42

Tracking the Quark-Gluon Plasma

by

Michael J. Duff

Submitted to the Department of Electrical Engineering and Computer Science and the Department of Physics in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and

Bachelor of Science in Physics

and

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 28, 1997

Author .....
Department of Electrical Engineering and Computer Science
May 28, 1997

Certified by .....
Craig A. Ogilvie
Thesis Supervisor
Laboratory for Nuclear Science

Accepted by .....
June L. Matthews
Senior Thesis Coordinator
Department of Physics

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses
Department of Electrical Engineering and Computer Science

Tracking the Quark-Gluon Plasma

by

Michael J. Duff

Submitted to the
Department of Electrical Engineering and Computer Science and the Department of Physics

May 28, 1997

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Physics, Bachelor of Science in Computer Science and Engineering, and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

One aim of the upcoming STAR experiments at the RHIC particle collider is to identify the quark-gluon plasma on an event-by-event basis. A new approach to tracking - pixel-based parallel tracking - is presented which has promising characteristics for achieving the high tracking efficiency needed to successfully identify the quark-gluon plasma. To meet running time requirements, a new data structure - the octant tree - is introduced which permits rapid extraction of pixels in a subregion of space. A fully operational prototype of the proposed tracker has been designed and implemented and awaits testing on simulated TPC data.

Thesis Supervisor: Craig A. Ogilvie
Title: Assistant Professor of Physics

Acknowledgments

First and foremost, I would like to thank my thesis advisor, Professor Craig Ogilvie, for his guidance and patience in leading me through this last and greatest endeavor of my MIT career. His warm character and kind words were always refreshing after a hard week of work.

I thank my parents and brothers for the loving and supportive environment they have provided throughout my life.

To my best friend, Meaw, I send my dearest thanks for her ongoing companionship and love. Her heart of gold, contagious laugh, and generosity will forever remind me of how she made these past three years the happiest I have known.

And finally, I thank good fortune for carrying me this far. Yet there is still so much to do...

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Quark-Gluon Plasma

A new collider is under construction which is different from others in that it will not be colliding single particles such as protons. Rather, the Relativistic Heavy Ion Collider (RHIC) will collide heavy composite particles such as gold nuclei. Each collision will produce a dense plasma of quarks and gluons which will help physicists study the characteristics of these particles under intense heat and pressure. Such were the conditions instants after the Big Bang. Thus, knowledge of how this "quark-gluon plasma" behaves is expected to help explain how the particles hadronized into the matter that we see today.

The quark-gluon plasma is only a transient state. Once the system expands and cools, the plasma will rehadronize into mesons and baryons. In order to explore the properties of the quark-gluon plasma, therefore, one must be able to detect variations in the signatures left behind. The goal of the Solenoidal Tracker At RHIC (STAR) experiment is to correlate these signatures on an event-by-event basis.

The signatures of the plasma, if present, are expected to be subtle in the sense that the measurable variations will be small. This translates into a requirement that each event be measured well (*i.e.* a large percentage of particle tracks be correctly identified), since inefficiencies in the tracking process would induce fluctuations in the measured quantities. If these fluctuations are as large as the underlying fluctuations, then the event-to-event variations will be masked.

1.2 Project Goals

In order to achieve the required efficiency, a new approach to tracking is needed. This algorithm must remain highly efficient even at high track multiplicities. This requirement is the downfall of current tracking algorithms. Figure 1-1 illustrates the tracker efficiency expected from an ideal algorithm. As

shown in the figure, the algorithm should produce a relatively-flat response over a broad range of multiplicities. The primary goal of this thesis was to design and implement such an algorithm and test it on simulation data.

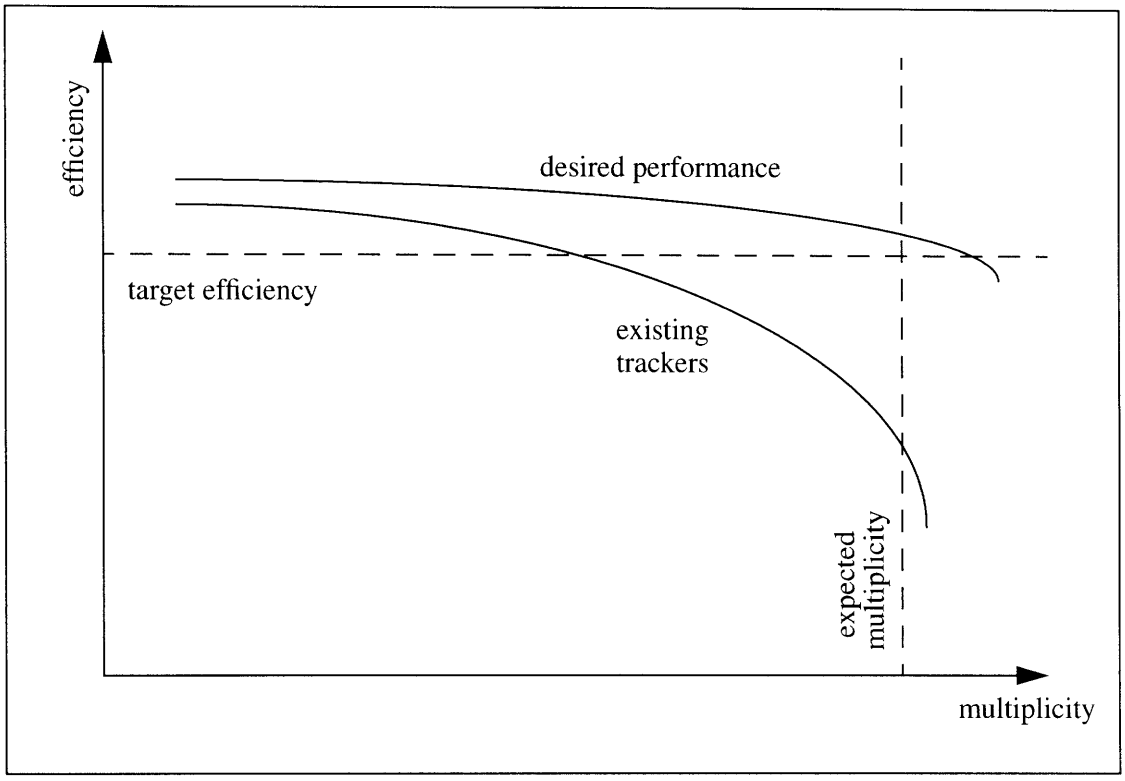

Figure 1-1: Desired tracker efficiency. The tracking algorithm used to search for the quark-gluon plasma should remain at or above the target efficiency at the expected multiplicity for full events. The efficiency of existing trackers declines rapidly with increasing multiplicity, leading to an unsatisfactory performance.

For the purposes of this thesis, tracking *efficiency* is loosely defined as the percentage of correctly-identified tracks. The events measured by STAR will have an expected multiplicity associated with them, and the required efficiency can be estimated. The performance of the tracking algorithm can be judged against these two quantities.

Designing an efficient tracking algorithm is one matter, while implementing one that runs well under the constraints of limited computational resources is another. The running time and memory

requirements of the proposed tracking algorithm present a formidable challenge to the feasibility of achieving the primary goal. The methods used to meet this challenge are discussed in detail as part of this thesis.

1.3 Overview

The purpose of this thesis is to introduce a researcher to STAR and to explain the proposed approach to tracking such that he/she can continue the work that has been completed to date. For this reason, the explanation of the tracker's design and implementation is discussed in depth.

Beginning with Chapter 2, the STAR detector is described along with the basics of tracking. In Chapter 3, the traditional approach to tracking is presented including a discussion of its weaknesses. Chapter 4 introduces the reader to the proposed tracker. Chapter 5 describes important high-level concerns about computational resources that must be considered when implementing the tracker. Next, Chapter 6 explains the implementation of the tracker prototype including explanations of the tracker's tunable parameters. Chapter 7 explains the process of generating test data for the tracker, and Chapter 8 suggests various diagnostics that can be used to evaluate the performance of the tracker. Finally, Chapter 9 suggests ways that the prototype could be improved in terms of speed and tracking efficiency.

Chapter 2

The STAR Detector

2.1 Time Projection Chamber

The detector that will be used for the quark-gluon plasma experiments, STAR, features a Time Projection Chamber (TPC) consisting of the gas chamber, magnets, electrodes, and detector pads. As subparticles scatter from the collision event center, they ionize the gas while passing through it. An electric field is then applied to the chamber to draw the gas ions towards the detector pads at either end of the TPC. When an ion reaches a detector pad, it causes an avalanche of current proportional to its charge.

