Detection and reconstruction of short-lived particles produced by neutrino interactions in emulsion

Uiterwijk, J.W.H.M.

Chapter 4

Track finding in emulsion

In Chapter 2 the setup of the CHORUS experiment has been introduced, including a hardware-based method to scan emulsion. This chapter describes the development of a new approach undertaken at CERN. The principles in this approach were to use off-the-shelf hardware and to use software as much as possible. This required the development of track-finding code for the particular case of emulsion images. Electronic detectors usually yield information in one or two projections; emulsion yields 3-D position information. Electronic detectors typically measure only a few track points; emulsion tracks typically contain many hits. The difficulty with emulsion is that these hits are burried in a large number of background hits.

An algorithm that could efficiently find tracks in this high noise environment was developed. Although originally written for track finding in emulsion, the algorithm and its tools could be used in more general applications and have, therefore, been implemented as an object-oriented C++ toolkit. Part of this chapter is a copy of a published paper describing this toolkit [233]. In this chapter, the algorithms and implementations of the track-finding classes and the container classes developed for fast searching in multi-dimensional spaces are presented. The track-finding efficiency, estimated using a Monte-Carlo simulating, is also presented. The expected performance of the algorithm has been investigated. The tracking code was originally designed to reconstruct all tracks. However, in the scan-back stage of event location, a track-selector like approach (section 2.9.4) is sufficient and faster. This was also implemented in software and is described in section 4.5. Finally, the application to real emulsion data is presented.

4.1 Introduction

Automatic emulsion scanning with computer-controlled microscope stages and digital read-out and processing of emulsion images was pioneered by the the FKEN laboratory in Nagoya (Japan). As is described in section 2.9, the Nagoya approach to emulsion scanning is based on a brute force, hardware based, track-finding system which examines a fixed set of 16 images. Originally, only a track with known slope could be located automatically. With the development of ever faster hardware, this restriction disappeared because the hardware could simply check for many slopes.

When one examines the emulsion-scanning strategies used in CHORUS in detail (section 2.10), three different stages can be distinguished: scan-back, net-scan and eye-scan. These stages differ in the area and thickness scanned and whether all tracks or only a single track is being searched for. During scan-back (section 2.10.2), a single track is looked for and the area scanned is large on the interface sheets but very small on the target sheets. During net-scan (section 2.10.3), all tracks are reconstructed and the area is large. In both stages, it is sufficient to examine only a thin slice of one emulsion surface to find all interesting tracks. The exception is scan-back on the interface sheets where both surfaces need to be scanned.

The net-scan procedure has a short-coming which becomes apparent for events with a secondary vertex or kink. Net-scan is comparable to electronic tracking detectors in the sense that tracks and vertices are reconstructed from a few measurements along the paths of the tracks. The complete particle track or the actual vertex is not seen. From the net-scan data alone, it is impossible to tell if a secondary vertex was caused by a decay or an interaction. The net-scan procedure can also not distinguish between the decay of a charged or neutral short-lived particle if the decaying particle does not cross the upstream surface of at least one emulsion plate. These limitations re-introduced human-eye scanning in the emulsion analysis. The advantage of net-scan is that now only a small sample of events needs to be scanned at a well known location in the emulsion and with a partially known topology. During such eye-scanning, one to several plates are examined through their full thickness and the tracks and vertices of interest (some of which are already known) are inspected, measured and registered in a computer readable format. Eye-scan corresponds thus to a scanning stage with small to medium areas but full thickness.

During the development of the scanning and track-finding hardware in Nagoya, the optics and the limitation to 16 images have never changed. So even though the scanning speed has increased several orders of magnitude (including the CCD camera speed), the optics still limit the field of view to about $150\,\mu\text{m} \times 150\,\mu\text{m}$ and the hardwired 16 image limit restricts the scanning to emulsion slices of around $100\,\mu\text{m}$ thick. Historically of course, the track-selector was designed for doing scan-back only; it was the increased scanning speed which led to the development of the net-scan procedure. Net-scan is probably close to the best that can be done for full automatic event reconstruction given the limitations of the hardware and a given time frame determining the time that can be spent on each event.

Within the CERN scanning laboratory, the idea took root to redevelop automatic scanning techniques, keeping the ideas that had already been developed while avoiding known limitations and human eye-scanning as much as possible. The guiding principles in these developments were to use up-to-date instrumentation and off-the-shelf electronic com-

ponents wherever possible and implement as much as possible all pattern recognition in software. Using off-the-shelf components and implementing pattern recognition in software, allows one to profit directly from Moore's law,[1] while avoiding the long development time and relative inflexibility of home-built designated hardware.

The main subject of this chapter is the software developed for track-finding in a set of emulsion images. Another pattern-recognition problem improved in the CERN developments was the location of reference points on the emulsion plates. This has already been addressed in section 2.10.1. In the next sections, some of the new instrumentation developed for the CERN scanning laboratory will be briefly described, before returning to the main subject with a discussion of the characteristics of emulsion data and the constraints these place on a tracking algorithm.

4.1.1 Microscope optics and stages

Normal microscope optics are designed to render an accurate image of an object to the smallest detail possible usually using white light. On the contrary, for the reconstruction and measurement of charged particle tracks in emulsion, the shapes of the grains are not important; the only relevant parameter for each grain is its position. For emulsion scanning, the optical system should yield an image of sufficiently high contrast such that grains can readily be identified and of sufficiently high resolution, both transversely and axially, such that their position can be accurately determined. The typical grain size in the CHORUS emulsion is $0.8\,\mu$m. Given a typical dimension of pixels on image sensors of around $10\,\mu$m, the transverse resolution dictates a magnification of around $40\times$ to have about three pixels per grain. The depth of field, the size of the axial region that is in focus, should be below $2\,\mu$m such that the z position of individual grains can be determined with reasonable accuracy. In order to scan large areas, the field of view should be as large as possible, while the field curvature should be minimal such that imaged slices in the emulsion are basically flat planes. The free working distance, the distance between the first lens surface in the system and the object in focus, has to be more than 1 mm to be able to scan the full-thickness of an emulsion plate.

In the 1970s, Tiyoda designed an objective specifically intended for emulsion work, on request from and in collaboration with Nagoya University. This $50\times$ oil-immersion lens represented a compromise between automatic scanning and comfort for eye-scanning. It was designed with a numerical aperture (NA) of 0.85 using green light at 550 nm. A higher NA or shorter wavelength would give a better resolution, but it also decreases the contrast making grain recognition more difficult for the human eye. Its field of view is free from distortions up to a diameter of about $200\,\mu$m, which is about the maximum a (trained) human can quickly oversee. The practical depth of field for grain recognition is about $2.6\,\mu$m.

A comprehensive study of the optics required for emulsion scanning [234], showed that a larger field of view could be achieved with a new optical design purely intended for automatic scanning. In collaboration with industry [235], a new optical system was developed with as goals a field of view of $500\,\mu$m diameter and a depth of field of $1.5\,\mu$m. The different refractive index of the various types of emulsion plates required that the

[1] Moore's law, posed in 1965, states that the number of transistors on integrated circuits doubles every 18 months which is accompanied by a similar increase in processor speeds. For various takes on this not-so-constant law, see `http://www.answers.com/topic/moore-s-law`.

optics could be tuned to deal with these differences. These specifications were realized with an oil-immersion objective with a NA of 1.05 using a blue light source at 436 nm (g-line of a mercury-vapour arc-lamp). It can accommodate a variable refractive index between the object and the front surface of the objective lens, within the range $1.49 < n < 1.54$, by moving a group of lenses inside the objective, which contains a total of eleven lenses. The magnification is selectable from 28×, 40×, 60×, and 80× by exchanging an adapter tube. The high NA and short wavelength ensure good resolution in both transverse and axial directions, even at the minimum magnification of 28×. With a typical one square centimeter image sensor, the actual field of view is $350\,\mu\text{m} \times 350\,\mu\text{m}$, seven times larger than the $150\,\mu\text{m} \times 120\,\mu\text{m}$ field of view of the Tiyoda lens and CCD system used in Nagoya. The field curvature is less than $1\,\mu\text{m}$ up to the very edge of the field of view. The practical depth of field for grain recognition is about $1.2\,\mu\text{m}$, more than two times better than the Tiyoda objective. A more extensive description of the optical system can be found in Refs. 215, 236.

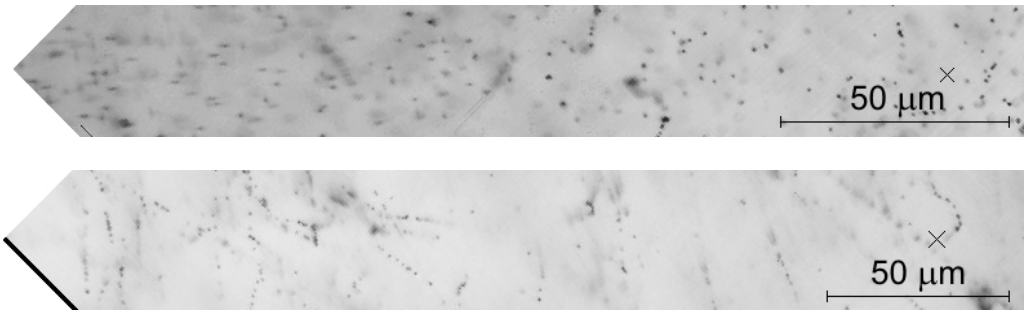


Figure 4.1: Mega-pixel camera view of a piece of emulsion with the 50× Tiyoda objective (top, width about $220\,\mu\text{m}$) and the 40× Jenoptik objective (bottom, width about $280\,\mu\text{m}$). The optical axis is located at the right-hand side of the images and indicated by the small black crosses. The Tiyoda objective suffers from clear radial distortion at longer radial distance from the optical axis (left-side of image), while the Jenoptik objective shows no such imaging artefacts. From the images, one can see that the Tiyoda objective has a higher contrast (due to lower NA, higher magnification, and longer wavelength) making it much easier for the human eye to spot the grains. The black line and area in the images are camera defects.

Due to the smaller depth of field of the new optics more independent images can be taken inside an emulsion layer. For the same reason, the z resolution of the grain positions is also increased. For scanning $100\,\mu\text{m}$ thick slices of emulsion, typically 20 to 30 independently imaged planes inside the emulsion are taken, called layers. The larger field of view reduces the number of views to be taken when scanning large surface areas. For small areas, the processing can be restricted to the central area of the image. As there is no hard limit on the number of images that can be taken, the eye-scan stage can be replaced by simply taking images through the whole depth of the emulsion. With a $3\,\mu\text{m}$ layer spacing, this gives between 100 and 120 images per side of the target emulsion plates.

Even though the new optics allow more of the emulsion to be viewed per operation, the amount of data collected in a single view is still only a very tiny fraction of the total information on a single plate. A single pixel in the view of the microscope covers

a volume of about $0.35\,\mu\text{m} \times 0.35\,\mu\text{m} \times 3\,\mu\text{m}$ of the emulsion. Considering that for scanning purposes its value can be given by a single bit as simply black or white, each target plate contains about 250 terabits of data. The 2304 plates in CHORUS contain thus 570 petabits of data. This corresponds to 344 years of continuous black and white TV images of 1024×1024 pixels at 50 Hz frame rate. From this amount of data, the need for a hybrid detector is clear as it is impossible to scan all the emulsion, see section 2.9.1. Even just scanning the predictions per emulsion module, the grain position data generated by scan-back and net-scan is several terabytes per module, with a typical burst data rate of about 2 megabyte per second. These data volumes and rates require some thoughtful design of computing and storage infrastructure. For example, interface-plate scan-back and net-scan data is normally processed offline on a cluster of computers, while scan-back in the target sheets can be handled online.

Another, straightforward, development was the introduction of a bigger microscope stage with a stroke of $40\,\text{cm} \times 80\,\text{cm}$ that accepts one complete emulsion sheet and that can handle the much heavier objective. Some minor technical upgrades are the use of an immersion-oil containment device and the introduction of plate holders that facilitate quick exchange of the plate on the microscope stage. Figure 4.2 is a photograph of one of the CERN microscopes in its latest configuration with the new optics and a new faster CMOS camera.

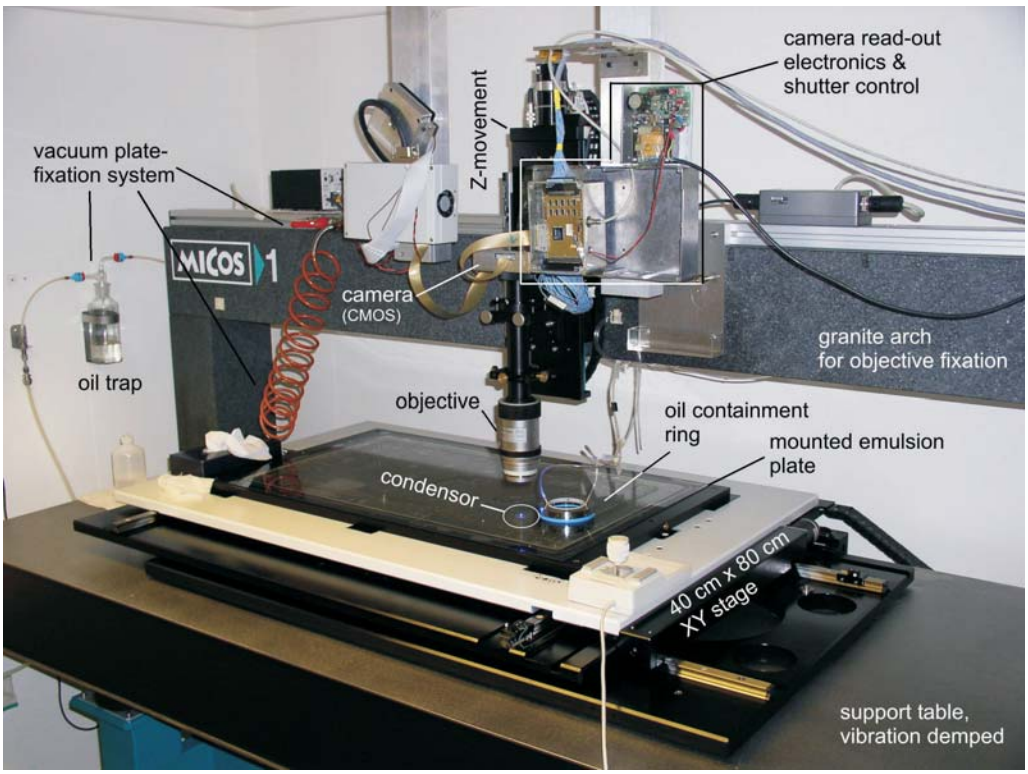


Figure 4.2: Photograph of an automated scanning microscope system at CERN. The photograph shows several of the components discussed in the main text.

In the CERN scanning setup, the 'image processing' and 'track finding' units in Figure 2.17 have been replaced by a digital signal processor (DSP) board and tracking software. Modifications have also been made to the 'offline analysis' unit in this figure, which are discussed in section 4.6.2. The DSP applies a digital filter to the images to recognize the grains. The processed images are then transferred to the control computer. On the control computer, a fast clustering algorithm reconstructs the grain positions from the pixels in the processed images. These grain positions are either stored in the database for later processing or fed directly to track-finding software running online. The tracks found online are also stored in the database. In the CERN setup, the 'database' block in Figure 2.17 is an object-oriented database. The data model is comprised of classes that store predictions, acquisition parameters, compressed grain data, several types of tracks, reference points, and alignment data with many references between them. The database is both read and written by the online scanning program and by the offline analysis tools. A detailed description of the instrumentation of the CERN scanning laboratory can be found in Ref. 215.

4.1.2 Tracking input characteristics

The track-finding's job is to reconstruct particle tracks out of the grains in a set of tomographic images. The input to the track-finding consists of processed grain data which just contains the 3-dimensional grain positions, referred to as hits. The largest part of the scanning results consists of 20 to 30 layers, where each layer contains about 4000 hits. The xy-resolution (in the plane of a layer) of the hit coordinates is of the order of $0.5\,\mu$m. The z resolution is defined by the layer spacing and is about $4\,\mu$m. In this sense, emulsion data are not much different from a multi-layer 2-D electronic detector, like silicon pixel layers, although with much better resolution and layer spacing. The typical emulsion thickness used for tracking is about $100\,\mu$m in which a track has about 30 high-resolution 3-D hits (for CHORUS emulsion).

Track reconstruction would be straightforward if the 30 track hits were not hidden in about 1200 other background hits. A typical volume of CHORUS emulsion data on which track finding needs to be performed, contains of the order of 10^5 grains. Of these, only about 2500 belong to interesting tracks. The noise consists mainly of randomly developed grains (fog) and low-energy tracks. Distortion of the emulsion implies that tracks can only be considered straight on a scale of about $20\,\mu$m, which complicates the track finding. Fog and distortion have been explained in section 2.9.2.

