hit's Y range and the extrapolation is checked again, with a tolerance **TrgForwardTool.uTolSlope = 170 mm.GeV** divided by the momentum. This is to take into account multiple scattering. The hit is extrapolated to the reference plane, using the parameterisation:

- dSlope is computed as the difference of the Velo slope and the slope between the seed point and the extrapolation of the Velo track to a default Z_{magnet}.
- Z_{magnet} is corrected from its dSlope dependence.
- The point at Z_{magnet} is recomputed, the seed point is projected to the reference plane, and then corrected by the product of dSlope times the sum of a quadratic and a cubic terms in dz (distance between the hit and the reference plane)

This has to be fast, as this operation is done basically the number of tracks time the number of hits!

The histogram has **TrgForwardTool.NbBins = 2000** bins, with **TrgForwardTool.BinSize = 3 mm**. As only the wire position is used for OT hits, the bin size should not be too small.

The histogram is filled with a weight. For IT hits, the weight is 1. For OT hits, as in principle there are two straws fired per plane, the weight is only 0.5, except when at the boundary between top and bottom parts: There is a dead area in this region, put as ± 50 mm.

5.4.3. Peak finding

The previously filled histogram is searched for peaks. To avoid boundary problems, the sum of two consecutive bins is used. When this sum is over a threshold, **TrgForwardTool.MinInPeak = 4.4** or **TrgForwardTool.MinInPeak2 = 3.4** for the two passes, a seed is found. Its position is computed by interpolating in the two bins. But if this seed is too close to the previous one, it is ignored: This is the case when a single bin is over threshold…

Seeds are sorted by peak height, so that the clearer candidates are looked at first. Note also that tracks having no good peak in the first pass are tagged, to be ignored at the second pass: The peak can only be lower! Speed concern again.

5.4.4. Track finding

This is the most complex, and dirty part… First, the `FwdLongTrack` object is set with the proper parameter, i.e. the selected seed. Then all compatible X hits are collected. The tolerance has a constant term (2.1 mm for IT, 5.1 mm for OT) plus a momentum and distance dependent term **TrgForwardTool.XTolSlope = 8 mm** times the z distance to the reference plane, divided by the momentum.

The number of different x planes should be at least **TrgForwardTool.MinPlanesX = 5** or **TrgForwardTool.MinPlanesX2 = 4** for the second pass. The track is fitted, and the worst χ^2 contribution is computed. As long as there are still enough x planes and one hit has a χ^2 higher than **TrgForwardTool.MaxKhi2 = 10** plus **TrgForwardTool.MaxKhi2Slope = 20** divided by the momentum in GeV, the worst hit is removed and the procedure repeated.

Having this acceptable x candidate, the stereo hits are collected. The Y tolerance is the same **TrgForwardTool.uTolSlope** (used already in 5.4.2) divided by momentum. The track is fitted again, with the same removal of the worst hit until good enough, or until there are less than **TrgForwardTool.minPlanesTot = 10** planes fired on the track.

This procedure involves complex methods of the `TrgLongTrack` object. In particular, setting the seed triggers the complete computing of the cubic and quadratic parameterisation of the x and y trajectories. Computing the worst χ^2 implies fitting the track, which is in fact a modification of the seed and a re-initialisation with the previous method for the X parameterisation, a modification of the Velo slope for adjusting the Y parameterisation.

5.5. Final processing

Once the track has been found, it is stored in the `TrgTrack` format. The first state is modified to add the momentum estimate. A new state is created at a fixed position after the T station, which contains the parameterisation of the track after the field, to permit extrapolation to RICH2, calorimeter and muon detectors. This state is deduced from the fitted points by applying a correction for the remaining magnetic field. This correction is determined using the stand-alone program mentioned in section 5.1, fitting the true hits of muons in the muon detector, and finding the change of slope and position compared to the parameterisation in the T station.

It should be noted that a performance improvement in the quality of the tracks could be obtained by re-fitting at this stage the Velo tracks, knowing their momentum, so that multiple scattering correction can be properly implemented. This is one of the possible improvements of this algorithm.

6. Performance

Two types of performance are relevant: The efficiency to find tracks, and the speed of the algorithms, as explained in the introduction. But the types of events on which these two measures have to be performed also differ: The efficiency is relevant on B decay tracks, assuming there will anyway be enough tracks to get the primary vertex. The speed has to be measured on minimum bias events having passed the previous levels of triggers, as it is these types of events the code will be confronted to in the experiment.