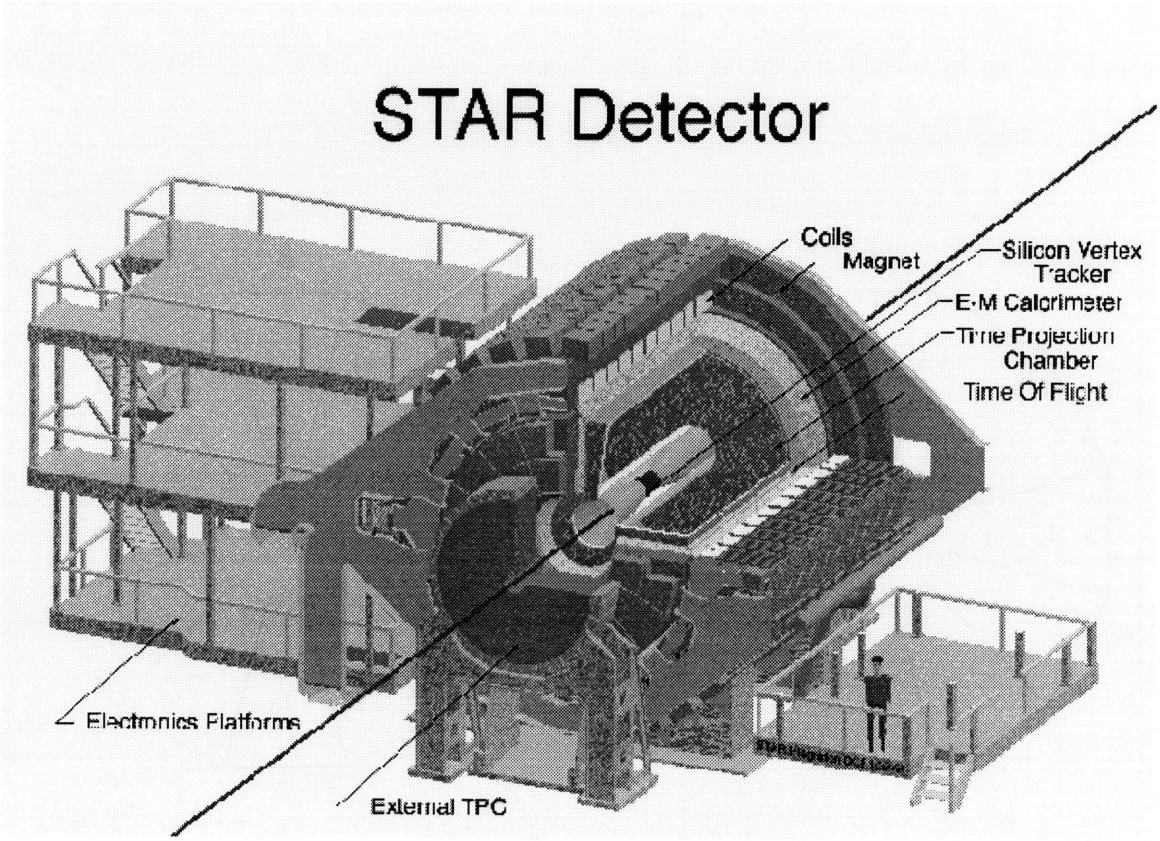

Figure 2-1: The STAR Detector[4]. The particle beams enter the detector along the longitudinal axis. When a collision occurs in the center of the detector, the emitted particles passing through the TPC are subject to the applied magnetic field, causing their trajectories to be helixes.

Because the strength of the applied uniform electric field is known, the rate at which the ions will be drawn towards the detector pads can be computed. Thus, by sampling the detector pads at regular time intervals after a collision, a three-dimensional representation of the event can be reconstructed. Due to the high track density, such a representation is necessary to carry out a meaningful analysis of the event.

2.2 TPC Geometry and Detector Pads

For completeness and to aid the reader in understanding the data produced during an event, the geometry of the TPC as well as the detector pad arrangement are described in this section.

The TPC is roughly cylindrical with a diameter of 1.9 meters and length of 4.2 meters from end to end. The STAR coordinate system sets the z-axis along the beam and the y-axis pointing upwards. The x-axis is defined by a right-handed coordinate system. (See Figure 2-2 below.) The point (0,0,0) is therefore in the center of the TPC.

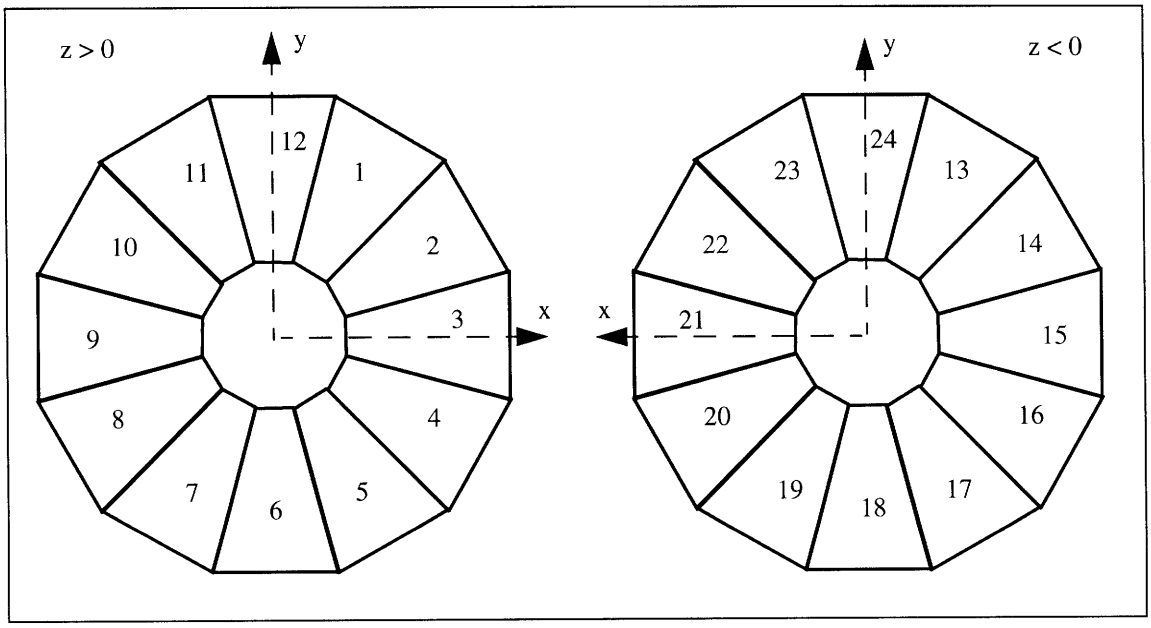

Figure 2-2: TPC Sector Layout and Numbering. The 24 sectors are numbered from 1 to 24. For z>0, the sector numbering follows the face of a clock. If a particle flying parallel to the beam entered sector 2 at one end of the TPC, then it would exit via sector 22 at the other end.

Each sector of the detector contains a matrix of detector pads. Figure 2-3 below illustrates the layout of each sector.

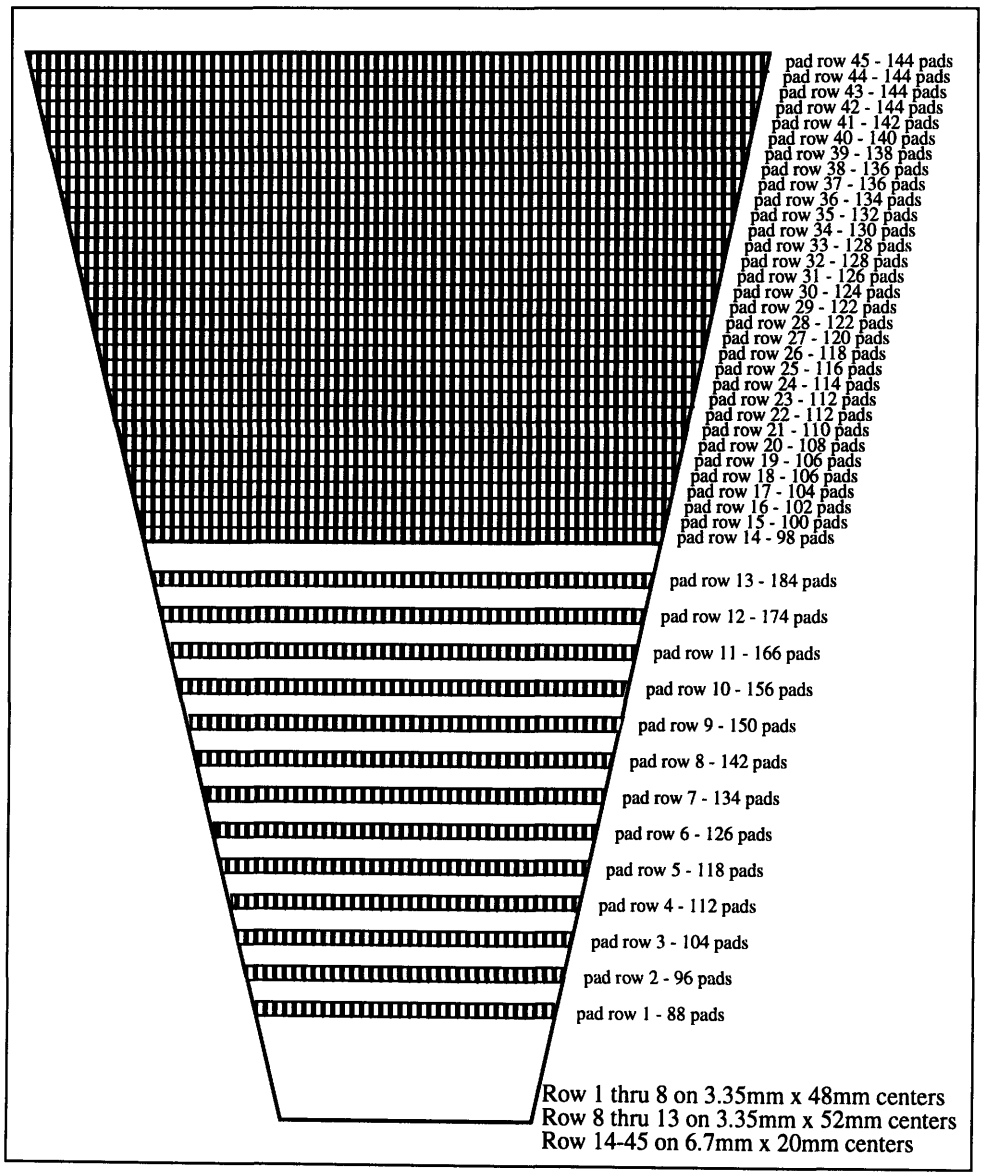

Figure 2-3: Detector Pad Layout. Pad rows are numbered from the innermost row to the outermost row beginning with index 1. In each pad row, the pads are numbered from left to right beginning with index 1. The number of pads in each row is indicated in the diagram. Each sector of the TPC has the same layout.