4.1.3 Algorithm restrictions and requirements

Due to distortion, the track's direction changes gradually over a distance of around $20\,\mu$m. Position correlations between track hits are therefore only well defined for a sequence of about 5 to 10 hits. This leads naturally to a track-finding algorithm which looks only at hits in close proximity to hits already considered as part of a track. The large total number of hits limits the time a algorithm can spend on investigating each hit. Therefore fast algorithms are required for retrieval of hits by position and for acceptance calculations. The close-range relationships and fast look-up can be achieved by constructing a connection network of links between neighbouring hits. Building this network, however, still requires finding all hits in the neighbourhood of each hit. To speed up this operation a set of multi-dimensional search tools were developed. These tools are based on the extension of a binary-search tree to multi-dimensional space. These tools, implemented as ordering containers in any dimensional space, are described in section 4.2.

The track-finding algorithm uses such a 3-D ordering container for creating the links network of close hits. The network is searched for patterns consistent with particle tracks. Conceptually, the method is based on selecting the best path of connected hits in a tree of all track-compatible paths from a certain starting point. The actual implementation follows more closely a depth-first search algorithm from graph theory [237]. The algorithm is described in detail in section 4.3.

4.1.4 Toolkit abstraction

The implementation of the algorithm is general enough that it can be used for any situation where points, not necessarily 3-D, have close-range correlations. Setting up the links network and searching it do not require any direct knowledge of the hit or track model. The algorithm only requires yes or no decisions for hit acceptance and a way of comparing track candidates. In the C++ programming language, these decisions are easily isolated by calling them as abstract methods of a class representing a track segment. The track-finding algorithm can therefore be implemented as an object-oriented toolkit. The user has to implement the concrete class for doing the acceptance and comparison calculations. In the implementation of the decisions, the calculation time can be balanced with the tracking input characteristics. This allows one to tune for a particular background condition or to tune the track-finding efficiency by considering more paths through the links network.

The toolkit is currently used in two applications: in CHORUS it is used to reconstruct tracks in emulsion; in HARP [238, 239] it is used to reconstruct bent tracks in the magnetic field of a time-projection chamber. These two applications use the same tracking toolkit, but a different implementation of the hit acceptance class. In CHORUS, the implementation is tuned to be efficient in an environment with a large number of noise hits. Because of the redundancy of track hits, high hit-to-track assignment efficiency is not required and therefore strict cuts are applied to avoid including noise hits. In HARP, the implementation takes into account the track curvature due to the magnetic field.

4.2 Multi-dimensional ordering containers

In general, a tracking algorithm in k-dimensions (abbreviated to k-D) requires a k-D look-up mechanism to search for other hits in a certain range near a given hit. A simple and fast algorithm for retrieving elements in a given range in 1-D is described first. In section 4.2.2, the properties of binary-search trees and their extension to more dimensions is presented. These trees are used to construct containers for ordering elements in k-dimensional space. The range look-up algorithm can be extended to k-D space using these containers. The implementation of the k-D containers is described in section 4.2.4. In section 4.2.5 a summary of the performance with respect to a simplistic approach is presented.

4.2.1 Find-in-range algorithm

Finding all elements in a set $\mathcal{P}$ of unique[2] numerical values which lie within a given range can be done fast if the elements are sorted. For an ordered set $\mathcal{S}$ with elements p_i,

[2] In practice identical values can be included.

$i \in [1, n]$, which has the property that

$$\forall i \Rightarrow p_{i-1} < p_i \quad , \tag{4.1}$$

it follows that

$$\forall k > 1 \Rightarrow |p_i - p_{i\pm1}| < |p_i - p_{i\pm k}| \quad , \tag{4.2}$$

where $|p_i - p_j|$ represents the distance between elements p_i and p_j. Equation (4.2) states that the element with smallest distance to some element p_i is one of its neighbours in the sorted set. To find all elements in a given range can then be done by locating the first element larger than the lower bound of the range using a binary search, which runs with an upper limit of $\log n$ in time. One then takes the following elements in the set as long as they are below the upper bound of the range. The time for sorting the set $\mathcal{P}$ has an upper limit of $n \log n$. Because the tracking requires a range search for each hit, the sorting time amortized over all searches is of the order of $\log n$.

This algorithm cannot be extended directly to more than one dimension, because the distance operator in equation (4.2) is not valid for vectors. There exist strict weak ordering operators defined on the set of k-D points that can be used in equation (4.1). However, none of these will leave equation (4.2) valid if the absolute difference is interpreted as a distance. The underlying reason is that there exists no mapping of a k-D space to a space with less dimensions that also maps distances. To make equation (4.2) valid for vectors, one would have to order them in a Voronoi tessellation [240–242], where each point occupies a volume whose borders are determined by the closest points around it. The time needed by the fastest algorithm to build a Voronoi tessellation is proportional to $n \log n$ for 2-D space and proportional to $n^{\lceil k/2 \rceil}$ for $k > 2$ [243].

4.2.2 Search trees

In a multi-dimensional space another range-finding algorithm is required, because of the impossibility to satisfy equation (4.2). The sorted sequence of equation (4.1) can be realized as a binary-search tree [237, 244]. In a binary-search tree, each node contains a value and has a left and right branch to sub-trees. The left sub-tree contains all smaller values than that of the parent, the right sub-tree all larger values. A node with no branches is called a leaf. The value stored in the node is usually associated with other data and is therefore often called a key. The time for a key search has an upper limit of h, where h is the height of the tree (number of levels). In balanced trees, the leaves are at almost equal height $h \propto \lg n$, with $\lg n \equiv \log_2 n$. Algorithms exist to build balanced trees in a time with an upper limit of $n \lg n$. Values can also be efficiently retrieved in sorted order from a tree by a walk through its nodes.

Although sorting in multiple dimensions is not possible, the concept of splitting a range can be extended to more dimensions. The equivalent of splitting a 1-D range into sub-ranges at some key value, is splitting a cube into 2^3 sub-cubes in 3-D. Each sub-cube is then the root of a 3-D sub-tree for an octant of the space around the parent's 3-D key value. This kind of trees are generally known as k-ary-trees or Kd-trees. Here, the space covered by a sub-tree for multi-dimensional trees is referred to as a sub-volume, independently of the dimension of the space.

A balanced k-D tree with n keys has a height proportional to $\log_m n$, where $m = 2^k$ is the dimensionality multiplicity. Balancing operations (see Ref. 237) rotate nodes, as shown in Figure 4.3a, to ensure that the sub-trees of each node are approximately of

the same height. In 1-D trees this rotation involves three sub-trees. Such a rotation is needed, at maximum, twice per insert of a key in a tree. The rotations can be done in constant time because the shaded area in Figure 4.3a that moves over to node 1 in the rotation, corresponds exactly to the sub-tree γ of the new top-node 2. Rotations in more than 1-D require a complete restructuring of a large part of the tree and can therefore not be done in constant time. As can be seen in Figure 4.3b, in two (or more) dimensions, a rotation involves partial areas of the old top-node 1 which have to be mixed with existing sub-trees of the new top-node 2. Only the sub-trees α, β, and γ are unaffected by the rotation.

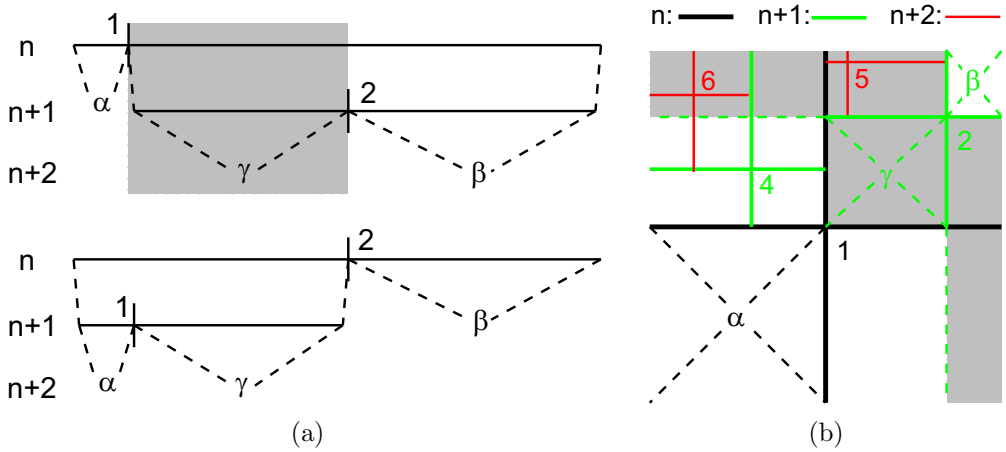


Figure 4.3: (a) Rotating the nodes 1 and 2 in a 1-D tree requires moving the shaded area. As this area is just the left sub-tree of node 2, the complete sub-tree γ can be set as the right sub-tree of node 1. The sub-trees indicated by α and β are not affected by the rotation. (b) In a two-dimensional tree the nodes 1 and 2 cannot be easily exchanged as the shaded area, which would become sub-areas of node 2, overlap with **parts** of sub-areas of node 1. Nodes 4, 5, and 6 need to be redivided as would all of their children.

A tree in which the points are inserted in random order has on average a height with an upper limit of $\log_m n$ However, in track-finding applications, the hits are usually sorted (by detector layer, row and column coordinates) and a tree could, in the worst case, degenerate to a linear sequence which has a look-up time of order n. A simple solution to avoid this kind of unbalance exists when the range of keys is known beforehand and the keys are more or less uniformly spread over the range. In this case one can build a binary tree in which all keys are stored in the leaves and internal nodes split their range through the middle. A node controls the range it covers and is in one of the following three states: it is either empty; it holds a key somewhere in its range; or it holds the branches to the nodes below it which each cover half its range. Because all keys are stored in the leaves, the key look-up time becomes proportional to $\log_m n$ which is still of the same order as for a normal tree.

The principle of inserting keys in this kind of tree is shown in Figure 4.4a. Inserting a new key starts at the root and it goes down the branches until it reaches an empty or an occupied node at the bottom of the tree. In the first case, the sub-volume represented by the empty node gets the new key assigned to it. If an occupied node is encountered, that node's range is split into equal size sub-volumes — half segment, quarter area, octant, etc. — and the current node key is moved to its sub-volume inside the split node. Next,

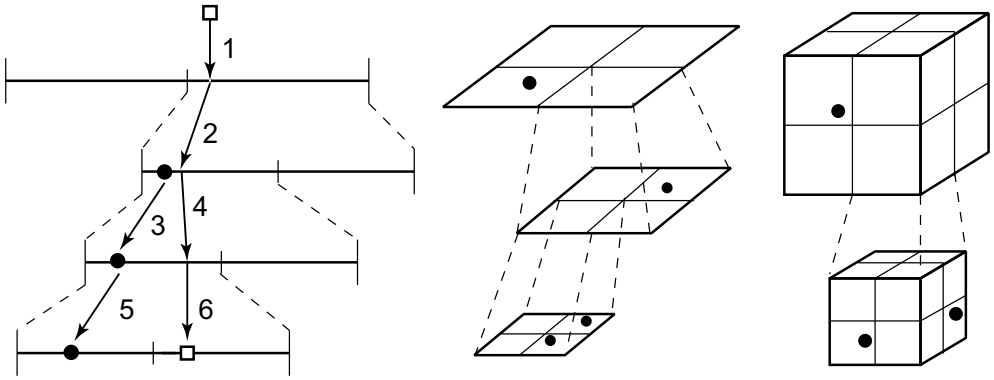


Figure 4.4: Structure of one, two, and three-dimensional fixed-range trees. The algorithm for inserting new keys into the sparse tree is illustrated for the 1-D case. When inserting a new key (open square) a decision into which sub-volume the key belongs is made at each level. If a sub-volume is a leaf and already occupied (black dots), that volume is split and both the new and the current key are moved to their own sub-volumes. This process is repeated until an empty leaf is reached. The numbers in (a) indicate the steps taken by the algorithm. Figures (b) and (c): structure of fixed-range trees for two and three dimensions, respectively.

the new key is placed in its corresponding sub-volume. This process is repeated if the two keys are close together and end up in the same sub-volume, as shown in Figure 4.4a at steps 3 and 4. An empty tree consists of just the empty top-node which controls the total range spanned by the tree.

There are two disadvantages of trees with a fixed range. First, prior knowledge of the range of keys to be inserted is required. In the type of application described in this work, this is not a problem. The maximum range of hit coordinates is known a priori and almost always limited by physical constraints, like the size of the detector. The second disadvantage is that the amount of memory needed to store a tree can become prohibitive. There are more internal nodes than keys when the tree is fully developed down to its smallest spacing between keys. A fully developed k-dimensional tree with height h (the root of the tree has $h = 1$), has $(m^h - 1)/(m - 1)$ nodes and ends in m^{h-1} leaves. However, only nodes actually used need to be created. The key-insert algorithm, described above, only creates those nodes which have occupied sub-volumes.

The algorithm to find elements within a given range (section 4.2.1) requires a binary search for the lower value of the search range. This search can now be replaced by a tree search which runs in the same time. The next step is to take ordered elements which in a 1-D tree can be done by a walk through the tree's nodes. However, no such walk exists for multi-dimensional trees and therefore the second part of the algorithm has to be adapted as well. Finding all elements within a certain range is done by traversing the tree structure down the branches. Any internal node whose range overlaps the requested range is searched recursively. Any key in the nodes traversed or in the leaves of the fixed-range tree lies in a range which overlaps with the search range. A final verification is needed to check whether that key actually lies inside the search range and, in that case, to add it to the list of found keys.

4.2.3 Hash table

A competitor of binary-search trees is a hash table [237, 244]. With proper tuning of the hash function and the number of hash bins, hash tables have constant insert and key look-up time. In k-D space the total volume can be divided in sub-volumes as a regular k-D array of bins, each bin containing the points which lie inside it. This is equivalent to a hashing algorithm with a simple linear hash function to convert key coordinates to bin numbers. For a range look-up to work with a hash table, the choice of hash function is in any case limited, because the hash-function has to preserve the order of the elements. For the key-coordinates to bin mapping applied here, that requirement is fulfilled.

Normally in hash-tables the number of bins is larger then the expected number of keys to be stored in the table, such that the average number of keys per bin is less than one. In a standard approach, multiple keys in the same bin are often stored as a linked list. In the type of application here, having more bins than points would slow down a range-search as many empty bins inside the search volume would have to be examined as well. The find-in-range algorithm for a hash container requires the selection of keys from the bins that overlap the search range. For bins completely inside the range all keys can be taken immediately. For bins overlapping the border of the range, selecting keys is a linear search, but now for a much smaller number of entries $n_{\text{bin}} \approx n/N_{\text{bins}}$, with N_{bins} the number of bins. A search using a hash table will therefore be faster than a k-D tree search if $c_{\text{linear}} \times n_{\text{bin}} < c_{\text{tree}} \times \log_m n$, where c_{tree} and c_{linear} are the time constants for a tree search and a linear search respectively. The number of bins needed for hashing to be faster than a tree is then given by

$$N_{\text{bins}} > n \cdot \frac{c_{\text{linear}}}{c_{\text{tree}} \times \log_m n} \quad . \tag{4.3}$$

In this calculation the overhead caused by many empty bins is ignored and the inequality of equation (4.3) is only an indication. Equation (4.3) grows almost linear with n. The constant c_{linear} is normally smallest when the keys to compare are stored sequentially in memory, which is not the case for a linked list. The keys in a bin can be stored sequentially in memory by ordering an array of keys by bin number. A hash-bin then points to a sub-range of the ordered array. However, inserting elements is now no longer a constant time operation, as for standard hash tables. The time taken for the sorting the keys array by bin number, amortized over n look-ups, is limited to $\lg n$. In the inequality of equation (4.3), one should also take into account that the requested search-range can span several bins due to overlap with the bin boundaries, even when the search-range is smaller than a bin-volume. Therefore, c_{linear} should be replaced by $c_{\text{hash}} = c_{\text{linear}} \times f_m$. The value of the multiplicity factor f_m can be anywhere from close to one, if the search volume is much smaller than the bin volume, to several times 3^k, if the search volume is much bigger.

In conclusion, if enough information about the input data and the search-ranges is known and the condition of equation (4.3) is fulfilled, this kind of k-D hash table can be faster than the k-D search tree. A comparison of the relative timing between the fixed-range binary-tree implementation and a hash table implementation is given in section 4.2.5. The k-D hash table is also known as a bucketting container and is used for example in many of the fast k-D Voronoi-tesselation algorithms [245].

4.2.4 Implementation

Both the normal and fixed-range tree have been implemented in C++. Because the types to store vary, the trees are designed as template classes. The classes follow the C++ Standard Template Library (STL) conventions [246,247] and are implemented as container adaptors on top of an STL `vector` class. Like the STL `map` class, the implementations differentiate between the objects to store, called elements or values, and the key to sort those objects with. Keys can have up to eight separate dimensions.[3] The classes provide two interfaces to the user to access the data. One is the standard STL-vector interface for linear access using iterators and indexing. The other accesses the data as an ordered set in k dimensions using the keys and are used to find a given key or to look up all elements in a given k-D volume. Figure 4.5 gives a unified modelling language (UML) diagram of the classes and methods.

Like the STL containers, the k-space container classes have different behaviour but (almost) identical interfaces. Which type of container to use depends on the type of application. The k-D tree class is called `map` as it behaves identical to the STL `map` class. The times taken for both the insert and find operations on this class have an upper limit of $\log_m n$. Different from the 1-D STL map, which is normally implemented using a balanced red–black tree, the worst–case timing for these operations for the k-space map is order n. The `map_fixed_range` class guarantees an insert and find time proportional to $\log_m n$, but can only be used if the range of keys is known beforehand. Hash tables are not implemented in the standard template library. A simple hash container in k-D space has been implemented. If the range and the number of keys as well as the typical volume of a search range is known beforehand, then the hash container class can be faster than the map classes as explained in section 4.2.3.