6.1. Speed measurement

The most critical test is the L1 code, as the budget is 1 ms in 2007, 6 ms on the standard 1 GHz Pentium III processor available on LXPLUS. The tracking should only be part of the budget, as the L1 trigger has also to compute the primary vertex, select which tracks to reconstruct in Velo 3D according to their impact parameter or matching to L0 muon candidates, and also to take the final decision. The values are given in Table 1.

Algorithm	Average time (ms)	Tracking time (ms)
Event initialisation	0.30	
`DecodeL1VeloR`	0.34	0.34
`VeloRTracking`	0.90	0.90
`L1PrimaryVtx2d`	0.40	
Track selection	0.24	
`DecodeL1VeloPhi`	0.26	0.26
`VeloSpaceTracking`	0.90	0.90
`TrgVeloTT`	1.48	1.48
L1Decision	0.34	
Result storage	0.19	
Total	5.35	3.88

Table 1 : Speed performance for L1

This means that the achieved speed is well within budget. The non-tracking part takes about 30% of the time budget. It should be noted that one important parameter for the overall speed of the processing is the number of tracks to be reconstructed in 3D and in VeloTT, selected form their impact parameter and their matching to L0 Muon candidates.

The speed performance for HLT is less critical, as a filter will be applied before doing the complete tracking: Only the interesting tracks will be reconstructed in Velo-TT and Forward, to confirm the L1 decision with improved momentum accuracy. According to recent estimates, this could reject two thirds of the events, and the complete tracking will be performed only on the remaining third, with a more comfortable budget. Also, the time taken to select the events is not yet known.

The result is given in Table 2, measured on minimum bias events accepted by L0 and L1. The limited statistics imposes to give the numbers with limited accuracy. One should note that, despite being executed with the same algorithm on all the tracks of the event, the `VeloRTracking` algorithm is now about 1.5 times slower. This is due to the fact that events selected by L1 tend to have higher multiplicity and thus to be reconstructed slower. The decoding is also slower, this is also partially explained by the somewhat more complex processing of each cluster, as the barycentre is computed on the ADC data available in the Raw buffer. But the overall figure is in the allowed range, even if progress is still needed. The Forward algorithm takes two third of the time, and is then the first target for improvements

Algorithm	Average time (ms)
`DecodeRawVelo`	1.2
`VeloRTracking`	1.5
`VeloSpaceTracking`	7.0
`TrgVeloTT`	9.6
`FwdHitFromRawBuffer`	3.9
`TrgForwardAlg`	47.1
Total	71.0

Table 2 : Speed performance for HLT

6.2. Efficiency measurement

As always for efficiency, a proper definition of the sample on which it is measured is essential. Here, we concentrate on the efficiency of B decay tracks, as this is our physics goal. Compared to other tracks, they tend to have higher p_T and thus higher momentum. The data sample used is $B \rightarrow J/\Psi(\rightarrow\mu\mu)\Phi(\rightarrow KK)$. Events should have passed L0, and L1 for HLT measurements. Tracks are associated to `MCParticle` using the standard criteria as described in section 2.3. Ghosts are tracks not associated to any `MCParticle`, as made of clusters from various particles, and/or from noise. Several tracks can be associated to the same `MCParticle`, they are called clones, and this is a small fraction, at the level of 1 %. The values are listed in Table 3. One should note that the selection of interesting tracks before Velo space tracking has been removed from the L1 sequence, to have a proper definition of the set of tracks that should be reconstructed.

Type of tracks	Ghosts L1 (%)	Efficiency L1 (%)	Ghosts HLT (%)	Efficiency HLT (%)
Velo R	6.5	98.1	10.3	99.1
Velo Space	5.4	94.9	7.3	97.3
Velo-TT	11.9	85.1	12.3	92.4
Forward			3.8	88.0

Table 3 : Ghost rate and efficiency for B decay tracks, L1 and HLT

7. Summary

This note has shown that LHCb has now an implementation fulfilling the speed and efficiency requirements for the online tracking, both for L1 and for HLT. But a lot is still to be done, in terms of increased efficiency and robustness. Even if the speed is in the correct range, progress, mainly for the Forward tracking, is still needed. One should also note that other tracking algorithms may eventually be needed in HLT, for example, if triggering on K_S^0 is required.

References

[1] Olivier Callot, "*A new implementation of the Relations: The Linker objects*", **Note LHCb 2004-007**

[2] Olivier Callot, *"Velo tracking for the High Level Trigger*", **Note LHCb 2003-027**

[3] Maurice Benayoun and Olivier Callot, *"The Forward Tracking, an Optical Model Method"*, **Note LHCb 2002-008**