Unfortunately, due to electronics space constraints, it was not possible to arrange the detector pad rows as densely in the innermost rows. This could pose a problem for tracking since the particle traces

are most dense in the region nearest the event center. Due to the large radial spacing between these rows coupled with the high track density, tracking in this region of the TPC is expected to be difficult.

2.3 Tracking

The discussion of tracking begins by focussing on the characteristics of particle tracks. As a charged particle passes through the inert gas in the detector, it ionizes the particles that happen to be in its path. Because a uniform magnetic field is applied longitudinally along the detector, the charged particles resulting from a collision experience a $q\boldsymbol{v}\times\boldsymbol{B}$ force. This resulting trajectory of each particle is therefore a helix. The parameters of this helix depend on the charge of the particle, the strength of the magnetic field, and the particle's energy (which depends on its mass and velocity). The ultimate goal of tracking is to identify each particle by its path through the detector. This goal is accomplished by calculating the particle's momentum and the sign of its charge from the track's helix parameters.

Another characteristic of particle tracks arises from the fact that the particles typically have a short half-life and often decay into other particles while passing through the detector. The result is a kink in the track (i.e. a discontinuity in the first derivative). The kink can be very pronounced or quite small, depending on the type of decay and the relative momenta and trajectories of the subparticles released in the decay.

As soon as a particle has ionized a gas molecule in the detector, the ionization begins to diffuse to neighboring gas particles. Thus, it is advantageous from a tracking perspective to measure the ionization as quickly as possible. There are two important effects of diffusion: 1) the concentration of ionization spreads out in space and 2) the peak magnitude of the ionization decreases as the ionization spreads. When a front of ionized gas particles reaches the STAR detector pads, it is typically spread across several pads because of diffusion. (See Figure 2-4 below.)

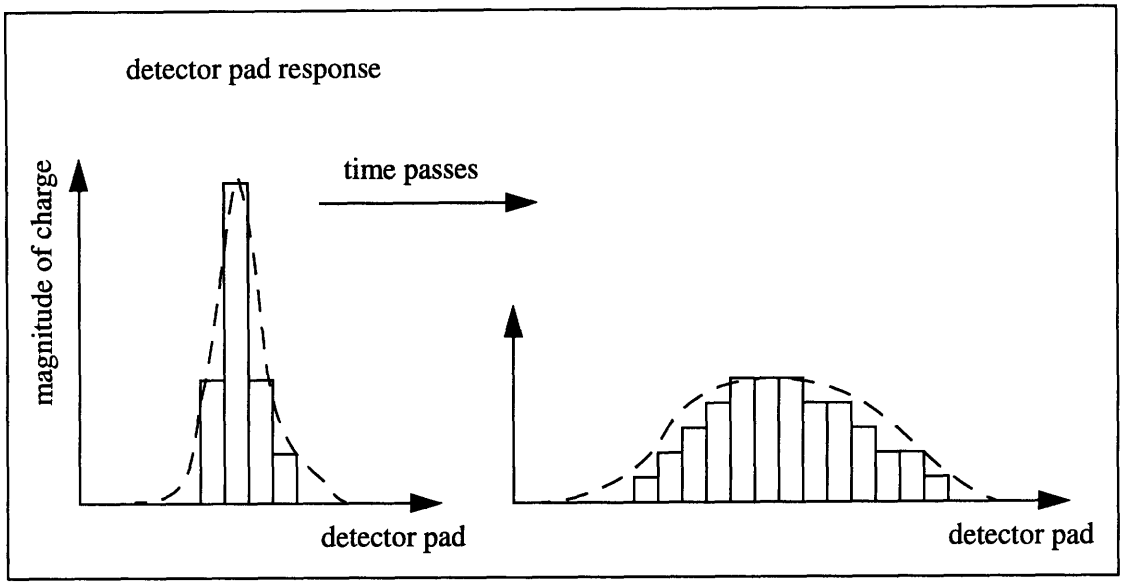

Figure 2-4: Time Dependence of Charge Localization and Detector Pad Response. Once a particle has ionized a region of the TPC gas, the charge immediately begins to diffuse. The amount of spreading is proportional to the elapsed time between the initial gas ionization by the particle and the time when the ions reach the detector pads. In terms of particle traces and the TPC, this translates to the tracks in the middle of the TPC (along the longitudinal axis) being more spread than the ones near either end of the TPC. This is because the ions nearest the detectors are registered sooner than those in the middle of the TPC.

The activity during an event is recorded in the form of digital data representing the coordinates of the emitted particles at sequential instants of time. The primary challenge of tracking algorithms is to identify continuous tracks from this digital data.

The data, when plotted and analyzed at a fine scale, is recognized as traces made up of numerous pixels. Each of these pixels is the ADC value on a pad at a given time. Although one can easily discern the tracks, there are a couple of reasons that the tracking task must be automated. First, the number of tracks from one event in the STAR detector is expected to be on the order of 10,000. Clearly, it would take too long for a human to identify each of these tracks (see Figure 2-5 below). Second, precise measurements of the track properties must be made (such as the particle's momentum), which also is best handled by a computer. And third, in certain circumstances, complicated tasks such as deconvolution must be performed on the data, which again is best suited to a computer.

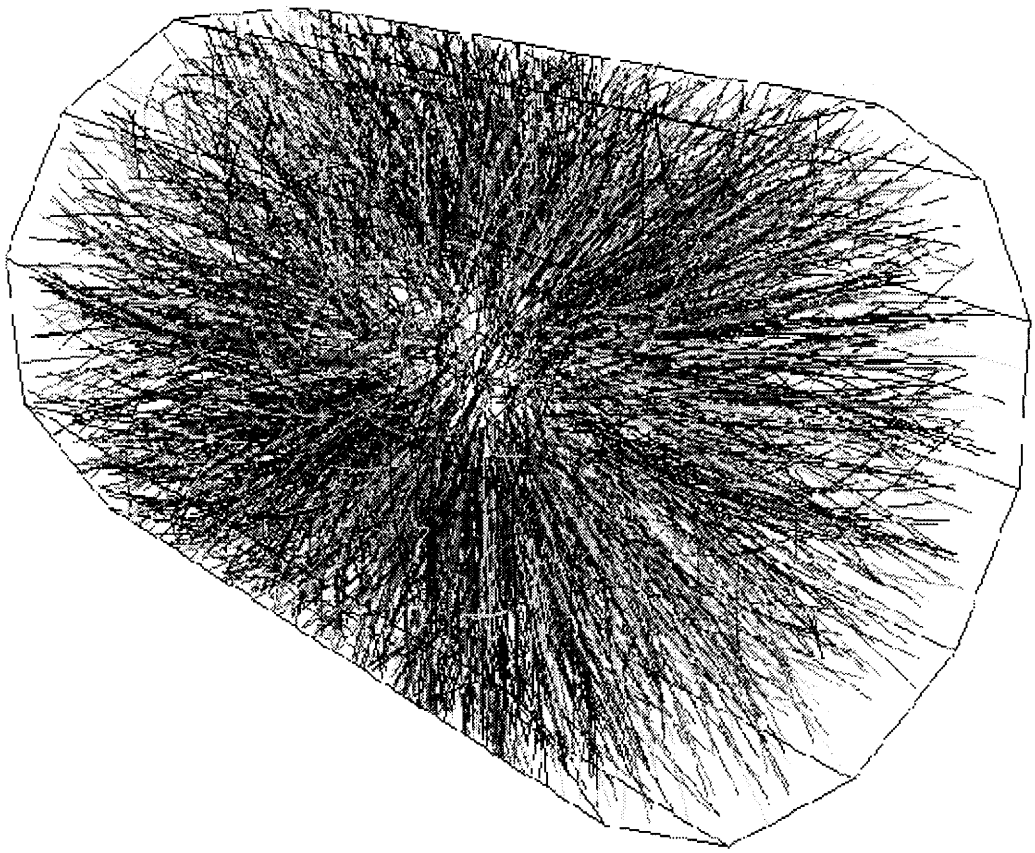

Figure 2-5: Simulation of Full STAR Event. In a typical event, approximately 10000 tracks will be produced. Each track is a helix emanating from the center of the TPC.

There are numerous complications to tracking. For example, because the STAR detector consists of numerous discrete detector pads, the resolution of the event's record depends completely on the placement and density of these detector pads. From a tracking standpoint, high detector pad density and fast sampling times is better. Nevertheless, there are practical (technical, physical, and financial) limitations in effect which bound the electronics used in the detector.