In the STL ordered-container classes, the ordering operator is given as a template parameter. For the multi-dimensional containers, this ordering operator is replaced by a key-traits class. The methods of this class are used for all key operations. A default key-traits implementation is provided that works for simple key classes (identical coordinate types accessible via index operator). The k-space containers have been optimized for speed. This optimization implies that there is no checking of the input parameters or key values.

Map containers

The map classes have four template parameters: the type of the elements to store, represented by class `value_t`; the type to sort on, represented by `key_t`; the dimension of the space (which gives the number of used coordinates in `key_t`); and a key-traits class which lets the map compare and modify key objects, represented by `key_traits_t`. All operations the map performs on key objects are handled by static methods in `key_traits_t`. The map classes therefore require no knowledge of the coordinates of `key_t`. The only requirement on the `key_t` class is that its objects can be constructed with the copy-constructor (using C++ placement-`new`); no default constructor or assignment operator for `key_t` is required. The only requirement for the `value_t` class is that it must be storable in an STL `vector`.

[3] The maximum dimension of keys can easily be extended to more than 8.

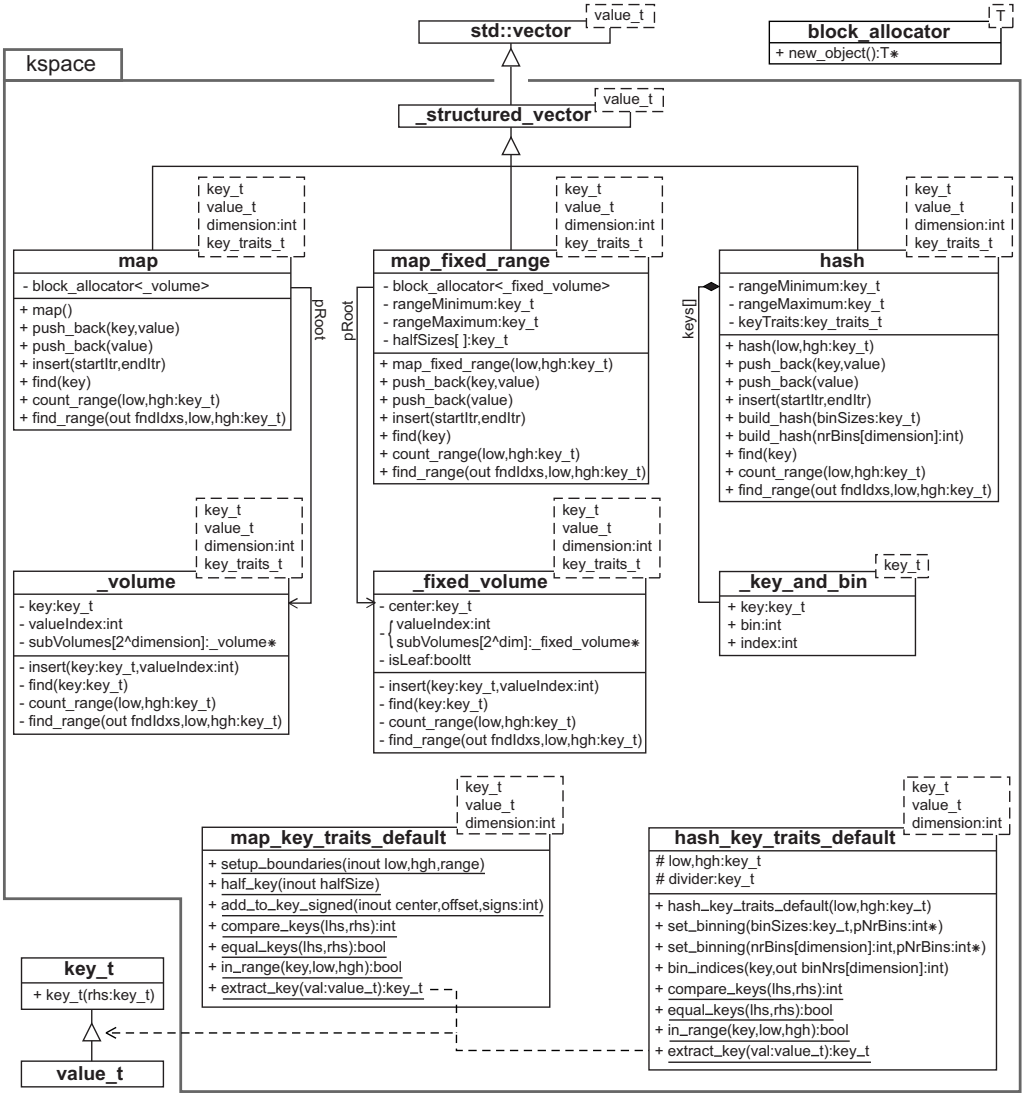


Figure 4.5: UML diagram of the *k*-space container classes and the node and helper classes. The `value_t` class represents the elements stored in the container, the `key_t` class represents the multi-dimensional points which are the keys on which the ordering is based.

The `_structured_vector` base class inherits all methods from the STL vector class from which it derives. However, values can only be inserted from one of the derived *k*-space container classes. Elements are inserted in the containers using the `insert` and `push_back` methods. They add a copy of the value to the underlying vector and update the tree's node structure for the associated key by calling the `insert` method on the root node. In general all methods using the underlying tree structure forward the call to the root node where the actual recursion over the tree's nodes is done. Often, the key is a sub-class (as indicated in Figure 4.5) or member of the `value_t` class. In this case, a

set of values can be inserted in the maps in one operation by extracting the key of each value using the `extract_key` method of `key_traits_t`. The `push_back` method follows the algorithm described in section 4.2.2 with one additional step. For the standard tree structure, at every node where the comparison of the new key and the node's key yields zero (equivalent to the new key is **not** less-than), the two keys are checked for equality. For the fixed-range tree, this equality check is done once and only if the key descends to an occupied leaf. If the keys are equal, the new element is either discarded or an exception is raised, depending on how the container object was configured.

The `find` method can be used to retrieve the value associated with the specified key. The `find_range` method retrieves all elements for which the keys lie within a volume specified by the inclusive lower boundary and the exclusive upper boundary. It returns an array with the indices of the elements within the search range. These indices can then be used to retrieve the values through the vector interface of the map.

Map node and leaf class

The private classes `_volume` and `_fixed_volume` represent both the nodes and leaves of the tree. The maps allocate blocks of these objects, using the `block_allocator` helper object. In the standard map, `_volume` contains the key, an index to the associated value in the map's underlying vector, and a set of 2^k child-node pointers stored in the `subVolumes[]` array. The `subVolumes[]` array is indexed using a bit-coded comparison of key coordinates. For the `map_fixed_range` class, the node and leaf are objects of class `_fixed_volume`. An internal node just holds a set of child-node pointers and the data member `center` contains the key value for the center of the volume spanned by the node. In a leaf node, indicated by the `isLeaf` data member, the `center` member is the key associated with the value and the corresponding index is stored in place of the child pointers. All methods in `_volume` and `_fixed_volume` call themselves recursively on all existing sub-volumes that contain part of the requested search key or range. For efficiency reasons, the `insert` method is actually implemented as a loop.

Key traits

The `key_traits_t` template argument contains a set of static methods required for the key operations of the map implementations. The `map` class uses only the key comparison methods: `compare_keys`, `equal_keys` and `in_range`. The `compare_keys` method returns an integer with the results of individual coordinate comparisons shifted into the corresponding return bits. The methods `equal_keys` and `in_range`, on the other hand, return a boolean which is the logical AND of all coordinate compares.

For the `map_fixed_range` class, one needs to calculate ranges and centers of the keys. The range given to a fixed-range map's constructor is first passed through the key-traits `setup_boundaries` method. This makes it possible for a traits implementation to, for example, adjust integer-type boundaries to be a power of two or to set string boundaries to the first character of the strings. The map calculates centers of nodes by adding or subtracting half the parent-node's range from each coordinate. The half-ranges for each level are calculated using the key-traits `half_key` method and are cached by the map. The center key for a sub-volume is obtained by adding the half-range of the current level to the parent's center correcting the sign for each coordinate. The signs are defined by the corresponding bit in the value returned by the `compare_keys` method.

A default implementation for the `key_traits_t` template member is provided by the `map_key_traits_default` class. This class is also set as the default template argument. The default implementation will work for `key_t` classes with simple numerical coordinates. It requires access to the `key_t` coordinates using the index `operator[]`. Most of its method are implemented as template-meta-programs that iterate the operation over all used key coordinates [248–250]. The default implementation of `extract_key` is just a C++ `static_cast` of `value_t` to `key_t` which works if `value_t` is indeed derived from `key_t`. This method is only used when a key and its associated value are combined in a single object as in the `push_back(value_t)` and `insert` methods of the maps.

If the coordinates in the key class are not accessible, are not numeric (e.g. strings), or require more complicated operations for the calculations of the centers in a fixed-range map, the user must provide a specific implementation for the `key_traits_t` class. One can either derive from the default traits class and override the methods that need to be changed or implement all methods in the specific traits class.

Hash container

The `hash` class has the same template parameters as the map classes, but its default key-traits class is different. The `key_traits_t` class takes the role of the hash function in standard 1-D hash tables and is responsible for the mapping of keys to bin numbers. The `bin_indices` method in the `key_traits_t` class is the actual hash function that maps `key_t` coordinates to a k-D array of bin numbers. In order to use the `find_range` method, the hash function should preserve the order of the keys which requires an ordering of the key's coordinates. If only the `find` method is used, no restrictions are imposed on the hash function.The default implementation of the hash key-traits is `hash_key_traits_default`. The `bin_indices` method in this default divides the range of each key's coordinate in equal sized bins.

The `hash` class requires a slightly different setup before being used, because, in this implementation, the hash structure can only be built once the number of bins to use is specified. For this, the `build_hash` methods are used. These methods assign a bin number to all inserted keys using the `bin_indices` method of the `key_traits_t` class. The key and the bin number, associated with an element, are stored in an array of the helper class `_key_and_bin`. The k-D bin numbers are collapsed into a single value $b = \overline{b} \cdot \overline{d}$, where $\overline{b}$ is the k-D vector with bin numbers given by the hash function to each coordinate and $d_j = \Pi_{i=1}^{j} n_{i-1}$ with n_i the number of bins for each dimension and $n_0 = 1$. The array of `_key_and_bin` objects can then be sorted according to the bin value, such that all keys in the same hash bin are stored sequentially in memory. The `find_range` method in class `hash` uses a recursive template-meta program to loop over the k-D range of bins that overlap the search volume and in each bin performs a linear search of the keys to select the ones inside its search range.

4.2.5 Timing performance

Because the `map_fixed_range` class requires no tuning and has an guaranteed time behaviour, it is used as the default range look-up container for the tracking algorithm. In this section, its range look-up time is compared to a very simplistic linear-search algorithm. This simplistic algorithm is still useful, because the timing for an optimized hash-table can be calculated from the linear search time.

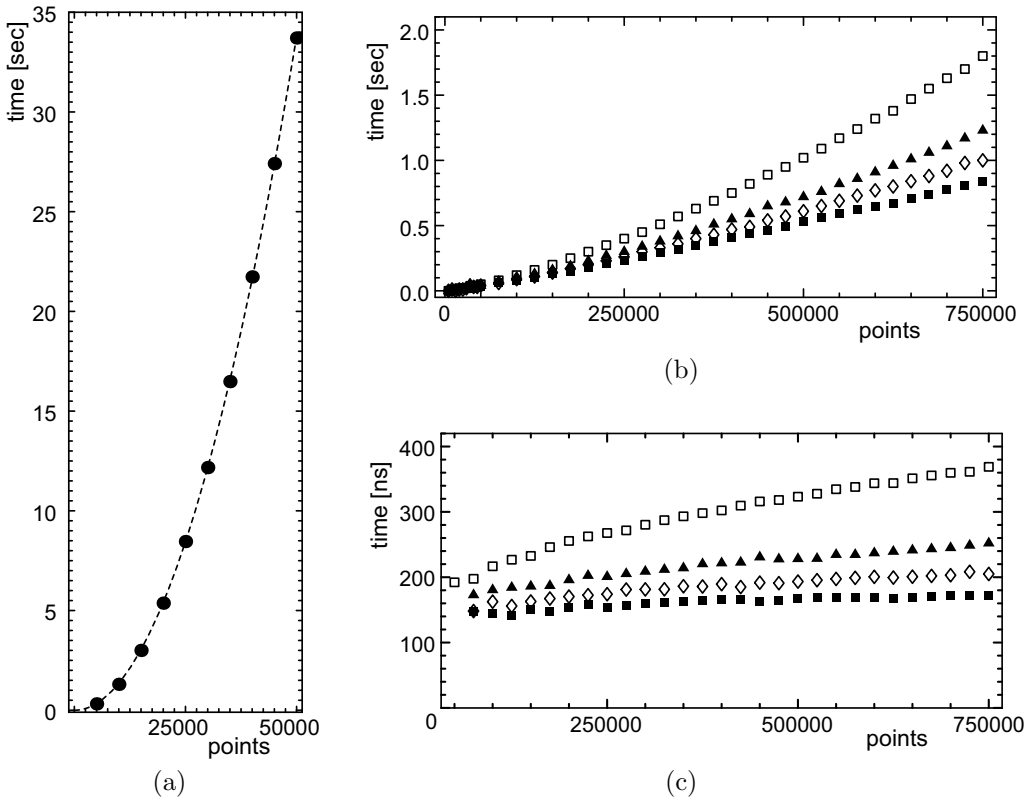


Figure 4.6: Time needed to find neighbours for all points versus the number of points: (a) using a simple linear search; the dashed curve indicates the expected n^2 time behaviour; (b) using a 3-D fixed-range map. The vertical line indicates the point range spanned by figure (a). Results in figure (c) are obtained from (b) after dividing the data by the expected $n \cdot \log_8 n$ behaviour. The dependence on the search volume is indicated by the different symbols. The length of the sides of the cubic search volume is: ■ $= 1$, ◊ $= 2^4$, ▲ $= 2^5$, and □ $= 2^6$. corresponds to a cube with sides of 2^{14}. For (a), the result is independent of the search volume.

The time it takes to traverse the fixed-range tree in a search of the map is proportional to $\log_m n$. In the link-building step of the tracking algorithm (see section 4.3.1), all points close to a given point need to be found. This operation is done for each point, which leads to an additional factor n (the number of points) in the total time needed for the link-building step. Figure 4.6b shows the timing results for a 3-D fixed_range map containing randomly distributed points on a 1.8 GHz CPU.[4] In Figure 4.6a, the time needed when using a straightforward linear search algorithm is given. A fit to the data (dashed line) with $t = c_{\text{linear}} \cdot n^2$ yields $c_{\text{linear}} = 13.5\,\text{ns}$. The search time for the map depends on the size of the search volume, indicated by different symbols. This can also be seen in Figure 4.6c which shows the same search time after dividing the data by the theoretical n-dependence of $n \cdot \log_8 n$. The tree's search-time coefficient c_{tree} depends slightly on n because a larger search volume (or equivalently a higher point density) leads to more

[4] all values in this paper are for a 32 bits AMD Athlon 2500+ CPU at 1.8 GHz.

points inside the search volume. In that case, the probability increases that the search volume spans multiple nodes in the tree, which all have to be searched. From Figure 4.6c one can determine that a single search in the tree scales as $t = c_{\text{tree}}(v) \cdot \log_8 n$, where c_{tree} depends slightly on the size of the search volume v and is about 170 ns for the smallest search volume.

The one-time overhead to build the underlying tree structure of the map should also be considered. To create the tree, each point in the map has to be inserted in a tree containing the already inserted points. For each insert, the tree has to be traversed to find the place where to insert the new point. The total time needed for n points is thus proportional to $\sum_{i=1}^{n} \log_m i \propto \ln(n!)$. In Figure 4.7, the tree building time is shown as a function of the number of points for a 3-D fixed-range map. A fit yields a building time of $t_{\text{build}} = 46.5 \ln(n!)$ ns. As $\ln(n!)$ is difficult to calculate for large n, an approximation using an upper limit for $\ln(n!)$ is $t_{\text{build}} = -26.6 + 46.9 \cdot 10^{-6}\, n \ln(n)$ ms, valid for the range of points in Figure 4.7.

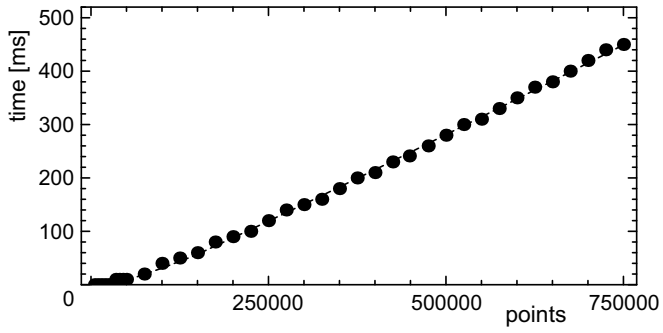


Figure 4.7: Building time for a fixed-range map as function of the number of 3-D points to insert. The dashed line is a fit to the data yielding $t_{\text{build}} = 46.5 \ln(n!)$ ns.

From the results found above, one can calculate the required number of bins for a hashing container using equation (4.3). For typical volumes and search-ranges involved in the track finding for emulsion data of the CHORUS experiment, the search-range bin multiplicity $f_m \approx 3^3 \times 0.6$. Substituting the values found above for $n = 75000$ gives $n_{\text{bin}} \approx 4$ and $N_{\text{bins}} \approx 17900$. Using 2^{15} hash bins, the k-space hash container is a factor 1.21 (for search box size = 1) to 1.55 (box size = 64) faster than the fixed-range map used by default. The dependence on the size of the search range shows that tuning of the number of bins is important.

4.3 Track-finding algorithm

As described in the introduction, the track-finding algorithm has been designed for an environment with a large number of hits and for tracks which can only be considered as straight lines on the scale of a few hits. The hits of a track show therefore only local correlations. The large number of background hits rules out tracking algorithms that do complex calculations or examine a large fraction of all possible combinations of hits. In the following, a collection of hits that are part of a possible track is referred to as a segment. A fully-grown segment is called a track candidate. In the implementation of the algorithm, a custom-made class is required to make all decisions about accepting hits in a segment. This class, containing all the hit and track acceptance criteria, is known as the criterion.

4.3.1 Concept

Linked hits network

The algorithm first builds up the network of links. A criterion defines a cuboid volume, relative to the position of a hit, to be searched for other hits. The k-D map containing all hits is used to find the neighbouring hits in this volume for every hit. The map's `find_range` method dictates the use of a cuboid region. In an application, however, the link acceptance region for track hits is not necessarily rectangular. For example, the hit acceptance region based on the extension of a segment with constant uncertainties for both position and direction has the shape of a topped-cone. The algorithm therefore applies a criterion to select acceptable links from all the links formed by the base hit and the hits found in the acceptance volume around it. The implementation of this criterion allows the user to define any arbitrary acceptance volume. The track-finding algorithm is in general isotropic, but can be restricted according to the experimental conditions. Any restrictions that are applied when building the links network also limit the tracks that can be found. For example, an angular restriction in the link-acceptance region limits the solid angle of the tracks that can be found.

The hits and links correspond to the vertices and edges of a large graph. If only forward links are accepted, this connection graph is a directed acyclic graph. The connection graph links each hit to the other hits which may belong to the same track. As a result, the look-up of all possible hits that might be added to a segment is very efficient.

Segment growing

The graph of linked hits, built in the previous step, is searched with a modified depth-first algorithm [237] for paths compatible with tracks. All hits are taken as possible starting points for segments. A criterion is applied to select the hits that should be used as starting points. All links attached to a selected starting hit form the initial set of segments containing just two hits. Each segment is then expanded recursively by adding hits linked from the last hit in the segment. For this, a criterion decides which new hits are accepted. In an application, this criterion should accept hits that correspond to a topology consistent with its particular track model. The growing of a segment stops when none of the links from the last hit are accepted. The segment then becomes a track candidate. A segment splits into multiple new segments whenever there is more than one accepted hit. Each new segment is also followed until it stops. To do this, the algorithm backtracks to previous hits that have multiple accepted links and then follows these. The algorithm behaves therefore as a depth-first graph search, except that stepping to already visited vertices (hits) is not disabled.

Each track candidate, formed this way for a selected starting hit, traces a path through the links network. All the track candidates share the same starting hit, indicated by **S** in Figure 4.8. No hit is ever exclusive to a single segment or track candidate. In fact, many track candidates are identical, apart from a single hit. In Figure 4.8, the growing procedure can be imagined as moving from left to right through the links network, creating track candidates for every path that is compatible with a given track model.

The algorithm selects the best track candidate from all track candidates for each starting point. The result of the comparison between track candidates is decided by another criterion. Comparisons are only meaningful if similar entities are compared, like track candidates or segments spanning the same range of hits. Because the algorithm

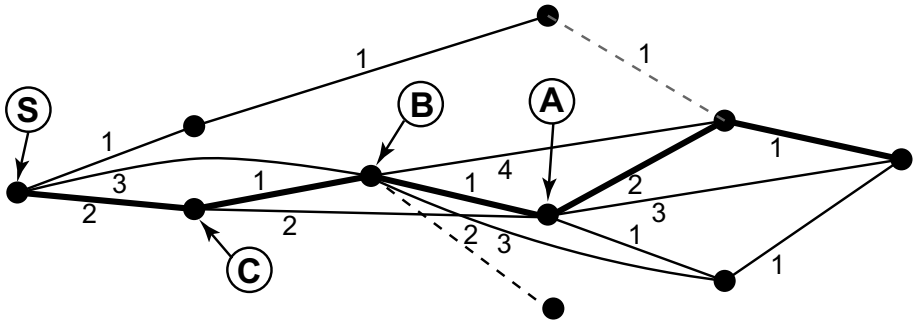


Figure 4.8: Topological representation of the recursive segment growing tree. The black dots represent the hits. The links between hits are indicated by the numbered lines. The numbering restarts at each hit. The starting point of the track search is indicated by **S**. The thick line is the final track candidate. Dashed links indicate links that are rejected by the hit-acceptance criterion using the preceding segment. Decisions which branch to retain have to be taken in points **A**, **B**, **C**, and **S**.

behaves as a depth-first search, comparisons are only made between track candidates which are complete segments. In practice, selecting the best track candidate is done on the fly whenever there are several accepted links for a segment. A first track candidate is created by following either the first or the best accepted link at every step. Backtracking along these hits, that track candidate can be compared with others following the other branches. At each hit that has more than one accepted link, the other accepted links are grown as well. The track candidate with the current best branch can be compared with a track candidate taking a new branch. Only the best of the two is kept at every step. Effectively, this amounts to a branch decision at each hit. In Figure 4.8, this branch-pruning procedure can be imagined as keeping the track candidate with the best right-hand side from the decision point and moving right-to-left back to the starting point **S**. The example in Figure 4.8 corresponds to the case in which the track candidate with the most hits is considered the best track. If two (or more) track candidates contain the same number of hits, the one that corresponds best to a straight line is chosen.

For every starting point after segment growing, there is therefore one remaining track candidate. A final criterion is used to pass a judgement about its validity as a possible track. If accepted, the track candidate is stored in a list of found tracks. After processing all starting points, one is left with a list of possibly overlapping tracks. The track candidates from different starting points can share hits. A criterion is used to decide which of the overlapping track candidates to keep.

Limiting combinatorics

The link-following algorithm, as described above, considers all possible combinations of linked hits in the network and therefore always finds the best track candidate. A determination of the tracking time (section 4.3.3) shows that the algorithm scales approximately as $\sum_k^{k_{\max}} (c_l\, n)^k$, where $k_{\max}$ is determined by the typical segment length and the volume fraction $c_l = v_l/V$ is the size of the link acceptance region v_l divided by the total volume V. The product $c_l\, n$ corresponds to the expectation value for the number of links per hit.

As long as this value is reasonably low for track-unrelated hits, the tracking time remains polynomial in the number of hits n. Unfortunately, the link-following algorithm can suffer from an inverse combinatorics problem. On a track, the number of possible segments could be very large, such that the tracking time becomes too long for all practical purposes. The problem is that the acceptance criteria for links and hits have to accomodate for hit-finding inefficiency. In the CHORUS experiment, the hit finding efficiency in each emulsion image is about 86 %. The segment-growing algorithm must therefore be able to cross one or more layers with no hit on a track. The link-acceptance region must thus span several layers. In that case, hits belonging to the same track can have both direct and indirect links pointing to them. A hit that can be reached following two or more short links, can also be reached by a single link. Links numbered 1 and 2 which connect hit **C** to **A** in Figure 4.8 are an example. Due to these longer links, the basic algorithm follows the same set of links at the tail of a track very often. The segments created in these steps are usually identical apart for one hit. In case a hit on a track can be reached either directly or via one intermediate hit (the link acceptance region spans two layers) and assuming that a track has hits on all layers, a hit i is visited from its predecessor and from the hit before that. In this case, the number of visits $HV(i)$ to hit i on a track is given by:

$$HV_2(i) = HV_2(i-1) + HV_2(i-2) \quad . \tag{4.4}$$

The subscript 2 indicates that the next two layers can be reached via links. The first two hits are only visited once, independently of the number of layers links can span, so that

$$HV(1) = HV(2) = 1 \quad . \tag{4.5}$$

Equations (4.4) and (4.5) correspond to the definition of the Fibonacci series. The total number of hits visited, $THV(N)$, for a track with N hits is then given by:

$$THV_2(N) = \sum_{i=1}^{N} HV_2(i) = HV_2(N+2) - 1 \quad . \tag{4.6}$$

This sum can become quickly large. For a track with $N = 25$ hits, the total combinatorics of the algorithm is $THV_2(25) = 196,417$. The result will be even higher when links spanning two or more intermediate hits also exist. The sum in equation (4.4) then gets extended with more previous terms (known as the tribonacci and tetranacci series [251]). For links connecting the next four layers $THV_4(25) = 3,919,944$.

In a standard depth-first search of a graph [237], the above problem of large combinatorics doesn't occur because vertices (hits) are marked as visited. If a search path reaches a visited vertex it stops. The same strategy cannot be applied in a segment-growing procedure because the first visit to a hit does not necessary correspond to the best path. A similar strategy can be used, however, if the correct path can be identified. One can then mark the hits on that path and disable future visits. One case in which the correct path is known is when a complete path is found and taken as a track candidate, but then it is too late as all the combinations have already been tried. However, because of the recursive nature of the segment-growing algorithm, this solution can be applied by marking the hits at the tail of a segment when the number of hits in the tail exceeds a given value t. The value t is chosen such that a tail containing more than t hits is probably the correct tail of the segment. The tail mark is only valid for a single starting point,

so it does not affect segments grown from other starting points. With this tail marking, the total number of hits visited is now limited to $THV_2'(N,t) = N - t + THV_2(t)$. In the example above with $N = 25$ and using $t = 7$ one obtains $THV_2'(25,7) = 51$. With a link acceptance region spanning 4 layers one finds $THV_4'(25,7) = 78$.

Unfortunately this marking strategy creates a dependence on the order in which the links are followed. The set of hits which will be included in a track is now affected by the link order. All possible paths at the tail of some segment are examined, but when the algorithm backtracks to earlier hits, continuations of other paths are blocked by the earlier segment that included sufficient hits. The optimization of the beginning of the segments is thus effectively disabled. To control which hits are included from the start of a segment, the links have to be followed in a certain order. There are two solutions to this problem. The first solution is called initial link ordering. When building the links network, the accepted links are sorted according to a value defined by a criterion. This value determines the order in which links are followed during segment growing. For example, it can favour short links such that the first segment built contains the largest number of hits. The second solution is called followed link ordering. In this solution, the segments are sorted each time a hit is added according to how good the new hit fits in the segment. If there is more than one accepted link, the best new segment is grown first. This solution yields better results as the most likely segment continuation is tried first. However, the first solution sorts the links only once and should therefore be faster. A simulation showed that the time gained in following the best link first compensates for the time spent in sorting the accepted links (section 4.4).

The tail marking solves the combinatorial problem in the link-following algorithm. A related problem exists with the starting points. Each hit is tried as starting point for a segment and therefore the same track is found again for all hits on a track. To avoid this, the hits in an already found and accepted track candidate can be marked. Hits marked in this way can then be skipped. Again, the restriction creates an ordering dependence, now on the order in which starting points are processed. Therefore, a criterion is used to sort the hits a priori to determine the starting order.

4.3.2 Implementation

The implementation is organised as a source-code libary containing the tracking toolkit. The user must provide the hit, track, and criterion classes for a specific application. Specific optimizations for each environment are dealt with by the corresponding criterion implementation. The user must also put code in its application program that feeds the hits to the track-finding procedure and converts the found track candidates to tracks. A simple Monte-Carlo simulation program, used to assess the efficiency of the tracking algorithm, can act as a template for such a main program. This program is described in section 4.4.

Two header files specify which input hit class, coordinate class, and criterion class to use. The criterion type specification has as default the abstract base class `VCriterion`. By replacing this default with a concrete class, the overhead of calling virtual functions can be eliminated. Using he abstract `VCriterion` instead, a criterion can be selected at run-time. A test showed that the time overhead for calling abstract methods is less than one percent (see Figure 4.11b). Another header file defines a set of compile-time flags. These flags specify which of the link ordering options, described before, to use. A class diagram of the implementation is shown in Figure 4.9.

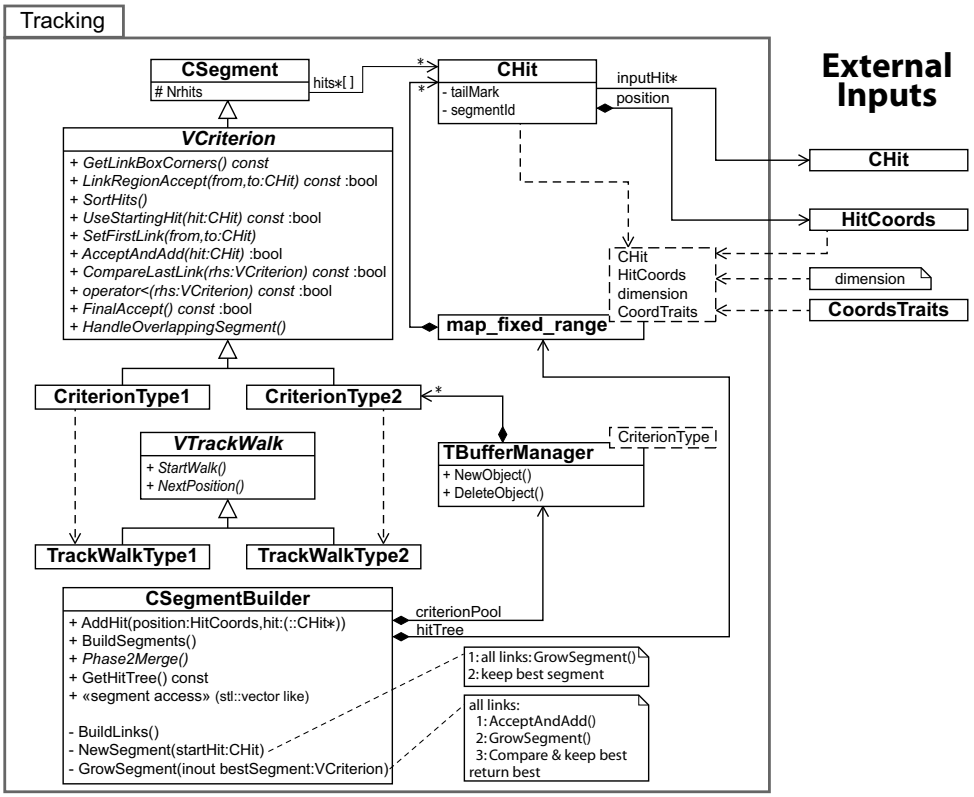


Figure 4.9: Class diagram (UML) of the classes involved in the tracking. All classes are in the 'Tracking' namespace. At compile time the user must provide the dimension of the hits, the input hit class (`CHit`), the coordinates class (`HitCoords`), and the traits for working with objects of this last class (`CoordTraits`). The last two classes correspond to the template parameters `key_t` and `key_traits_t`, respectively, of the *k*-D fixed-range map (section 4.2.4). The `HitCoords` class corresponds to the position of a hit and it is a data member of the internal hit class (`Tracking::Chit`). An array of these internal hits forms a segment and is the type of object included in a criterion object to make acceptance decisions. The user must supply concrete implementations of the criterion class, indicated by `CriterionType1` and `CriterionType2` and their corresponding track-walk implementations `TrackWalkType1` and `TrackWalkType2`.

The tracking classes use an internal hit class which points to a user-defined hit object. The user-defined hits are not directly used by the tracking code; the pointer can be used in the main program to locate hits assigned to tracks. The input to the tracking code consists only of the hit coordinates represented by class `HitCoords`. In the tracking code, only the map's `key_traits_t` operates on this type of objects which is represented by the `CoordsTraits` class (see section 4.2.4). The coordinate class can be replaced, at compile time, by any coordinate class chosen by the criterion implementer, as long as a corresponding `CoordsTraits` and the dimension is defined as well. For most simple coordinate classes, the map's default traits class can probably be used.

The class `VCriterion` encapsulates the segment class `CSegment`, which holds a fixed-size array of pointers to hits to avoid time-consuming heap allocations. Removing an automatic heap allocation in the inner segment-growing code in a previous version, reduced the tracking time by about a factor three. The assignment operator in `CSegment` is optimized to only copy the used part of the array. All criteria, mentioned in section 4.3.1, appear as abstract methods in `VCriterion`.

The actual segment building and bookkeeping of track candidates is performed by an object of class `CSegmentBuilder`. The structure of its recursive `GrowSegment` method is shown in pseudo-code in Figure 4.10. The actual code contains several optimizations to avoid loop overhead, to undo tail marking and to reuse discarded segment objects. The method starts with a pointer to the current segment. The procedure will replace this pointer with the new segment it has built from the given one.