Besides the resolution limitation, several other "real-world" effects complicate tracking. One effect is noise. That is, a detector pad may avalanche due to random fluctuations in the ionized gas or due to other noise in the environment of the detector. Also, detector pads may be defective or simply cease to function, thereby leaving a gap in a track.

Other complications involve the characteristics of the tracks themselves. Particle decays, which were mentioned previously, are troublesome because of the kink in the track that can occur at any point along the track without warning, thereby potentially confusing the tracker.

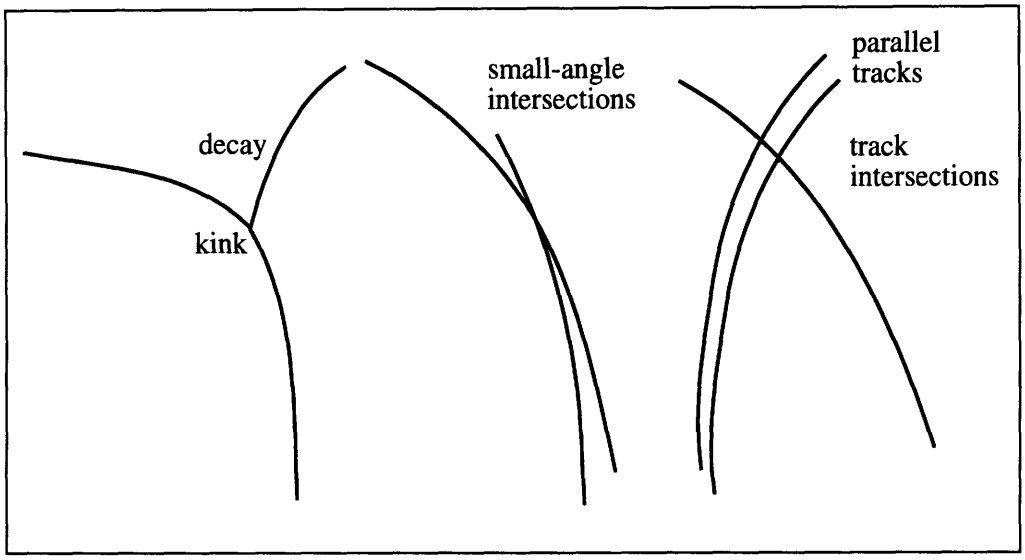

Figure 2-6: Sources of Tracking Complications. Particle decays, track intersections, and parallel tracks make it difficult to follow a track especially when the resolution of the data is limited.

When multiple tracks are present, there are additional complications to consider. One case is when two tracks intersect (which appears very similar to a particle decay). Second, two tracks which lie close together and do not intersect or intersect at a small angle may appear as one single track along some interval of the overall track. Charge delocalization effects are particularly troublesome for tracking these tracks. In this case, the charges from the two tracks begin to overlap and therefore cannot be distinguished by the detector pads.

It should be apparent that, in order for a tracking algorithm to attain a high tracking efficiency, all of these complications must be taken into account.

Chapter 3

The Traditional Approach: Sequential Centroid Tracking

3.1 Hit-Finding

The traditional approach to tracking is carried out in two steps. First, the entire data set is processed and clusters of pixels representing *hits* are identified. This is done by finding groups of pixels which appear in contiguous positions in pad and TDC space. Next, beginning with one such hit on the outer portion of the TPC, the algorithm searches for three correlated hits to form the beginning of a segment. Once this is found, the segment is extended as far as it can go by adding successive centroids.

This method has been shown to work quite well for low-multiplicity events. And, until recently, only low-multiplicity events have been produced. The high-multiplicity events that will be generated at RHIC, however, could be troublesome for this algorithm. It has been shown that the efficiency (roughly defined as the percentage of particle tracks correctly identified) of the sequential centroid tracker falls off significantly with increased multiplicity.

The problem with this approach has to do with the serial nature of the tracking. Once a hit is used by a track, it is removed from the data set and is not available for use by another track. This is problematic when two tracks intersect or one track decays. In low multiplicity events, such intersections and decays are few so the tracking algorithm is not affected significantly. In high-multiplicity events, however, these are quite common (estimated to account for 10% of the clusters).

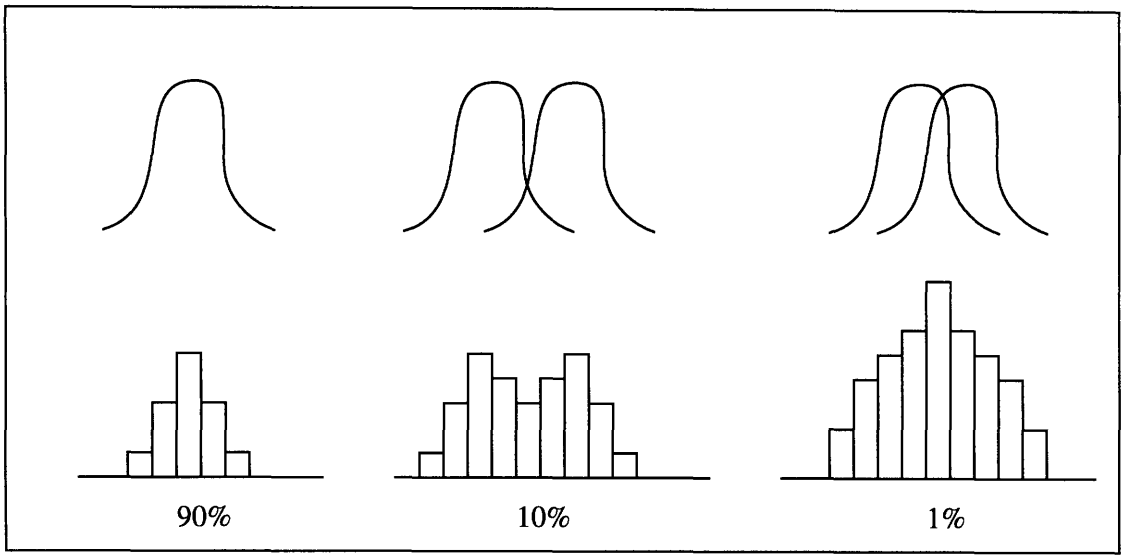

Figure 3-1: Overlapping Tracks and Detector Pad Response. Different degrees of track overlap produce different hit shapes. Below each configuration is the estimated percentage of that type of hit in a typical event recorded by the STAR detector.

Although the hit-based serial tracker performs well on the single track hits (90%), it is weak on the other 10% of the hits, which are formed by overlapping tracks. This 10% is crucial for attaining the needed efficiency for identifying the quark-gluon plasma and provides the incentive to develop a tracker better suited to high-multiplicity events.

It should be noted that the problem is actually not due to the multiplicity of the event. Rather, it is a result of higher multiplicity while the volume of the detector and the detector resolution have remained relatively constant. Therefore, increased multiplicity implies greater track density, and higher track density implies that more of the tracks will overlap. In a low-multiplicity event, for comparison, the percentages of the overlapping/intersecting tracks would be closer to 1-2%.

Chapter 4

A New Approach

4.1 Pixel-based Parallel Tracking

A new tracker has been designed with the goal of maintaining high efficiency at high track multiplicities. This tracker includes mechanisms to deal with the aforementioned complications involved with tracking in dense track conditions. This chapter discusses the general approach behind this new tracker.

The main differences between the traditional and the proposed trackers are found in the track extension phase. The flow diagram below (Figure 4-1) summarizes the main features of the proposed tracker, each of which will be discussed in more detail. The cluster finder and proto-track finder modules are quite similar to those used in the traditional tracker, since there is little improvement to be made in these modules as far as tracking efficiency is concerned.

The parallel tracker proceeds in spherical shells beginning from the outermost portions of the TPC toward the event center. Each iteration of the tracker loop corresponds to one shell of pixel data. The track extender module only operates on one shell of pixels at a time. However, the cluster finder and proto-track finder modules may operate on multiple shells of pixels, since the shells may be too small for these modules to function properly on only one at a time. Nevertheless, this is a detail which does not affect the general explanation of the proposed tracking algorithm.