The lines 6-12 in Figure 4.10 iterate over all links of the last hit in the current segment. The criterion is asked to add the destination hit of each link to a copy of the input segment in line 10. If this hit is accepted, the extended segment is stored in a buffer in line 11. The followed-link-ordering solution to the tail recursion problem (section 4.3.1) changes the type of this buffer from a FIFO to a priority queue. The priority queu is implemented using a binary heap. If no hit was accepted, the segment has reached its end and the method returns in line 14. If there are accepted links, the current segment is replaced with the top of the buffer contained stored segments in lines 18 and 19. If there is more than one accepted link, the first segment is recursively grown in line 22. If the tail from the current hit to the end of this now complete segment contains more hits than the tail-mark limit, the tail is marked. The iteration in lines 25-33 pops the other stored segments one-by-one from the buffer and grows each one using a recursive call to `GrowSegment`. If one of these fully-grown segments is better than the current one, they are swapped in lines 29 to 32, which also update any tail marking. The worse segment is immediately discarded at line 33. At the end of the iteration, any tail mark is removed in line 35 before the method returns. The infinite loop in line 3 and the if-statement in line 21 handle the case that only one link is accepted. In that case a recursive call can be avoided and the current segment is simply replaced by the one accepted and the loop repeats.

The `GrowSegment` method creates and discards many criterion objects. Using the C++ `new` and `delete` calls for individually allocating these objects is not efficient. Instead, the template class `TBufferManager` is used which allocates a pool of objects. Individual objects can quickly be taken from and returned to this pool. The pool works as a LIFO which has the additional advantage of improving data cache efficiency.

Found tracks are represented by criterion objects that pass the criterion's method `FinalAccept`. They are kept by the segment building class. In the end, the list of criterion objects retained are the tracks found by the procedure. The main program, which knows the exact type of the criterion object, can access this list and process the tracks further. Standard operations would include fitting a track model through the hits and adding possible missed hits close to the fitted track. For this, the map containing all hits can be used and it is therefore accessible via `CSegmentBuilder`. An implementation of such code is available as the `Phase2Merge` method in `CSegmentBuilder`. This method uses the abstract interface class `VTrackWalk` to walk along the track contained in a criterion object. The criterions instantiate the corresponding concrete implementation of the track-walk interface. The `Phase2Merge` implementation looks for hits within a certain

```
1  void GrowSegment(VCriterion*& pCurSegment)
2    VCriterion* pNewSegment = 0
3    loop: // Loop if only one link is accepted ⇒ no recursive call needed!
4      // — STORE NEW CRITIONS FOR ACCEPTED LINKS —
5      AcceptedLinkBuffer acceptedLinks // Type is link-ordering dependent
6      iterate: pCurSegment→lastHit→links ⇒
7        if: pNewSegment == 0 ⇒
8          pNewSegment = criterionPool.NewObject()
9          *pNewSegment = *pCurSegment
10       if: pNewSegment→AcceptAndAdd(link→toHit) ⇒
11         acceptedLinks.push(pNewSegment)
12         pNewSegment = 0
13     nrAccepted = acceptedLinks.size()
14     if: nrAccepted == 0 ⇒ done. // NO LINK ACCEPTED ⇒ DONE
15
16     // — START ITERATION ACCEPTED LINKS WITH FIRST ONE —
17     currentLength = pCurSegment→nrHits // to calculate tail length new segments
18     criterionPool.DeleteObject(pCurSegment)
19     pCurSegment = acceptedLinks.pop()
20     // — MORE ACCEPTED LINKS ⇒ LOOP AND KEEP ONLY BEST —
21     if: nrAccepted > 1 ⇒
22       GrowSegment(pCurSegment) // RECURSIVE CALL
23       if: pCurSegment→nrHits - curLength > tailMarkLength ⇒
24         MarkSegmentTail(pCurSegment)
25       while: acceptedLinks.not_empty() ⇒
26         pNewSegment = acceptedLinks.pop()
27         GrowSegment(pNewSegment) // RECURSIVE CALL
28         if: *pNewSegment > *pCurSegment ⇒
29           UndoSegmentTailMark(pCurSegment)
30           swap(pNewSegment, pCurSegment)
31           if: pCurSegment→nrHits - curLength > tailMarkLength ⇒
32             MarkSegmentTail(pCurSegment)
33         criterionPool.DeleteObject(pNewSegment) // Delete discarded segment
34       // — UNDO TAIL MARK BEFORE RETURNING —
35       UndoSegmentTailMark(pCurSegment)
36       done.
37     else: ⇒
38       // continue grow segment infinite loop for 1 accepted link
```

Figure 4.10: Pseudo-code of the recursive segment growing code. The indentation marks blocks of code to be executed in a loop or if-statement.

distance of the (extension of) path of the track candidate. These hits are then added to the track candidate. The `Phase2Merge` method also checks if several track candidates share the same hits. The implementation of the criterion's `HandleOverlappingSegments` method decides if these track candidates should be merged. Running the merge step after looking for hits in the extensions of track candidates reduces the number of split tracks. Tracks can split into several found track candidates due to gaps of several missing hits on a track path.

4.3.3 Tracking time

In this section a determination of the running time of the tracking algorithm is given. The time needed to do track finding depends of course on the complexity of the criterion implementation and on the tracking environment. In this section, the cylindrical acceptance criterion implementation, described in section 4.4.2, is used. The results have been obtained using the Monte-Carlo simulation program described in section 4.4. The simulation includes noise and realistic track hit distributions. The setup generates 80 tracks in a volume of 512 μm $\times$ 512 μm $\times$ 100 μm with 25 evenly-spaced z-layers. The default tracking and criterion parameters include a link-span maximum of three layers, a tail-mark limit of $t = 7$, and both link-ordering options enabled (section 4.3.1). When followed link ordering is enabled, the initial link ordering option is used to determine the order in which seeds are created from each starting hit. The segment builder was configured to call the cylindrical-acceptance criterion's methods directly. Naturally, the results are processor dependent.

Coordinate type

In Figure 4.11, the tracking time versus the number of hits is given as function of the coordinate type. It can be seen from Figure 4.11a that building the fixed-range map and the links network is slightly faster when using a 16 bits integer as the coordinate type. This is mainly due to the four times larger size of a 64 bits floating-point number. The segment growing time is shown in Figure 4.11b. The segment growing is fastest when using floating point types for the coordinates. This is probably due to scheduling of instructions in the floating-point unit, which leaves the processor free to execute other instructions.

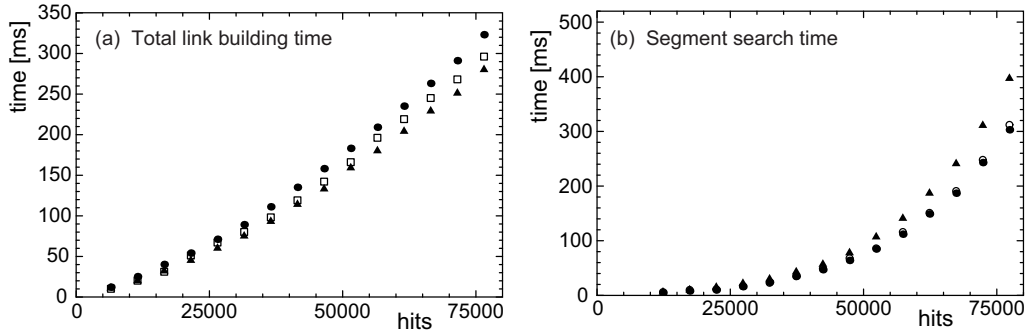


Figure 4.11: Tracking time as a function of the number of hits for different types of hit coordinates. Figure (a) gives the total time needed to sort the hits, build the fixed-range map, and build the links network. Figure (b) shows the segment-search time. The basic coordinate type for the different symbols is: • = double (64 bits); ○= double and virtual criterion methods; □ = float (32 bits); and ▲ = integer (16 bits).

Link ordering and tail marking length

As described in section 4.3.1, ordering the links helps in finding the best segments and is necessary when tail marking is enabled. Figure 4.12 shows how the tracking time depends on the tail recursion limiting parameter and the link ordering options. These parameters influence also the efficiency and one has therefore to optimize. The optimization depends on the particular environment where the tracking is applied. A simple test showed that using a priority queue to sort the accepted links takes on average about 160 ns instead of 60 ns for a non-sorting ring buffer. This time is more than compensated for by finding better segments first, as can been seen from Figure 4.12. The time dependence can be fitted with a constant term and a function of the tail-marking length t. A good fit is obtained with the function $c_0 + c_1 t + c_2 F(t)$, where $F(t)$ is the t^{th} Fibonacci number.

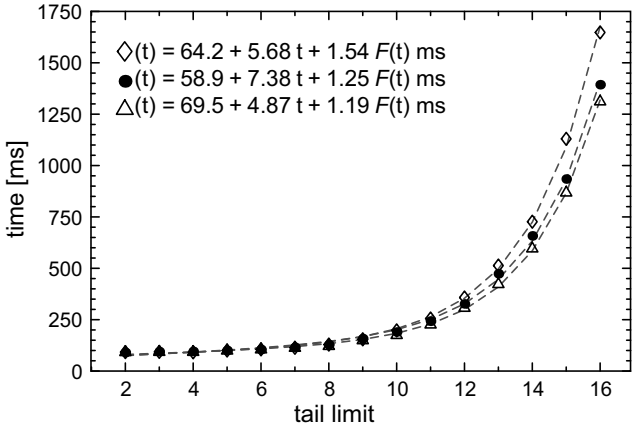


Figure 4.12: Segment search time for 50,000 hits as function of the recursion-limiting tail–marking parameter. Upper curve, ◇ symbol: links are ordered after being accepted for segment extension. Middle curve, ● symbol, links are sorted at build time. Lower curve, △ symbol: links are sorted at build time and after being accepted for segment extension. Note that each point in this figure has a different tracking efficiency. The dashed lines and legend in the figure are fits (see text) as function of the tail-marking parameter t.

Final segment length

In the following, a derivation of the running time of the segment-growing algorithm is presented. Reasonable estimates can be made in the case of uniformly distributed background hits and a criterion which uses fixed-size hit-acceptance volumes v_ℓ. In this case the expected number of accepted links per hit can be derived. The probability that a hit falls in the link acceptance volume v_ℓ is then given by v_ℓ/V, with V the total volume. The probability $P(k)$ that a volume v_ℓ contains k hits if n hits are distributed in V is given by the binomial distribution:

$$P(k) = \binom{n}{k} \left(\frac{v_\ell}{V}\right)^k \left(1 - \frac{v_\ell}{V}\right)^{n-k} \quad . \tag{4.7}$$

The last factor in equation (4.7) can be written as $e^{n \cdot \ln(1 - v_\ell/V)}$. For $v_\ell/V \ll 1$, the logarithm can be approximated to first order as $-v_\ell/V$. With the definition of the

volume fraction $c_\ell = -v_\ell/V$, the last factor in equation (4.7) is approximately equal to $e^{-c_\ell\, n}$. As n/V is just the density ρ, $c_\ell\, n$ is just the average number of hits in the volume v_ℓ. Using some approximations valid for large n, equation (4.7) turns into the Poisson distribution with mean $c_\ell\, n$:

$$\begin{array}{rrcl} & P(k; c_\ell\, n) & = & (c_\ell\, n)^k\, e^{-c_\ell\, n}/k! \\ \Rightarrow & P(0) & = & e^{-c_\ell\, n} \\ & P(>0) & = & 1 - e^{-c_\ell\, n} \\ & < k > & = & c_\ell\, n \quad . \end{array} \tag{4.8}$$

The parameters c_ℓ and n in the definition of P are dropped in the following equations unless they are different.

The track-finding algorithm uses all hits which have links as starting points for growing segments. If the criterion rejects a fixed fraction $1 - c_0$ as starting points then the number of starting points n_0 is given, according to equation (4.8), by

$$n_0 = c_0\, n\, (1 - e^{-c_\ell\, n}) \quad . \tag{4.9}$$

The procedure yields one track candidate for each starting point by keeping the best of all segments grown from that starting point. The number of track candidates of a certain length $s_k(n)$ as function of n is shown in Figure 4.13. The sum over all k for some fixed n satisfies equation (4.9) with $c_0 = 0.72$ which is the expected value for this setup where 7 out of 25 layers not used. For increasing k, the curve $s_k(n)$ peaks at higher n as the probability to get k hits in a segment first increases with n before the probability to include the $(k+1)^{\text{th}}$ hit becomes dominant.

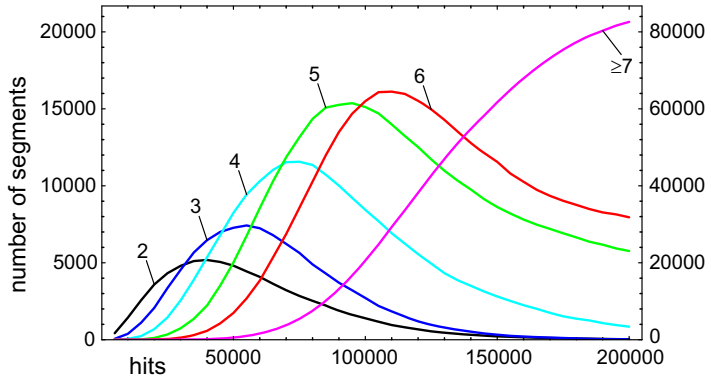


Figure 4.13: Number of track candidates of a given size versus the number of background hits. Each size peaks at a different number of hits as the probability to include k hits in a segment grows with n until the probability to include a $k+1^{\text{th}}$ hit becomes higher. The curve for segments with more than 7 hits (tail-mark length) is scaled by a factor $1/4$ (scale on right-hand side).

The probability to get a segment with two hits is given by the probability to have at least one link from the starting point and no links from any of the second hits. If the volume fraction for accepting a third hit is c_3 and there are k second hits, then the probability that none of these k hits accepts a third hit is given by $P(0; c_3\, n)^k$ provided

that volumes do not overlap. The number of segments which include exactly two hits is then:

$$\begin{aligned} P(1 \to 2) &= \sum_{k=1}^{\infty} P(k)\, k\, e^{-k\, c_3\, n} \\ &= c_\ell\, n\, e^{-\left(c_3 + c_\ell - c_\ell e^{-c_3\, n}\right) n} \quad . \end{aligned} \tag{4.10}$$

The volumes are very likely to overlap and therefore not independent. In practice, the formula $s_k(n) = c_0\, n^k\, e^{-n(c_1 - c_2\, \exp[-c_3\, n])}$, which has the same form as equation (4.10) but groups coefficients, fits the curves of Figure 4.13 reasonably well.

Segment search timing

For a uniform background, the number of links per hit should simply be given by $\rho\, v_\ell = c_\ell\, n$, which is consistent with the expectation value of the Poisson distribution. If the time spent in the criterion's acceptance calculation is dominant, then the time for tracking n hits can be estimated as follows. For the n starting hits, the algorithm creates segment seeds with a fraction a_2 of the $c_\ell\, n$ linked hits. The fraction a_2 is less than one if marked hits (segment growing is actually a depth-first search) or hits already on other tracks are excluded. For each of these seeds of size two, there are $c_\ell\, n$ new links to check of which a fraction a_3 yields segments of size three. In total there are then $a_2 a_3 \times c_\ell^2 n^3$ segments of size three. For each of these segments there are again $c_\ell\, n$ linked hits, and so on. The number of acceptance calculations is then given by:

$$\begin{aligned} N_{\text{accept}} &= \sum_{k=2}^{k_{\max}} c_\ell^{-1} \left(\prod_{j=2}^{k} a_j \right) (c_\ell\, n)^k \\ &= \sum_{k=2}^{k_{\max}} (c_k n)^k \quad . \end{aligned} \tag{4.11}$$

In the second equation, all coefficients for the term with n^k are absorbed in a single constant c_k. The upper limit $k_{\max}$ of the sum is determined by the point where the number of accepted hits decreases rapidly. From equation (4.11) one can deduce that if $c_k n > 1$ for all k, the algorithm's running time and the number of segments will diverge, because for all segments there will always be more than one hit to extend it with. In practice this boundary condition is higher due to the effect of tail marking.

Figure 4.14 shows the segment-search time for n background hits with no tracks. The tracking time increases rapidly for $n > 10^5$, which is also the point where the number of segments with more than 7 hits increases rapidly, as can be seen in Figure 4.13. Equation (4.11) suggests that the tracking-time can be fitted by a polynomial of order $k_{\max}$. A 5^{th} degree polynomial seems indeed to fit the tracking time reasonably well. The residuals show that the fit is less good above 125,000 hits where the tail-marking effect becomes important and where exclusion of already assigned track hits lowers the number of starting points.

4.4 Tracking efficiency for simulated data

In order to estimate the performance of the tracking algorithm, a simple Monte-Carlo simulation program has been used. In this, track and background hits are generated that can be used by the tracking code. The simulation is a simple model of the real CHORUS emulsion data and does not include any physical models of particle propagation and hit formation in emulsion, emulsion distortion, or emulsion scanning. The track and noise

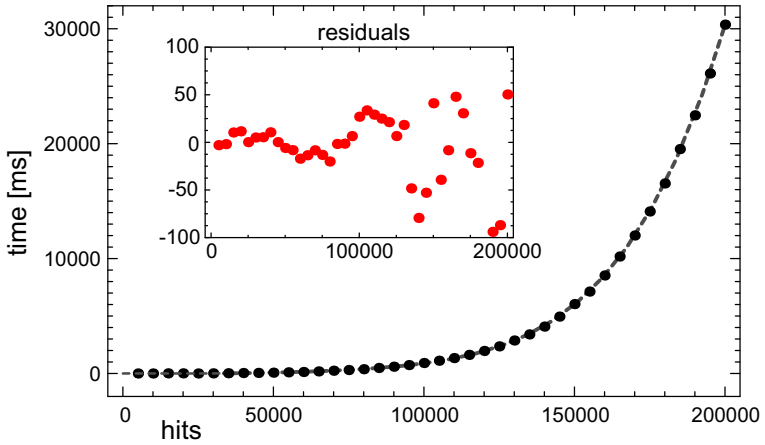


Figure 4.14: Tracking time as function of number of background hits. The dashed line is the result of a fit to the points with the function $\sum_{k=1}^{k=5}(c_k n)^k$:
$\text{time}(n) = (n/826)^1 - (n/2581)^2 + (n/5731)^3 - (n/11511)^4 + (n/20675)^5$.

hit-generators use parameters determined from measured tracks and scanned emulsion. Without distortion, tracks are perfectly straight and easily found. A simple distortion model was therefore introduced to test the tracking in real conditions.

The program uses the hit generators described in the next section. It uses an instance of `CSegmentBuilder` and (one of) its associated criteria (see section 4.4.2) to do the tracking. The actual analysis is done by one or more analysis classes that can be plugged into the simulation program. The analysis classes use a general framework for creating, filling and saving histograms or n-tuples to disk. The histograms or n-tuples can be displayed or further analyzed using Mathematica [252]. The results of two analyses are presented in section 4.4.4. The first analysis yields statistics of tracking time and the number of fake and found tracks. In the second analysis, tracking and hit-assignment efficiencies are quantified. Other analysis plug-ins were used to histogram the differences between the track parameters of the Monte-Carlo and the reconstructed tracks or to write all tracks to n-tuples for more detailed studies.

4.4.1 Hit generators

The generators use an occupation grid for generating hits. Each bit in the grid represents a small volume in 3-D space. The grid is used to avoid generating multiple hits with similar coordinates and takes into account that hits cannot overlap in the emulsion. The hits, like in real data, can still be spaced arbitrarily close, because the position of a hit inside a grid cell is random. The occupation-grid's cell size reflects the scanning microscope's resolving power for neighbouring grains (xy). The spacing in depth (z) is set either to a fixed layer spacing or to the microscope's depth of field, depending on whether the generators are setup for fixed layers or for real 3-D hit positions. When hits are generated for fixed z-layers, the generated 3-D hit positions are projected onto the layer. For hits on a track, the slope of the track is thus not taken into account, in accordance with the grains seen by the microscope. Any grain detected in the depth of field of the microscope is assigned a z-coordinate at the center of that layer.

Background generator

The background generator is rather straightforward. It generates uniformly distributed random positions for a requested number of hits. The hits are marked as background to distinguish them from hits belonging to tracks. Track-correlated background hits (like delta-rays) can be important in the real data. However, this is not simulated.

Track generators

The track generator is chosen at run-time. The simplest track generator creates straight tracks. An extended version, used in the results presented in section 4.4.4, distorts the tracks using a simple distortion model explained below. The basic track generator is set up with the following parameters: fixed layers or free 3-D coordinates; a variable number of tracks originating from a common vertex; the range in z where tracks can start and end; a minimum track length in case a track leaves the volume; and a distribution for the track slope (θ) with respect to the z-axis. Another set of parameters determines the distribution of hits along a track: the hit residual, defined as the sigma of a normal distribution modelling the perpendicular distance to the track; the probability of having a hit on a layer in the case of discrete layers, or the parameters of a Poisson distribution for free hit positions. In the CHORUS experiment, the number of hits per unit track length for a fixed number of layers is accurately described by a binomial distribution. This binomial distribution is the cumulative effect of, among others: intrinsic emulsion sensitivity, blind spots in the emulsion, microscope depth of field, layer spacing, image filtering, and grain detection. The use of a constant hit probability per layer is justified because it also leads to a binomial distribution for a fixed number of layers. When using free hit positions, the response of a uniform emulsion to a traversing particle is modelled. The model uses a fixed probability per unit length of having a developed emulsion grain. This also leads to a Poisson distribution for the number of grains on a fixed length of track.

For the distorted-track generator, only the most common form of distortion is modelled. One end of the tracks (at $z = 0$) is shifted by a random distance δ from its original starting point. The shift $\delta(z)$ is a quadratic function of z, such that the original track's direction is preserved at $z = 0$, $d\delta(z)/dz = 0|_{z=0}$, and the track's position at the other end is fixed, $\delta(z_{\max}) = 0$.

In the results presented here, no vertices were generated. Hits close to the vertex can usually be assigned to multiple tracks, an effect which requires a study by itself, but is not relevant to the track-finding efficiency discussed here. The other settings were chosen to reflect the typical CHORUS emulsion scanning values: 25 fixed layers of $4\,\mu$m thickness; tracks enter from the bottom at $z = 0$ and exit via the top or sides of the volume; minimum track length for detection is 14 layers; and a ± 300 mrad wide uniform distribution of track slopes. The hit residual is set to $0.38\,\mu$m and the probability of detecting a hit on a layer is set to 75 %, lower than that of the real data which has 86 % hit efficieny per layer. The goal of this study is to estimate the track-finding efficiency for recognizable tracks, therefore tracks which have not enough hits to pass the final acceptance criteria of the criterion class (see section 4.4.2) are discarded. Distortion parameters were set according to a normal distribition with a mean shift of $0\,\mu$m and a sigma of $5\,\mu$m for all tracks in a single set. Each track has an additional random shift with a sigma of $0.63\,\mu$m, 1.7 times the assumed track residual.

The track hits are merged with the generated background. In order to compare reconstructed tracks on a hit-by-hit basis with the generated tracks, the track to which each hit belongs is recorded. In the analysis of reconstructed tracks, the Monte-Carlo track with the largest number of hits in a reconstructed track is considered to be the matching Monte-Carlo track. A reconstructed track with only background hits is considered a fake track.

4.4.2 Acceptance criteria

As discussed in section 4.3, the criterion implementation has to make all decisions whether to accept hits in a segment or not. As it is called very often, its execution must be fast. Using vectors for the track's parameters, one can avoid time-consuming calculations involving trigonometric functions. The points $\overline{p}$ on a straight line, parametrized by a single parameter λ, can be represented by:

$$\overline{p}(\lambda) = \overline{b} + \lambda\overline{r} \quad , \tag{4.12}$$

with a base vector $\overline{b}$ and a direction vector $\overline{r}$. In this model there are two free parameters, $\overline{b}$ can be any point on the line and the length of $\overline{r}$ is not fixed. Using vector products (dot and cross-products), all calculations for distances and angles consist of fast multiplications and additions as shown in Figure 4.15. The division is optimized out as a multiplication with $1/\overline{r}\cdot\overline{r}$. This vector model has as advantage that it is applicable in any dimension and that it is isotropic (no prefered direction).

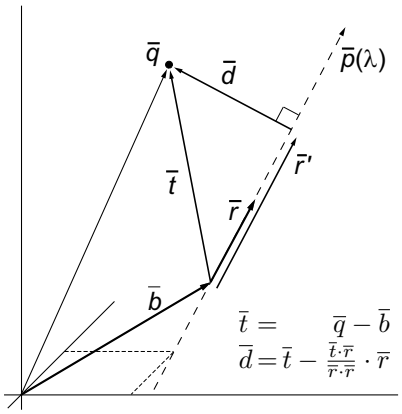


Figure 4.15: Straight track representation using vectors in 3-D, but applicable in any dimension. The vector equation for calculating $\overline{d}$ from the straight–line parameters, $\overline{b}$ and $\overline{r}$, and the position of the hit, $\overline{q}$, are shown in the figure.

As discussed in section 4.1.3, tracks can be considered straight only for short sections of their path. To get the local direction for the prediction, the criterion uses only the last k of n hits in a segment. The criterion uses two running averages:

$$\begin{array}{rcl} \overline{S}_{\text{low}} & = & \sum_{i=n-k+1}^{n-k/2} \overline{p}_i \quad , \text{ and} \\ \overline{S}_{\text{up}} & = & \sum_{i=n-k/2+1}^{n} \overline{p}_i \quad , \end{array} \tag{4.13}$$

with $\overline{p}_i$ the position of the i^{th} hit. The direction is estimated using the difference of the upper and lower half of the end of the segment: $\overline{r} = \overline{S}_{\text{up}} - \overline{S}_{\text{low}}$. The base vector is calculated from the average position of the last k hits in the segment: $\overline{b} = k^{-1}\left(\overline{S}_{\text{low}} + \overline{S}_{\text{up}}\right)$. The running averages can be updated quickly for each hit added to the segment by simply subtracting the hit that has been dropped and adding the new hit. If a segment has

fewer than k hits the sums in equation (4.13) run over $\lfloor n/2 \rfloor$. If n is odd, the middle hit is not used for the direction estimate but is added to determine the average position $\overline{b}$.

In the design of a hit-acceptance criterion there are typically two quantities that determine how well a hit matches a prediction obtained from previous hits, namely the transverse distance to the prediction $\left(\left|\overline{d}\right|\right)$ and the accuracy of the prediction. Two criteria were implemented. The first uses a cylindrical acceptance region for new hits using only the transverse distance. The second takes the uncertainty of the direction into account by defining an acceptance region with the shape of a topped-cone. The cut on $\left|\overline{d}\right|$ is then a linear function of the extrapolation length of the prediction $\lambda = |\overline{r}\,'| \,/\, |\overline{r}|$ (see Figure 4.15).

If all hits have the same position uncertainty, a prediction obtained with fewer hits has a larger uncertainty. Assuming the direction vector of the tracks runs mainly in the positive z-direction, the track slope with respect to the z-axis is given by:

$$\begin{aligned} \tan\theta_z &= \Delta\overline{xy}/\Delta z \\ &= \left(\sum_{up} \overline{xy}_i - \sum_{low} \overline{xy}_i\right) / \left(\sum_{up} z_i - \sum_{low} z_i\right) \quad , \end{aligned} \tag{4.14}$$

where the sums run over $n/2$ hits. If the uncertainty in the xy-coordinate σ_{xy} is independent of the uncertainty for the z-coordinate σ_z, propagation of errors in equation (4.14) yields:

$$\sigma(\tan\theta_z) = \frac{2}{\sqrt{n}\Delta z}\sqrt{\sigma_{xy}^2 + \sigma_z^2 \tan^2\theta_z} \quad . \tag{4.15}$$

Assuming that the hits are evenly spaced by the fixed layers, $z_i = i \cdot \Delta l$ with Δl the layer spacing, the sums for Δz in equation (4.14) give $\Delta z = n \cdot \Delta l/2$. This leads to the following dependence of $\sigma(\tan\theta_z)$ on n:

$$\sigma(\tan\theta_z) = \frac{4}{n\sqrt{n}}\frac{\sigma_{xy}}{\Delta l}\sqrt{1 + \frac{\tan^2\theta_z}{12}\left(\frac{\Delta l}{\sigma_{xy}}\right)^2} \quad , \tag{4.16}$$

using $\sigma_z = \Delta l/\sqrt{12}$ as the uncertainty for hits distributed uniformly within the thickness of a layer. The uncertainty for the prediction's direction decreases therefore as $n^{-3/2}$. Both criteria take this effect into account by letting the cut applied to $\left|\overline{d}\right|$ vary with n as long as $n < k$.

The criteria are tuned by histogramming $\left|\overline{d}\right|$ as function of λ and n for segments created directly from the Monte-Carlo tracks. The cuts are determined by requiring a certain fraction of these distributions to be contained. In the results given in section 4.4.4, a containment of 95 % was used. The number of hits k used to generate the predictions was set to 6. The value k must be chosen such that the direction of the last piece of the segment is determined sufficiently accurate with respect to the position resolution of the hits, provided that other effects (distortion, magnetic field) do not affect the resolution. The cone-opening angles thus found are as expected from equation (4.16).

The track-walk algorithm used in both criteria first calculates track positions on all internal layers between its first and last hit, before extrapolating. Any additional hits compatible with the track are used to recalculate track parameters for the next step. The implementation of the `HandleOverlappingSegments` method in the criterion is based on a comparison of the track parameters. Only if two track candidates have similar position

and direction are they compared on a hit by hit basis. If the two track candidates share most of their hits, only the best of the two is kept. Two track candidates are merged if they share a significant part of their hits.

4.4.3 Efficiency criteria

The main objective of the simulation is the determination of the tracking efficiency: how many of the generated tracks are reconstructed. The efficiency depends on the parameters of the track-finding algorithm. These parameters also influence the tracking time. The simulation is used to choose the trade-off between efficiency and tracking time. The reconstruction efficiency ε is given by the ratio of the number of found tracks and generated tracks. The Gaussian approximation of the binomial distribution for large N is used to determine the uncertainty on the calculated efficiency from $\sigma_\varepsilon = \sqrt{\varepsilon(1-\varepsilon)/N}$, with N the number of generated tracks. For small N and when ε approaches its limits 0 and 1, the Gaussian approximation is no longer valid.

Other quantities can be studied in the simulation, for example: reconstructed versus real track parameters, number of background hits assigned accidentally to real tracks, hit residuals for reconstructed tracks, etc. The results presented here include the hit-to-track assignment efficiency and the number of background hits accidentally included. The hit-to-track assignment efficiency is given by the ratio of the generated track hits assigned to a reconstructed track and the total number of hits in the simulated track.

4.4.4 Results and discussion

The tracking code has been applied to simulated track and background hits in 27 different configurations for the two criteria implementations. Each configuration has been run 125 times with different sets of 80 tracks and a different background of 50,000 hits every five runs. As already discussed in section 4.3.1, the combination of initial link ordering and followed link ordering gives the best results and is also the fastest. The largest effects of disabling one or both of the link-ordering options are to lower the hit-to-track assignment efficiency and to increase tracking time.

Table 4.1 gives the results with both options enabled as function of the tail-marking length and the number of layers that links can span. The 'found tracks/run' entries list the average and RMS of the number of found tracks that are genuine. It can be higher than the number of generated tracks because a single track can be found several times, for example when it gets split in several pieces. The average multiplicity for split tracks (so excluding the ones found only once) is given in the row marked 'multiplicity'. The third part of the table lists the results as they are after running the tracking algorithm on the input data. The phase-II hit-look-up code (section 4.3.2), contains a simple algorithm to merge overlapping or split tracks. The results after application of this step are reported in the fourth part at the bottom of the table.

The reconstruction efficiency is derived from the total number of tracks not found from the 10,000 generated tracks. From the table it can be seen that to increase the track-reconstruction efficiency, one has to increase the number of layers that links can span. For three layers, the reconstruction efficiency is already close to 100 %. The tracks that are not found are mainly due to a combination of several gaps with missing hits and a background hit in or near the gap that pulls the segment away from the actual track. From the table it can be seen that the more layers that links can span, the higher the

average multiplicity becomes. At first glance, the opposite is expected as the tracking can bridge longer gaps with no hits. The reason for the higher multiplicity is that overlapping tracks are allowed by the algorithm. If links span more layers, more small segments are found that consist of background hits. Some of these background segments are then connected to a real track and acquire enough hits for final acceptance as a track candidate. With increasing size of the link-acceptance region, the total number of links and therefore the tracking time increases. The number of track hits included in a track candidate ('hit-assignment efficiency') also increases with a bigger acceptance region. However, the amount of background hits within the link-acceptance region increases proportional to the region's size, which leads to more fake tracks. Both for three and four acceptance layers, the tracking efficiency is about 99.5 %. Except for the tracking time due to the number of links, there is only a small 2 % difference in hit-assignment efficiency between three and four acceptance layers. The lower hit-assignment efficiency is because track segments are shorter and therefore less well determined when a smaller number of layers can be spanned by links.

The tail-marking length does not affect the reconstruction or hit-assignment efficiency very much, but does have a large effect on the tracking time. A tail-marking length $t = 7$ is 25 % slower than $t = 3$ but yields slightly better determined track parameters (not shown), while $t = 11$ increases tracking time by a factor of 2 with almost no gain. The reason for the small effect of this parameter is the link-ordering applied during segment growing. If followed link ordering is disabled, this parameter becomes more important.

The last part of the table shows the efficiencies after picking up additional hits close to the reconstructed tracks and merging overlapping track candidates. Adding close hits improves the hit-assignment efficiency to almost 100 %, at the cost of a bit more included background. The description at the end of section 4.4.2 explaines how the criterion decides to remove or merge overlapping track candidates. The merging of track candidates leads, however, to a small ($\approx 0.2\,\%$) loss in the number of found tracks. A closer investigation showed that the few real tracks that are lost actually get merged with another close-by real track.