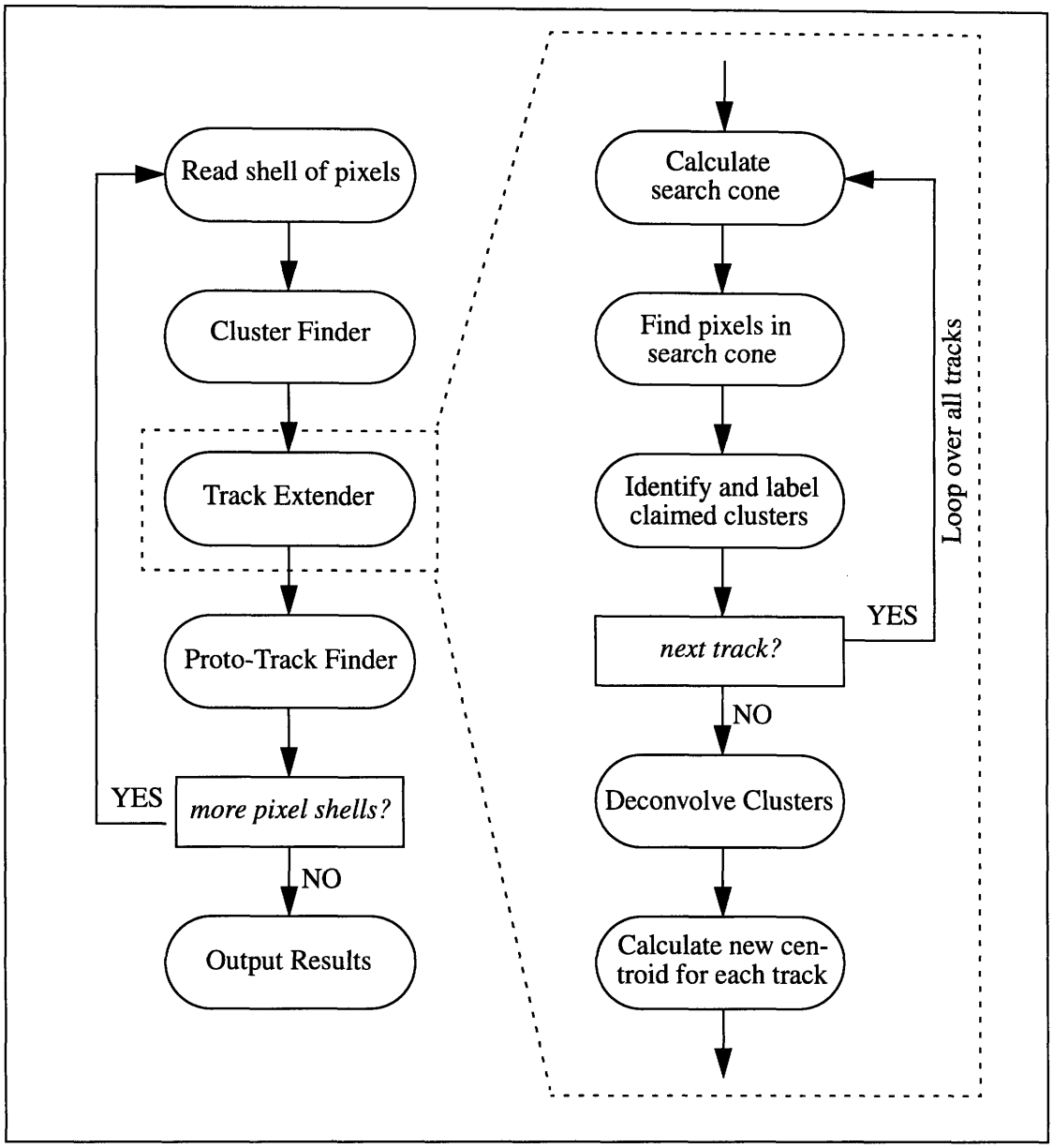

Figure 4-1: Flow Diagram of Tracker. The track extender module of the tracker is shown in detail at the right.

Once provided with a shell of pixels, our algorithm first groups the pixels into clusters. This step is performed using basically the same algorithm as the traditional tracker. Next, from these clusters, the program identifies proto-tracks. Each proto-track consists of three clusters, which have passed tests for their suitability as a proto-track.

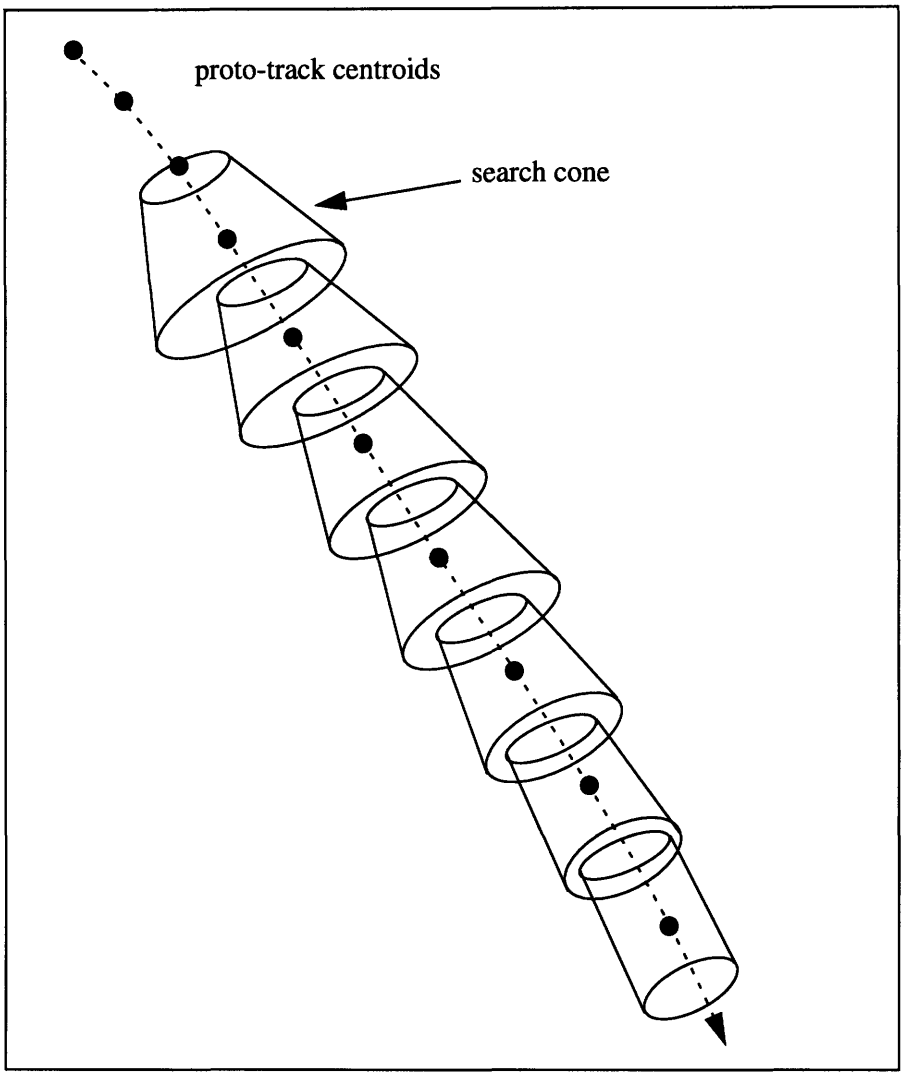

Figure 4-2: Track Formation and Extension. Beginning with the proto-track, search cones are used to locate pixels which should be claimed by a track. Using these pixels, a new centroid is calculated for the track and added to it. Notice that the search cone is tightened as the track progresses.

In the proposed tracker, a track is represented as a sequence of centroids. These centroids are used to steer the track for the track extension phase. Based on the trajectory of the most recent centroids added to the track, a *search cone* is calculated for each track. This search cone stretches from one shell boundary to the next. Intuitively, using a search cone means that the track extender is looking for pixels in *the right place*. Such an approach also allows the search space to be smaller in general, thereby improving the accuracy of the tracking.

Searching in pixel space, all pixels lying within each search cone are extracted. Next, a list of clusters associated with these pixels is generated for each track. The segment extender then calculates the amount of overlap of the search cone onto the clusters. If the overlap is greater than a specified amount, the cluster is considered to be claimed by the track and it is labelled with the track's ID (see Figure 4-3).

Once this process has been completed for all tracks, each cluster will have a list of the tracks that are claiming it. This list may contain zero, one, or many entries, corresponding to the cases when the cluster is not claimed by any track, the cluster is claimed by one track, or the cluster is claimed by many tracks, respectively. Once it is known how many and which tracks claim a given set of pixels, a deconvolution of the clusters is carried out based on this information to decide how to allocate the clusters (or fractions thereof) to each track.

The next step is to extend all of the segments simultaneously, which is done before proceeding to the next spherical shell. This is accomplished by calculating the centroid of the deconvolved clusters and adding it to the track. As the track grows in length, the search cone for that track can be tightened since it is more likely that a track has been successfully identified (see Figure 4-2). This is particularly beneficial when tracks lie close to one another or intersect at small angles.

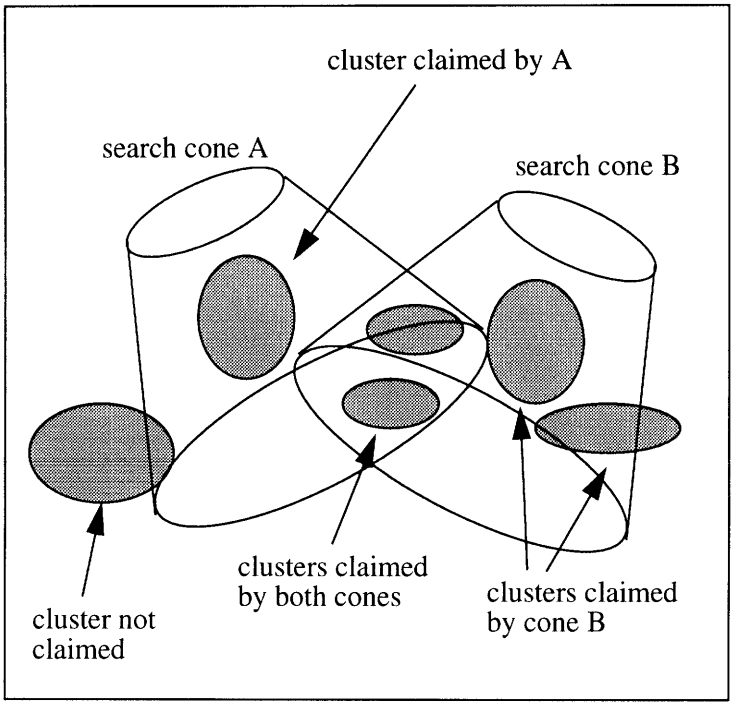

Figure 4-3: Cluster Claiming and Overlapping Search Cones. This figure illustrates the various possibilities of cones claiming clusters.