Another set of simulation runs showed that the reconstruction efficiency remains at almost 100 % for all background levels between 50,000 and 250,000 hits (highest value tried). The number of fake tracks, though, increases from about 13 per run at 50,000 background hits to 26,500 per run at 250,000 background hits. Therefore, one should either restrict the hit acceptance criteria or remove the fake tracks by examining the χ^2 of the hit residuals for higher background levels.

Table 4.2 reports the results for the cone-shaped acceptance criterion. Due to a square root in its acceptance calculations it turns out to be a factor 1.6 slower for the largest number of links (4 accepted layers). Because its hit-acceptance region is more restricted, this criterion creates about a factor 3 fewer fake tracks and also includes slightly fewer background hits in the tracks. The cone-shaped acceptance criterion rejects hits on the tails of the hit-residual distribution ($|\overline{d}| > 3\sigma_{\mathrm{hit}}$), whereas the cylindrical acceptance region is largely oversized ($5.5\sigma_{\mathrm{hit}}$ for short prediction extensions) and accepts these outliers. As a result, the efficiency using the cone-shaped acceptance criterion is slightly smaller (0.2 %). The hit-assignment efficiency is reduced by 2 % in the results from the segment-growing procedure, but 0.2 % higher after close hit look-up. This increase is due to a better determination of the track parameters because there are fewer background hits included.

Cylindrical acceptance criterion

Tracking configuration values									
Accepted layers	2			3			4		
Tail-marking limit	3	7	11	3	7	11	3	7	11
Tracking input parameters and input processing time									
Number of runs	125								
Background hits/run	50000								
Tracks/run	80								
Hit probability/layer	75 %								
Track hits/run	1490 ± 19								
hit sorting time [ms]	32								
Tree build time [ms]	15								
Link building time [ms]	75 ± 5			134 ± 5			213 ± 5		
Number of links	$27,876 \pm 184$			$73,981 \pm 349$			$152,233 \pm 630$		
Segment searching results									
Segment search time [ms]	17 ± 5	19 ± 3	33 ± 6	92 ± 4	115 ± 6	245 ± 26	349 ± 8	440 ± 11	792 ± 59
Found tracks/run	87.8 ± 3.9	88.2 ± 3.9	87.7 ± 3.8	101.5 ± 6.5	101.6 ± 6.1	98.3 ± 5.7	108.0 ± 7.8	108.1 ± 8.7	101.2 ± 7.6
Fake tracks/run	1.8 ± 1.4	1.9 ± 1.3	1.9 ± 1.3	11.3 ± 2.9	13.6 ± 3.8	13.6 ± 3.7	11.2 ± 3.4	14.8 ± 4.1	14.9 ± 4.0
Multiplicity found tracks	2.1 ± 0.3	2.1 ± 0.3	2.1 ± 0.3	2.4 ± 0.8	2.5 ± 0.8	2.4 ± 0.7	2.7 ± 1.0	2.8 ± 1.1	2.7 ± 1.0
Background hits/track	0.86 ± 0.98	0.85 ± 0.98	0.83 ± 0.97	0.94 ± 1.02	0.92 ± 1.03	0.91 ± 1.01	0.91 ± 0.98	0.88 ± 0.98	0.86 ± 0.97
Hit-assignment efficiency [%]	78.32 ± 0.10	78.71 ± 0.10	78.96 ± 0.10	90.20 ± 0.07	90.56 ± 0.07	90.81 ± 0.07	92.45 ± 0.06	92.79 ± 0.06	93.06 ± 0.06
Total tracks **not** found	431	415	413	50	47	49	48	47	50
Reconstruction efficiency [%]	95.69 ± 0.20	95.85 ± 0.20	95.87 ± 0.20	99.50 ± 0.07	99.53 ± 0.07	99.51 ± 0.07	99.52 ± 0.07	99.53 ± 0.07	99.50 ± 0.07
After phase 2 hit pickup and track candidate merge									
Phase 2 time [ms]	7 ± 4	7 ± 4	7 ± 5	8 ± 4	9 ± 3	8 ± 4	8 ± 4	9 ± 3	9 ± 3
Found tracks/run	76.3 ± 2.0	76.5 ± 1.9	76.5 ± 1.9	79.9 ± 1.2	80.1 ± 1.2	80.0 ± 1.1	80.5 ± 1.2	80.8 ± 1.5	80.4 ± 1.3
Background hits/track	3.56 ± 1.88	3.56 ± 1.88	3.56 ± 1.88	3.55 ± 1.90	3.55 ± 1.89	3.54 ± 1.89	3.49 ± 1.88	3.49 ± 1.88	3.49 ± 1.88
Hit-assignment efficiency [%]	98.10 ± 0.03	98.15 ± 0.03	98.18 ± 0.03	98.82 ± 0.03	98.85 ± 0.02	98.85 ± 0.02	99.17 ± 0.02	99.20 ± 0.02	99.25 ± 0.02
Total tracks **not** found	469	446	449	81	72	71	69	67	70
Reconstruction efficiency [%]	95.31 ± 0.21	95.54 ± 0.21	95.51 ± 0.21	99.19 ± 0.09	99.28 ± 0.08	99.29 ± 0.08	99.31 ± 0.08	99.33 ± 0.08	99.30 ± 0.08

Table 4.1: Tracking time and efficiency as function of the tracking parameters indicated in the first two rows. The second part of the table gives the tracking input parameters and setup times. In the third part, the track candidates are counted as they are found by the segment builder. The fourth part shows the efficiencies for the case when hits are recovered and (identical) tracks merged. The errors, except for efficiencies, correspond to the RMS values of the underlying distributions. For efficiencies the error corresponds to the statistical uncertainty on its value (see text). An explanation of the different rows is given in the main text.

Cone-shaped acceptance criterion

Tracking configuration values									
Accepted layers	2			3			4		
Tail-marking limit	3	7	11	3	7	11	3	7	11
Tracking input parameters and input processing time									
Number of runs	125								
Background hits/run	50000								
Tracks/run	80								
Hit probability/layer	75 %								
Track hits/run	1490 ± 19								
hit sorting time [ms]	32								
Tree build time [ms]	16								
Link building time [ms]	75 ± 5			135 ± 5			214 ± 5		
Number of links	$27,876 \pm 184$			$73,981 \pm 349$			$152,233 \pm 630$		
Segment searching results									
Segment search time [ms]	15 ± 5	18 ± 4	28 ± 5	99 ± 4	124 ± 6	262 ± 24	581 ± 12	748 ± 17	1301 ± 79
Found tracks/run	84.7 ± 3.6	84.4 ± 3.7	84.1 ± 3.6	96.1 ± 5.4	94.5 ± 4.9	92.1 ± 4.3	99.8 ± 6.5	98.4 ± 6.0	92.8 ± 5.2
Fake tracks/run	0.5 ± 0.8	0.5 ± 0.9	0.5 ± 0.9	4.6 ± 1.6	5.2 ± 2.1	5.3 ± 2.0	4.3 ± 1.8	5.8 ± 2.1	5.9 ± 2.2
Multiplicity found tracks	2.1 ± 0.2	2.1 ± 0.2	2.1 ± 0.3	2.4 ± 0.7	2.5 ± 0.7	2.4 ± 0.6	2.7 ± 1.0	2.8 ± 1.0	2.7 ± 0.9
Background hits/track	0.62 ± 0.81	0.62 ± 0.81	0.61 ± 0.81	0.74 ± 0.89	0.73 ± 0.89	0.71 ± 0.88	0.71 ± 0.87	0.70 ± 0.86	0.67 ± 0.84
Hit-assignment efficiency [%]	75.61 ± 0.10	76.18 ± 0.10	76.48 ± 0.10	87.87 ± 0.08	88.33 ± 0.07	88.70 ± 0.07	90.14 ± 0.07	90.58 ± 0.07	90.99 ± 0.07
Total tracks **not** found	586	555	561	74	72	76	59	73	72
Reconstruction efficiency [%]	94.14 ± 0.23	94.45 ± 0.23	94.39 ± 0.23	99.26 ± 0.09	99.28 ± 0.08	99.24 ± 0.09	99.41 ± 0.08	99.27 ± 0.09	99.28 ± 0.08
After phase 2 hit pickup and track candidate merge									
Phase 2 time [ms]	7 ± 5	6 ± 5	7 ± 5	7 ± 4	8 ± 4	7 ± 4	8 ± 4	8 ± 4	8 ± 4
Found tracks/run	75.0 ± 2.1	75.2 ± 2.1	75.2 ± 2.1	79.4 ± 1.0	79.5 ± 0.9	79.4 ± 0.9	79.8 ± 1.1	79.9 ± 1.2	79.7 ± 1.0
Background hits/track	3.47 ± 1.86	3.47 ± 1.86	3.47 ± 1.86	3.48 ± 1.88	3.48 ± 1.87	3.48 ± 1.87	3.44 ± 1.86	3.45 ± 1.86	3.44 ± 1.86
Hit-assignment efficiency [%]	98.51 ± 0.03	98.54 ± 0.03	98.56 ± 0.03	98.99 ± 0.02	99.09 ± 0.02	99.11 ± 0.02	99.35 ± 0.02	99.40 ± 0.02	99.42 ± 0.02
Total tracks **not** found	631	598	599	105	99	98	84	95	90
Reconstruction efficiency [%]	93.69 ± 0.24	94.02 ± 0.24	94.01 ± 0.24	98.95 ± 0.10	99.01 ± 0.10	99.02 ± 0.10	99.16 ± 0.09	99.05 ± 0.10	99.10 ± 0.09

Table 4.2: As table 4.1, but for a cone-shaped acceptance region.

4.5 The track trigger

The tracking algorithm and its implementation is by design completely isotropic and was developed to find all tracks in a set of emulsion images. As was discussed in the introduction (section 4.1), during scan-back of the target sheets only a small volume of about $50\,\mu\text{m} \times 50\,\mu\text{m} \times 100\,\mu\text{m}$ of emulsion on the upstream surface of a plate is scanned. The only question concerning this volume is whether a track with the predicted slope is present or not. The tracking algorithm, described in this chapter, does not use this additional information. On the contrary, the track selector hardware, described in section 2.9.4, is exquisitely tuned for this job. A similar approach as the track selector applies in hardware can also be implemented in software. The slope information of the predicted track is then used to speed up the track-finding algorithm. The basic idea is that the direction of the track prediction defines positions for acceptable grains on subsequent layers. A similar summing algorithm, as applied in hardware to the pixels of the images by the track selector, is now applied to the grains. The sum serves as a track trigger. The trigger defines a small region of interest in which the full tracking algorithm is then applied. The tracking step verifies the existence of a track, assigns grains to it, and calculates the track parameters.

4.5.1 Concept

The steps taken in the track-trigger algorithm are depicted in Figure 4.16. All grains are given an initial sum value of one. Starting with the first layer, the algorithm iterates over all grains in a layer. It looks up all grains in the next layer within a certain acceptance region. The position of this region is given by the grain position in the previous layer shifted by the direction of the track prediction. The angular acceptance determines the dimensions of the acceptance region. The sum values of all found grains are set to the sum value of the grain in the previous layer incremented by one. However, this is only done if that value is higher than the current value of a found grain. Such a case happens at area E in Figure 4.16. If there are no grains in the acceptance region, a virtual grain is created at the predicted position without incrementing the sum value. This is necessary because a track does not need to have a grain on each layer. To maintain the angular acceptance, the acceptance area in the next layer for these virtual grains is enlarged (see area D in Figure 4.16). The whole procedure is then repeated for the next layers. Throughout this procedure, the information on the position of grains in preceding layers is discarded. As a result, small curvatures due to distortion can be accommodated.

After summing all layers this way, the grains on the last layer with a sum value above threshold indicate the position of a possible track candidate. The tracking algorithm is then applied on the grains inside a rhombohedral volume. The position of the grain in the last layer that triggered the volume is the center of the top rectangle. The lengths of the sides are given by the angular acceptance. The slope of the volume is given by the predicted-track slope. Because this volume contains only a small number of grains, typically 30 to 100 of which about 20 belong to the track, the tracking procedure runs very fast.

4.5.2 Implementation

The implementation of the track-trigger algorithm is straightforward. The grains for each layer are stored in 2-D fixed-range maps. Iterating over all grains in a layer, the look-up of grains in the acceptance region in the next layer is done using these maps. Virtual grains are stored in a temporary linear array. The procedure is then repeated

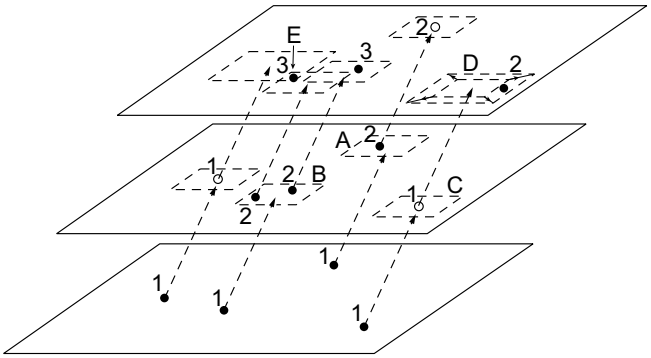


Figure 4.16: Illustration of the grain-summing procedure for the track trigger. For each grain, the predicted-track slope and the angular acceptance define a region in the next layer. The sum value (indicated next to the grains) for the one (A) or more (B) grains in that region is incremented. If no grains are present in the predicted region (C), a virtual grain (open circles) is created at the predicted position. To keep the same angular acceptance, the acceptance region for virtual grains is increased (D). If a grain falls in multiple acceptance regions, the highest value is used for the new sum value (E).

for the next layer. Processing of each layer starts with the virtual grains first. This way, the same array can be reused for storing any new virtual grains, because each virtual grain can create at maximum one new one. If a grain's sum value is too low to pass the trigger threshold taking into account the remaining number of layers, it is immediately discarded.

After processing the one but last layer, the sum values of the virtual grains for the last layer and of all grains on the last layer are compared against a threshold. If the sum value is above threshold, an object is created to hold all the grains in the rhombohedral acceptance volume and the parameters of a possible track. The maps for each layer are used to find the grains inside this volume and insert them in object. Finally, the track-finding algorithm is run over each object. If a track is found which matches the prediction, the track parameters are included in the object, otherwise the object is deleted.

4.6 Application in CHORUS emulsion scanning

The tracking code described in this chapter has been applied to the emulsion data taken by the CHORUS experiment at CERN. The tracking code could have been applied directly online, but in most cases it is not sufficiently fast on the microscope control systems. Typically, a tomographic image set of 25 images of $300\,\mu\text{m} \times 300\,\mu\text{m}$ is acquired every two seconds. The tracking time for such an image set on the online computer[5] is somewhere between several seconds to half a minute, depending on the grain density. Applying the track-finding algorithm online would therefore introduce considerable overhead and would leave the microscope idle. Two solutions have been adopted to eliminate this overhead.

First, for scanning areas that consists of one or more full microscope views, the scanning system writes the grain data to an object-oriented database. In the emulsion

[5] Dual-processor 500 MHz Pentium-III.

scanning at CERN, the grain data is stored for all scan-back on the interface sheets, for all net-scan, and for full vertex-reconstruction data. These grain data are then processed offline, while the data taking of the emulsion continues online. For processing the grain data offline, a main program was developed. This program reads the grain data from the database, applies the tracking algorithm, and writes the found tracks to the same database. The track data consist of the track parameters and the grains in the small volume around it. The parameters used by the tracking algorithm are also stored in the database. In normal operation, many instances of this program run on a computer farm, consisting of standard PCs running the Linux operating system. Each instance processes a different event and the distribution of the events is controlled by an external program. This program communicates with the tracking programs via the network using a message dispatching server [253].

The second solution is to increase the track-finding speed by using the angular information available in scan-back. The track trigger (section 4.5) is used for scan-back in the target sheets, where an area of only $60\,\mu\text{m} \times 60\,\mu\text{m}$ is scanned. The track-finding efficiency during scan-back of the target sheets is also less important, because two consecutive missed plates are required for scan-back to stop for a prediction (see section 2.10.2). Therefore, in the tuning of the track-trigger procedure, efficiency can be traded in for speed. Typical tracking time using the track trigger on target sheet data is about 0.5 seconds. The tracks found online during target-sheet scan-back are also written to the database. The grain data around the track is stored as well.

The tracking results stored in the database are available for further analysis. Because much of the grain data is stored in the database, many options are available for analysis. One can, for example, refit tracks, rerun track finding, or look for hot pixels in the data and eliminate fake tracks based on the grain positions. A typical analysis task needed for CHORUS is the matching of predictions to found tracks and the generation of new scanning predictions for the upstream emulsion plate. Because of the new approach to the scanning of the interface sheets, a more efficient candidate matching procedure could be applied which is described in section 4.6.2. The matching involves determination of the precise alignment for which a new strategy has been developed. This local-alignment strategy is discussed in section 4.6.3.