When the algorithm working on a particular track becomes confused (possibly in the case of a track intersection or decay) or can not find any pixels in the search space, the track is terminated. Although a track may be prematurely terminated, the algorithm attempts to find new proto-tracks during each iteration of the tracker. This approach does lead to more segmented tracks, but the goal is to not sacrifice correctness and efficiency. A segment joining method which complements the tracking algorithm will be discussed in Chapter 9.

Note that the tracker extender module is given precedence over the proto-track finder in the flow diagram. The tracker is arranged in this way so the track extender has first pick of the clusters in the shell. Once the clusters claimed by tracks have been removed from the eligible clusters, the proto-track finder can use the remaining ones to begin new tracks.

Once all the shell of pixels have been read from the data file, the tracker writes the resulting track information to an output file. This information includes the centroids of each track labelled with the track's ID.

Chapter 5

Computational Considerations

5.1 Running Time and Space

From a computer science standpoint, an interesting issue that the proposed tracking algorithm raises is the amount of memory and computation time required to analyze each event. Each event will produce approximately five million pixels of data forming approximately 10,000 tracks.

Although workstation memory and processing power is becoming more affordable, there are practical limitations that must be considered. This means that the algorithm must be modified in various ways to account for and optimize for these limitations.

From the beginning, it was realized that limitations on memory would preclude the possibility of reading in all pixels at once (i.e. into RAM). And because the tracking algorithm needs ready access to all of the pixels currently being searched, it is important to avoid using swap space. Swap space is most useful when there is a portion of data that is not being used for some time, which is not the case here. Since retrieving data from swap space is on the order of 1000 times slower than retrieving it directly from RAM, the performance of the algorithm would suffer dramatically if swap space were depended upon. With this understanding, the use of spherical shells is seen to have the added benefit of breaking up the data into manageable chunks. This way, the algorithm can read in and process the pixel data as needed.

The original prototype of the new tracking algorithm required between 10 and 20 hours to analyze a single event. After profiling the code and diagnosing the relative amounts of time spent in various portions of the algorithm, the bottlenecks were identified. This allowed us to focus optimization efforts particular parts of the program. The current prototype now takes less than one hour to analyze an event.

The running time efficiency of the tracker is due mostly to the *octant tree* data structure, which will be explained in detail in the next chapter. Basically, the octant tree permits extremely efficient searches through the pixel data, so all pixels in a region of space in the TPC can be extracted quickly.

Chapter 6

Implementation

6.1 Introduction

The majority of work for this project was done in the C programming language. C was chosen primarily for its flexibility, standardization, and efficiency. No specialized libraries were required for the tracker program, so the code will be very portable across platforms.

The implementation of the tracker program is split into three main modules: main tracking routine, cluster finder, proto-track finder, and track extender. Each of these modules will be explained fully in this chapter along with the data structures and parameters on which they depend.

For the pseudocode in this thesis, the following conventions are used:

- *"a←b"* indicates that a value *b* has been assigned to variable *a*.
- *"a->b"* indicates a reference to an element *b* of a data structure *a*.
- "=" indicates a test for equality.

6.2 Data Structures

To understand the implementation of the tracker, one should first take time to familiarize himself with the data structures used in the program. Below are tables of these structures containing the fields in each structure along with a short explanation of its function. In the tables, an asterisk (*) indicates a pointer. The indicated field names are the ones actually used in the source code, so the reader can easily refer to them when reviewing the code.

Table 6.1: PIXEL Data Structure

Field Name (Data Type)	Explanation
`true_tid` (integer)	Holds the track ID assigned by the event simulator (used for analyzing the performance of the tracker on simulation data)
`found_tid` (integer)	When the pixel is assigned to a track, this variable is set to that track's ID number. (Initially set to 0.)
`rho` (double-precision float)	Holds the distance of the pixel from the center of the TPC. rho, of course, can easily be calculated from the coordinates of the pixel, but there are many tight loops where `rho` is needed, so it is better for efficiency reasons to calculate and store `rho` when the pixel is created.
`x`, `y`, `z` (double-precision float)	Cartesian coordinates of the pixel calculated from the sector, row, pad, and tdc of the pixel (in units of centimeters).
`sector`, `row`, `pad`, `tdc` (integer)	Hold the position of the pixel as given in the original TPC data. These are needed by the cluster finder to determine whether pixels lie in contiguous positions in the TPC.
`adc` (short integer)	Holds the ADC value of the pixel, which is the measured magnitude of the avalanche at the detector pad for that pixel.
`cluster` (*CLUSTER)	Points to the cluster which has claimed this pixel. (Initially set to NULL.)

Table 6.2: CLUSTER Data Structure

Field Name (Data Type)	Explanation
`claimed_by_track` (boolean)	Indicates whether this cluster has been claimed by a track yet or not.
`track_adc_weight` (integer)	Holds the sum of the ADC values of the pixels from that cluster which lie in a track's search cone. The ratio of this to the cluster's total ADC weight is the parameter which determines whether a track can claim the entire cluster or not.
`total_adc_weight` (integer)	Holds the sum of the ADC values for all the pixels comprising this cluster.
`centroid` (*CENTROID)	A pointer to the centroid which holds the calculated centroid of the cluster.
`pixel_list` (*PIXEL_LIST)	A pointer to a list of pixels which comprise the cluster.
`claiming_tracks` (*TRACK_LIST)	A pointer to a list of tracks which have claimed the cluster. This information is useful for deconvolving the cluster based on how many tracks claim the cluster.

Table 6.3: CENTROID Data Structure

Field Name (Data Type)	Explanation
`rho` (double-precision float)	Distance of the centroid from the center of the TPC.
`x, y, z` (double-precision float)	Cartesian coordinates for the location of the centroid.

Table 6.4: TRACK Data Structure

Field Name (Data Type)	Explanation
found_tid (integer)	Holds the unique "found" track ID number for this track.
num_shells_wo_new_centroid (short integer)	Keeps track of the number of shells that the track has gone without claiming any new clusters (i.e. not been extended). This is used to decide when the track should be terminated.
centroid_list (*CENTROID_LIST)	A pointer to a list of centroids which represents the track. The first three centroids in this list are the track's proto segment. As the track is extended toward the interaction point, centroids are appended to this list.
claimed_clusters (*CLUSTER_LIST)	A pointer to a list of clusters that the track has found in its search cone during the track extension phase.

Table 6.5: SHELL Data Structure

Field Name (Data Type)	Explanation
rho_min, rho_max (double-precision float)	Specify the range of distance from the center of the TPC for the pixels in this spherical shell. The shell thickness = rho_max - rho_min.
pixel_list (*PIXEL_LIST)	A pointer to the list of all pixels in this spherical shell.
cluster_list (*CLUSTER_LIST)	A pointer to the list of pixel clusters associated with this shell. Note that this may include clusters which point to pixels from other shells.
next_shell (*SHELL)	A pointer to the next concentric spherical shell inward toward the center of the TPC.
next_shell_in_queue (*SHELL)	A pointer to the next shell in the shell queue (which may be different from next_shell).

Table 6.6: SHELL_QUEUE Data Structure

Field Name (Data Type)	Explanation
num_shells (short integer)	Indicates the total number of shells in the queue.
first_shell (*SHELL)	A pointer to the first shell in the queue.
last_shell (*SHELL)	A pointer to the last shell in the queue.

Table 6.7: OCTANT_TREE_NODE Data Structure

Field Name (Data Type)	Explanation
pixel (*PIXEL)	A pointer to a pixel if the node is a leaf, NULL otherwise.
x_min, x_max, y_min, y_max, z_min, z_max (double-precision float)	Ranges for the octant tree.
oct1, oct2, oct3, oct4, oct5, oct6, oct7, oct8 (*OCTANT_TREE_NODE)	Pointers to sub octants (NULL if there are no pixels lie in that sub octant).

Table 6.8: CLUSTER_TREE_NODE Data Structure

Field Name (Data Type)	Explanation
`cluster` (*CLUSTER)	A pointer to a cluster if the node is a leaf, NULL otherwise.
`x_min`, `x_max`, `y_min`, `y_max`, `z_min`, `z_max` (double-precision float)	Ranges for the octant tree.
`oct1`, `oct2`, `oct3`, `oct4`, `oct5`, `oct6`, `oct7`, `oct8` (*OCTANT_TREE_NODE)	Pointers to sub octants (NULL if there are no clusters lie in that sub octant).

Also included in the tracker's data structures are list structures of many of the structures listed above. For example, there is a pixel list which can hold multiple pixels. These data structures consist simply of an array of the data elements and an integer variable indicating the number of elements currently in the array. This index allows simple management of the array, especially when appending an additional element to the array (which is a common action in the tracker program).

6.3 Octant Tree

The octant tree data structure is used throughout the tracking program and plays a crucial role in the tracker's computational efficiency. Since the data used by the tracker is expressed primarily in three-dimensional cartesian coordinates and because there is a need to access regions in that space, a data structure had to be adopted which would allow this operation to be conducted efficiently. For instance, in the track extension phase, the parameters of the search cone for a track are known. The straight-forward approach to extracting the pixels in this search volume would be to check each pixel in the spherical shell to see if that pixel lies within the search cone. However, since this operation must be conducted so often, the $\Theta(n)$ running time of this approach quickly becomes a problem.

An octant tree divides a region of space (defined by the minimum and maximum values for pixels in each coordinate axis) into eight octants. This is done by dividing each of the dimensions by two. The numbering for the octants is unimportant for understanding the octant tree, although they are numbered for reference in the implementation.

Figure 6-1 below depicts a two-dimensional representation of how the octant tree works.

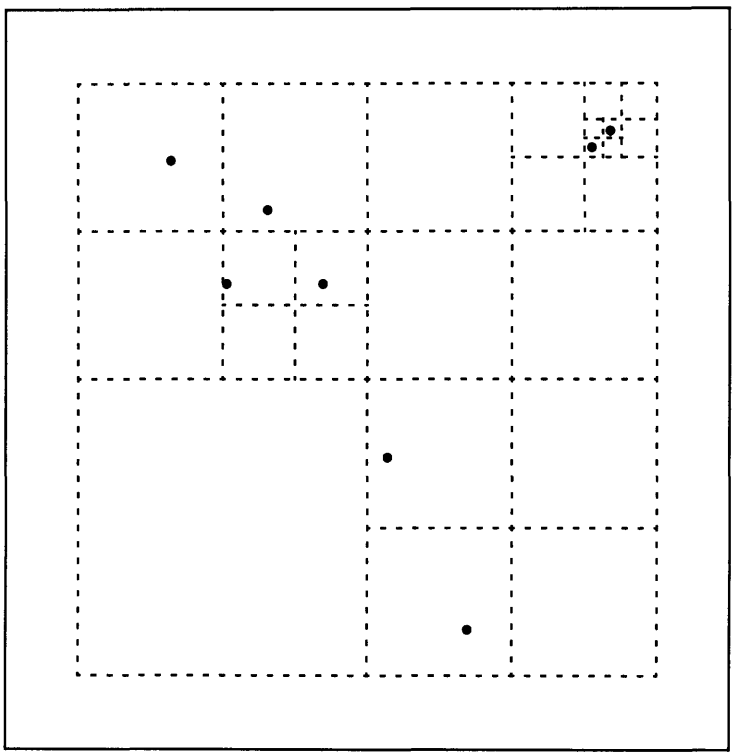

Figure 6-1: A Two-Dimensional Example of Octant Tree Partitioning. Each point in the diagram represents an individual pixel. Note that there is at most one pixel in each sub-octant. This invariant is maintained by creating new sub octants as needed when adding additional pixels.

The invariant of the octant tree is that there can be no more than one pixel at any leaf of the tree. Each pair of intersecting lines represents an internal node of the octant tree and indicates that there is more than one pixel in the subtree rooted at that node. The octant tree turns out to be an excellent representation for the sparse pixel arrangement in the TPC data. As shown in the diagram above, regions of space that have no pixels in them require no extra octant nodes.

Before an octant tree can be used for pixel extraction, it must first be constructed. To construct an octant tree, the first step is to create a root node. All sub-octants of this root node are initially empty. When the first pixel is to be added, its coordinates are compared to the boundaries of each sub-octant of the root node. Once the correct sub-octant has been identified, a new node is created to hold the pixel. The corresponding sub-octant pointer is assigned to this new node in order to connect it to the tree.

Additional pixels are added to the tree one at a time in a similar manner. If a pixel is added which falls into the same sub-octant as a previously added pixel, then the following steps are carried out:

(1) *temp_pixel* ← *node–>pixel*
(2) *node–>pixel* ← *NULL*
(3) *Insert_pixel (node, temp_pixel)*
(4) *Insert_pixel (node, new_pixel)*

where *Insert_pixel* is the pixel insertion routine.

If the new pixel still falls in the same octant as the original pixel (i.e. the two pixels are near each other in space), additional interior nodes are added until the pixels fall into different octants.

The addition of each pixel to the octant tree takes $\Theta(\log_8 n) + \Theta(1) = \Theta(\log_8 n)$ time to complete. That is $\Theta(\log_8 n)$ time to find the insertion point for the pixel and $\Theta(1)$ time to actually insert it. Thus, creating a full octant tree of n pixels will take $n*\Theta(\log_8 n) = \Theta(n*\log_8 n)$ time to complete. The tree-building time is acceptable since numerous tree searches are conducted on an octant tree once is has been created. Once the octant tree is constructed, pixels can be extracted within a rectangular box specified by the minimum and maximum along each coordinate axis. Note that using only the minimum and maximum values of each coordinate means that the box will be square with the coordinate axes.

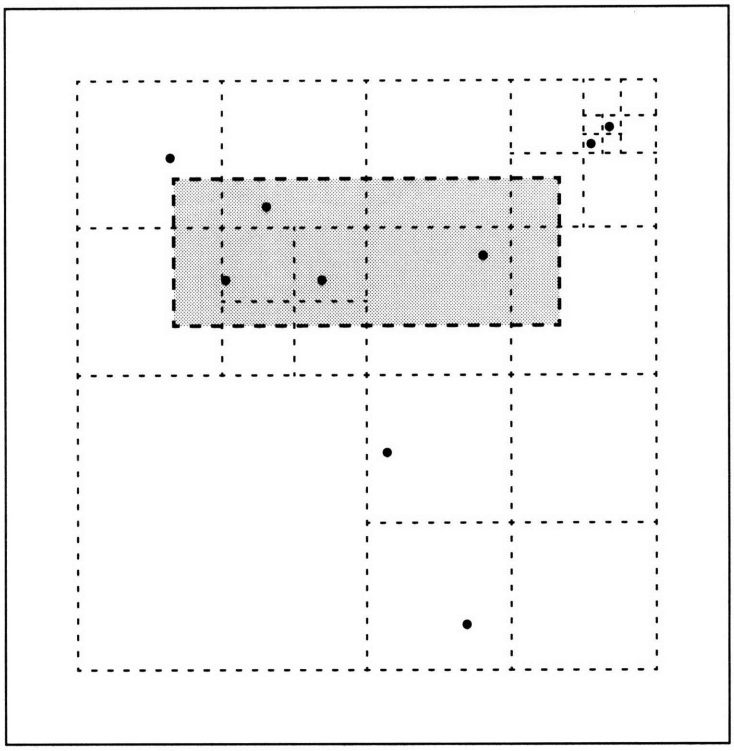

Figure 6-2: Pixel Extraction Using an Octant Tree. The figure depicts a two-dimensional view of an octant tree. The desired region to search is in gray. The dashed rectangle in the diagram is an example of a search space that may be needed. Following the rules explained next, the pixel extraction procedure will find the four pixels enclosed by the bounding box.

There are five cases that can occur regarding the extraction of pixels from the octant tree:

1) The node is a leaf, in which case the pixel contained at that node should be tested for inclusion to the extracted pixel list.

2) The desired box is completely outside of an octant. In this case, the procedure returns, halting the recursion along that branch of the octant tree.

3) The desired box is completely within an octant. Therefore, the procedure recurses on each of that octant's sub-octants.

4) The desired box completely encloses an octant. In this case, all pixels in this octant (including the ones in the subtree rooted at that octant) can be added to the extracted pixel list without further testing.

5) The desired box partially overlaps an octant. Therefore, the procedure recurses on each of that octant's sub-octants.

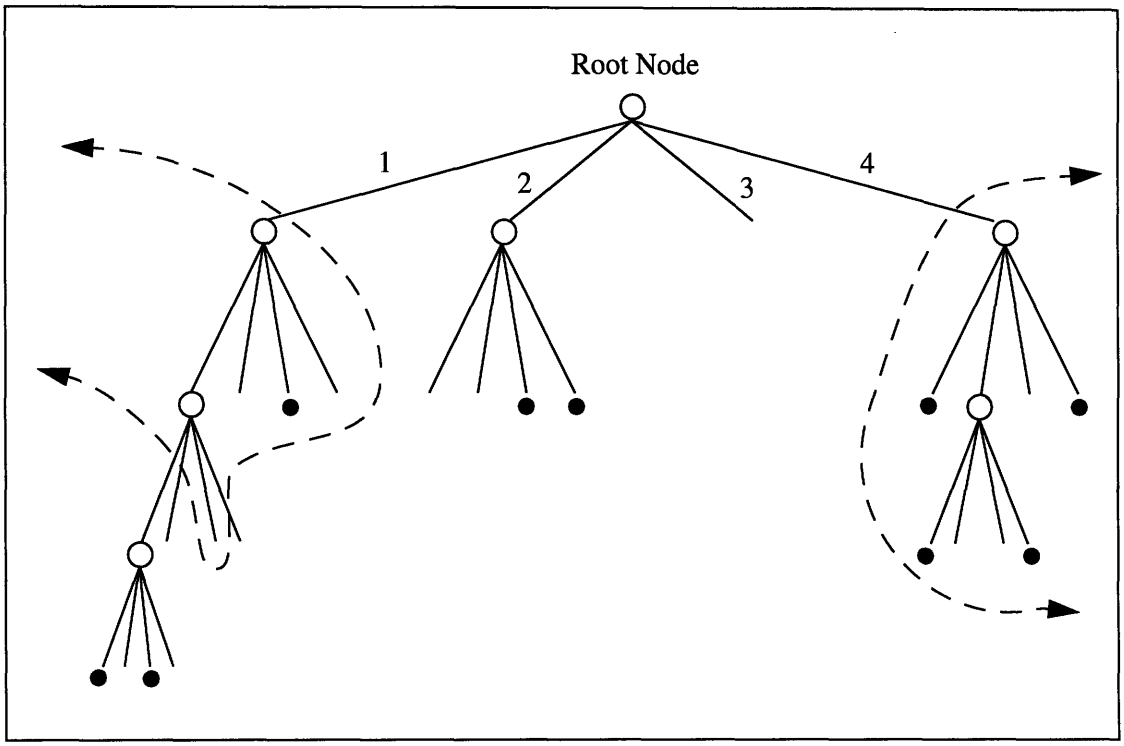

Figure 6-3: Tree Structure of Two-Dimensional Version of the Octant Tree. This diagram shows the internal representation of the (two-dimensional) octant tree shown in Figure 6-2. The dashed line encloses the nodes which are searched by the pixel extraction algorithm. Not all pixels included in the search will be in the search region, but they still must be tested.