4.6.1 Tracking configuration

The main difficulty in applying the tracking code to real emulsion data lies in the large variability of the actual data. Emulsion differs from plate to plate and even within one plate. Effects like shrinkage and distortion, grain size, and fog density can change from one microscope view to another (section 2.9.2). The physical grain size and the fog density depend on the development process. Some of these variations arise from the scanning conditions, like transparency of the emulsion layer or optics tuning. Shrinkage and distortions are unavoidably different for each plate and depend also on the relative water content of the plate when scanned. Part of this variability is already dealt with while scanning, for example by changing the illumination or exposure time, and part, like contrast and grain size, is dealt with by the grain recognition algorithm (section 4.1.1). As the input to the tracking code has large variations in the number of grains per view, it is difficult to predict the tracking time required, which excluded running the tracking online. The variation in distortion and shrinkage implies that the hit-acceptance criteria have to be insensitive to these parameters.

To deal with these variations, the criterion class implemented for emulsion tracking uses simple but fast acceptance calculations, relatively large acceptance criteria, and a small number of consecutive hits for making the prediction. The acceptance criterion has been designed based on real emulsion data. The criterion implementation is identical to the cylindrical criterion described in section 4.4.2. To accomodate the distortions, only the last six grains in a segment are used to define the track prediction. A track segment of six consecutive grains has a typical length of about 20 μm and is therefore unlikely to be affected much by distortions. The acceptance radii for additional grains depend on the number of grains already in the segment and are set as follows: 1.6 μm for a 2 grain segment, 1.4 μm for 3 grains and 1.2 μm for 4 or more grains. Although this last value is large with respect to the track residual (5σ), the grain density in the emulsion is sufficiently low that not too many background grains are picked up.

The other settings of the criterion and tracking code were determined by the characteristics of the events in CHORUS. Event-related tracks lie in a forward cone (relative to the direction of the incident beam) with 400 mrad half-opening angle. As the layer-to-layer uncertainty of the grain positions corresponds to about 100 mrad, the link acceptance region is therefore limited to a forward cone with 500 mrad half-opening angle. The grain-detection efficiency per layer has an average value of about 86 %. In order to have high tracking efficiency, the link-acceptance region has been set to span four layers. The tail-marking length parameter was set to 7 hits. The track-walk code used in the close-hit look-up procedure, picks up all grains in a radius of 1.5 μm around the track candidate, but only grains closer than 0.6 μm are used in a new track fit. Track candidates which pass a final selection criterion based on the number of grains on the track and the grain density are then handed of to the application using the tracking code.

4.6.2 Prediction matching on the interface sheets

During scan-back, the candidates for a prediction are selected using a χ^2-probability cut, $P\left(\chi^2\right) > p$, based on four quantites: $\Delta x,\ \Delta y,\ \Delta\theta_X,\ \Delta\theta_Y$. The exact cut value p is chosen depending on the amount of background present. A procedure that derives the real efficiency from the number of multiple candidates shows a clear plateau below some value for p. A value slightly below the onset of this plateau is then used for the cut value p. The covariance matrix needed to calculate $P\left(\chi^2\right)$ is derived from the data. In a first approximation, the covariance matrix is taken to be diagonal and defined by the sigmas extracted from Gaussian fits to the distributions of the residuals. In an iterated procedure, the covariance matrix is used to select candidates and then recalculated from the correlations between predictions and candidates.

Originally in scan-back on the changeable and special sheet, the matching required four segments to be found that are matched with the base measurement of both emulsion sheets, indicated by the check marks in Figure 2.20. The base measurement is required for an accurate slope measurement to reduce pick-up of accidental background (section 2.4.2). With this selection, the efficiency for matching a track to the prediction includes a factor

$$\varepsilon_{\mathrm{f}}^4 \varepsilon_{\mathrm{base}}^2 \quad , \tag{4.17}$$

where ε_{f} is the probability of finding a track segment on one side of the emulsion and $\varepsilon_{\mathrm{base}}$ the efficiency of connecting two segments over the plastic base. Inefficiencies lowering the value of e_{f} are very costly due to the power of four of ε_{f}.

In the scanning approach used at CERN, the changeable sheet and special sheet are scanned independently. As the tracking algorithm does not look for the scan-back prediction in particular, many tracks are reconstructed on both sides of the plastic base. These additional tracks make some extra options for matching the found tracks to the prediction available. With sufficient tracks found on both emulsion layers and matched over the base, the distortion and shrinkage of the emulsion layers can be corrected using the base-slope measurements. This reduces the uncertainty in the measured direction of track segments found in each emulsion layer to about 4 mrad. These single-sided track segments can then be matched independently to a prediction. At CERN, this has been used to match candidates requiring only **three out of four** segments. A prediction has to match at least one track with a base measurement on either the changeable or special sheet. In addition, that matching track, extended to the other plate, must have either a matching base track or one matching segment on one emulsion side. The efficiency for matching predictions to track segments in the changeable and special sheets then contains the factor:

$$\varepsilon_{\mathrm{f}}^4 \varepsilon_{\mathrm{base}}^2 + 4\varepsilon_{\mathrm{f}}^3 (1-\varepsilon_{\mathrm{f}}) \varepsilon_{\mathrm{base}} \quad . \tag{4.18}$$

Assuming $\varepsilon_{\mathrm{base}}$ to be 1 and setting conservatively $\varepsilon = 0.95$ for the CERN system, the value of equation (4.18) is 98.6 %. This should be compared to equation (4.17) which yields 92.2 % for the quoted track-selector efficiency of $\varepsilon_{\mathrm{f}} = 0.98$.

There is however a complication in the distortion and shrinkage correction procedure. The distortion varies from one small piece of emulsion to the next. The typical scale over which the distortion changes is about a centimeter, but sometimes it can change rapidly within less than 100 μm. If such an abrupt change happens within the area scanned for a prediction, a global distortion correction for the whole area cannot be applied. To overcome this problem, a local distortion correction is applied instead. First the shrinkage factor, which changes over much larger scales than distortion, is determined from all tracks in the scanned area. The distortion for every segment is then determined from about 20 nearby tracks with a base measurement. The base slope of these tracks is used to estimate the local distortion at the position of the segment under consideration. A 2-D map is used to look up the nearby tracks. This map contains the positions of all tracks with a base measurement.

4.6.3 Local-alignment procedure

A similar problem with local variations was also found to exist in the alignment of the interface sheets. Although the interface sheets are quite rigid, a global alignment, calculated from minimizing the distances between all found predictions and their candidates, showed residuals which depend on the position and slope. Further investigation showed that these variations were probably due to deformation of the interface sheets at boundaries of the honeycomb support structure on which they were mounted.

These kind of local deformations were corrected for by applying a local-alignment correction. This is done efficiently using again a 2-D map. The map contains a set of track prediction and candidate pairs, indexed by position. A local-alignment correction can then be applied to any track prediction. This correction is calculated from a number of prediction-candidate pairs (usually about 50, depending on the track density) in the vicinity of the prediction being corrected. Because the alignment correction is local (typically several square centimeters), it takes into account the remaining deformations

of the sheets. This local-alignment procedure improves the overall resolution and reduces the variation in the residuals. By reducing position and slope dependent residuals, it also reduces the unknown bias in the candidate selection based on a χ^2 probability cut (section 4.6.2) from the position and slope depedence of the covariance matrix.

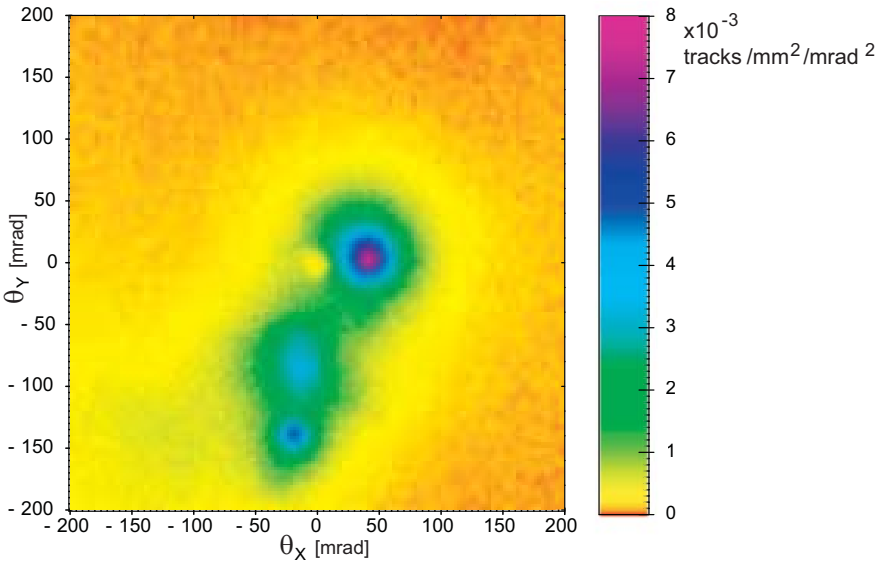
4.6.4 Tracking results

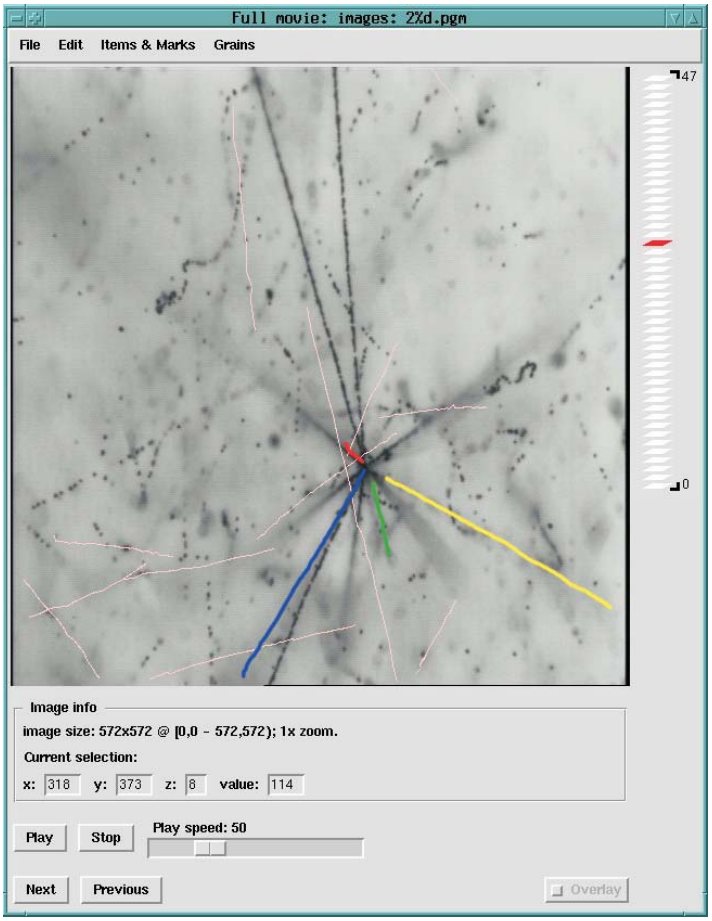
Some of the results obtained with the tracking code applied to emulsion data are shown in the following. In Figure 4.17 the slope distributions of all tracks found on a special sheet are shown. The peak from beam muons at $\theta_x = 42\,\mathrm{mrad}$ is clearly visible. The typical track density on the special sheet is more than a factor two lower than that for the target sheets as they are exposed during only seven months before development. The target sheets are exposed for two times this period. In addition, background is accumulated during the five months that they are stored underground between two runs. From this figure, the necessary accuracy for making predictions can be deduced (section 2.4.2).

The result of running the track reconstruction on a set of images containing a neutrino-interaction vertex is shown in Figure 4.18. The emulsion slice containing the interaction vertex is displayed in Figure 4.18a and shows the typical star of black tracks due to the break-up of the nucleus. The tracks of interest are almost perpendicular to this picture and are only visible as single dots. The tracks reconstructed in the full image set are overlaid in the image. The interaction vertex can be distinguished from the four tracks that point to the same location. Some unrelated tracks can been seen as well. Close to the vertex, the four vertex tracks are not reconstructed, because they disappear in the black region caused by the nuclear break-up. A 3-D representation of the tracks and grains attributed to these tracks close to the vertex is given in Figure 4.18b. One track is only reconstructed further away from the vertex and not visible in this close-up of the vertex.

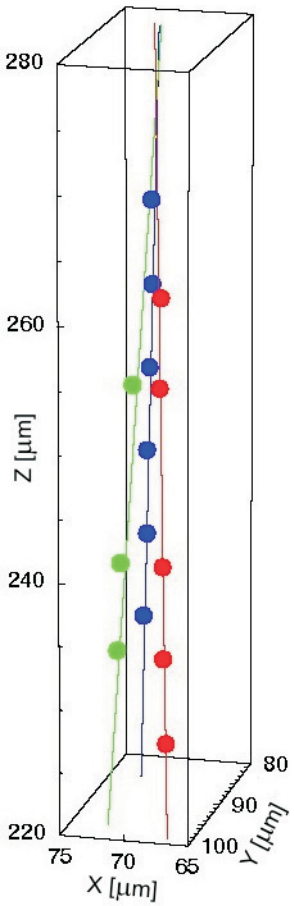
Figure 4.17: [top opposite page] Track density as function of track direction in a single special sheet of one of the corner modules of the emulsion stack. Data is based on 1.42 million tracks in 6156 scanned events with a total scanned area of $130\,\mathrm{cm}^2$. Three peaks of particle directions are clearly visible. The highest peak is from beam-related muons at $\theta_x \approx 40\,\mathrm{mrad}$, $\theta_y \approx 0\,\mathrm{mrad}$. The two other peaks with $\theta_x \approx -100\,\mathrm{mrad}$ are due to background from beams in the adjacent experimental hall.

Figure 4.18: [bottom opposite page] A neutrino interaction vertex in the emulsion. (a) The emulsion layer containing the vertex is shown inside the emulsion player program which allows to view a movie of the tomographic image set. The black tracks are the result of the nuclear breakup. Overlaid on the image are all reconstructed tracks. The four coloured tracks come from the vertex, pink tracks are event unrelated tracks. (b) Shows a close-up of the vertex in 3-D, where one track is not visible anymore as it is only reconstructed further away from the vertex.


200
150
100
50
0
- 50
- 100
- 150
- 200
θY [mrad]
- 200 - 150 - 100 - 50 0 50 100 150 200
θX [mrad]
8
7
6
5
4
3
2
1
0
x10 -3
tracks /mm2 /mrad 2


Full movie: images: 2%d.pgm
File Edit Items & Marks Grains
47
0
Image info
image size: 572x572 @ [0,0 - 572,572); 1x zoom.
Current selection:
x: 318 y: 373 z: 8 value: 114
Play Stop Play speed: 50
Next Previous Overlay

(a)


280
260
240
220
Z [µm]
75 70 65
X [µm]
80
90
100
Y [µm]

(b)

4.7 Conclusion and discussion

The multi-dimensional container classes have proved useful in other applications. For instance, to determine the initial alignment of target sheets, one has to connect tracks from track maps on one plate to the other plate (see section 2.10.1). With several thousand tracks per track map, the look-up of a possible match becomes time consuming, because trying to match each track with all the other tracks has quadratic complexity. Using a 4-D map, the look-up for matching tracks can be done fast. Furthermore, the look-up automatically matches both position and slope of the track within a limited 4-D volume, as the map is indexed by both the position and slope of the tracks. The slope matching reduces the number of fake matches. The maximum number of matches as function of the position offset determines the alignment. Trying out a grid of position offsets yields an initial estimate for the alignment. The set of matched tracks with this offset are then used to fit a refined alignment which also includes possible small rotations and a longitudinal shift.

If one examines the track-finding algorithm, it is clear that the principles on which the track finding is based require the following two conditions:

1. hits that belong together have close-range relationships,
2. a set of hits defines a volume that is defined by the close-range relationships of condition 1.

If examined in a more abstract way, condition 1 is actually not a strict requirement of the algorithm. The track-finding algorithm, described in section 4.3.1, can be used in any situation where a volume can be defined in which other hits of a track can lie with respect to some track hit. As long as that volume is small with respect to the total volume and the volume is continuous in its coordinates, the k-space containers can be used to build up a hit-connection graph. The notion of hits and tracks can also be replaced by other abstractions. The algorithm and its implementation as a C++ toolkit can be used in any environment where nodes in a graph need to be connected according to some acceptance and rejection criteria. As long as the above conditions are fulfilled, the tracking framework can be applied for such cases in any dimensional space.

Separating the acceptance criteria, which involve the actual hit and track model, from the track-finding algorithm has allowed to create a flexible toolkit. As discussed in section 4.3.2, all environment-specific characteristics — like geometry, track propagation, and magnetic fields — are handled by the criterion class implementation which the user has to provide. The toolkit can therefore easily be adapted to other applications. For example, the code has been used to combine five-dimensional track segments (3-D position and 2-D slope) from multiple emulsion sheets into single tracks.

For many track-finding applications, more specific algorithms, for example Kalman filtering [254, 255]), could be more efficient and faster. In general, tracking algorithms need to look up hits in regions of space and the k-space containers can be used to do this efficiently. The track-finding algorithm, described in this chapter, performs very well when applied to the CHORUS emulsion data. It has been shown that the tracking finds almost all tracks despite the high background. Its application has enabled improvements in the vertex-location efficiency in scan-back. If applied to data taken over the full thickness of the emulsion, it can also reconstruct all tracks and vertices in an event. This could be used to reduce the need for human confirmation of secondary vertices after the net-scan procedure. The algorithm has also been applied successfully to the reconstruction of tracks in the HARP time-projection chamber.