Assuming a well-balanced tree (which is a reasonable assumption considering the fairly even distribution pixels throughout the TPC), the amount of time it takes to extract p pixels from an octant tree containing a total of n pixels is $\Theta(p \log_8 n)$. This is because the tree has a depth of $\log_8 n$, and since p pixels are extracted, it takes $\Theta(1)$ time to add each one to the list of extracted pixels. Thus it can be

seen that the octant tree makes an excellent structure to represent the pixel data - both because of the ability to extract pixels in any rectangular region of space and the efficiency with this operation can be carried out.

6.4 Main Tracker Routine

The primary functions of the routine that controls the tracker are to read in data from the data file and to invoke the cluster finder, track extender, and proto-track finder. The data file is sorted by ρ in decreasing order. This allows the tracker to easily read in a shell of pixels (specified by `rho_min` and `rho_max`). Given `rho_min`, the data-reading procedure simply reads from the current position in the file until the first pixel which has a value of ρ less than `rho_min`. These pixels are all of the pixels in the next spherical shell.

The tracker proceeds in concentric spherical shells starting with those pixels farthest from the center of the TPC. The intuition applied here is that the track density decreases with distance from the interaction center. Tracks therefore overlap and intersect less frequently, so they should be more easily identified.

The shell thickness is decided upon by considering the amount that each track should be extended during each round. However, other routines may require more than one shell of pixels to effectively do their work. For this reason, shell queues were introduced.

When a new shell is read in, it is first added to the cluster finder shell queue. Shell queues are FIFO queues, so each subsequent shell is added to the bottom of the shell queue. The cluster finder always operates ahead of the track extender and proto-track finder because both of these routines operate on clusters. The number of shells in the cluster finder shell queue is a parameter which can be adjusted by the experimenter.

The overall order of the sub-procedures for the tracking program is important. The cluster finder comes first, as explained previously. Next, the track extender is executed in order to extend any exist-

ing tracks through the next spherical shell, which gives the track extender first chance to claim the pixel clusters in the next shell. The remaining clusters are left for the proto-track finder to use in attempts to create new tracks.

When a new shell is added to the cluster finder shell queue which causes the number of shells in the queue to be greater than the maximum number of shells, the first shell in the queue is popped off the top of the queue. This shell now becomes the shell that the track extender operates on. It is also added to the proto-track finder's shell queue (which also has a maximum queue size associated with it).

When a shell is added to the proto-track finder's shell queue which causes the number of shells in the queue to exceed the specified maximum, the first shell in the queue is popped off. At this point, this shell of pixels has been completely processed by all parts of the tracker so the pixel information contained in it is written to the output data file. After that, the entire shell of pixels is freed from memory.

When the end of the input data file is reached, the main routine will continue to pop shells of the top of the cluster finder queue until it is empty. As before, these are passed on to the track extender and the proto-track finder queue. Once the last queue has been processed by the proto-track finder, the main loop exits and the remaining shell data in the proto-track finder is written to the output file. Finally, the entire program exists.

The diagram below shows the dynamics of the cluster finder shell queue, track extender shell, and proto-track extender shell queue as shells are read in and passed from one queue to another. Shell boundaries here are represented as straight horizontal lines.

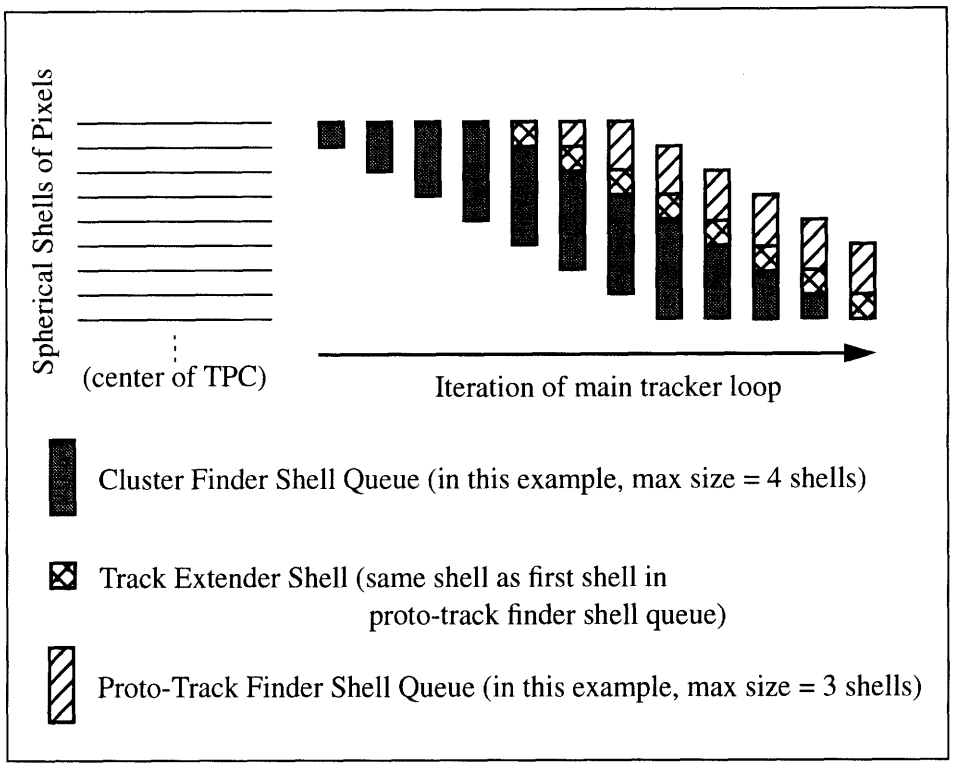

Figure 6-4: Shell Queue Dynamics for Main Tracking Routine. The cluster finder shell queue grows until it reaches its maximum size, at which point it pops off one shell for each shell that is added to it, thereby maintaining the maximum size. The removed shell is first used as the track extender shell and is also added to the proto-track finder shell queue. The proto-track finder shell queue also has a maximum size. Shells popped from this queue are no longer needed and the memory is freed. As the last few shells are processed, the cluster finder shell queue shrinks. And finally, the track extender processes the last shell and the tracker is finished.

6.5 Cluster Finder

The cluster finder's job is to find pixels that are nearby and group them into a *cluster*. Why group pixels in this way? It is known that pixels near each other in space are most likely from the same track, therefore they should be processed in such a way that acknowledges and exploits this association.

Recall that the ADC value of a pixel is proportional to the magnitude of charge on the TPC gas particles at that location. An ADC value can be in the range of 0-1023, but pixels with ADC values less than 4 are not even included in the input data file since they mainly represent noise.

The cluster finder generally operates on more than one shell of pixels at a time. This is done for two reasons. The first is because many clusters may lie on the boundary between two shells. Secondly, a single cluster may extend across an entire pixel shell. This, of course, depends on the shell size.

Below is the pseudocode for the top-level cluster finder procedure:

(1) *Create octant tree of all pixels in cluster finder shell queue*
(2) *Create a master list of all pixels in cluster finder shell queue that have not yet been clustered*
(3) *Sort the master pixel list in decreasing order by ADC value*
(4) *For each pixel in the master pixel list*
(5) *Check if pixel has already been clustered*
(6) *Check if pixel's bounding box has a corner inside the innermost sphere*
(7) *Extract pixels from the octant tree which lie in the bounding box*
(8) *Filter these pixels for those pixels in the same row and sector as the current pixel*
(9) *Generate a new cluster using the current pixel as the local maximum*
(10) *Decide which shell should claim the cluster and add it to that shell's cluster list*

After the octant tree has been built, the next operation is to concatenate the sets of pixels from the cluster finder shell queue. Before each pixel is added to the list, it is first checked to see whether or not it has already been clustered. If the pixel has been clustered, then there is no need to consider it further.

The next step is to sort the entire list of unclustered pixels in decreasing order based on their ADC values. The reason for this will become apparent as the cluster finder procedure is described further. The list of sorted pixels is traversed one pixel at a time. The pixel is first tested to see if it has been clustered yet.

Next, a bounding box around the base pixel is formed using a user-adjustable parameter (E) defining the edge length of the search cube around the pixel. A cube is used because the cluster is assumed to be centered on the peak pixel. The bounding box is calculated from the pixel's cartesian coordinates (x,y,z):

$$x_{min} = x - E/2,\ x_{max} = x + E/2$$

$$y_{min} = y - E/2,\ y_{max} = y + E/2$$

$z_{min} = z - E/2$, $z_{max} = z + E/2$

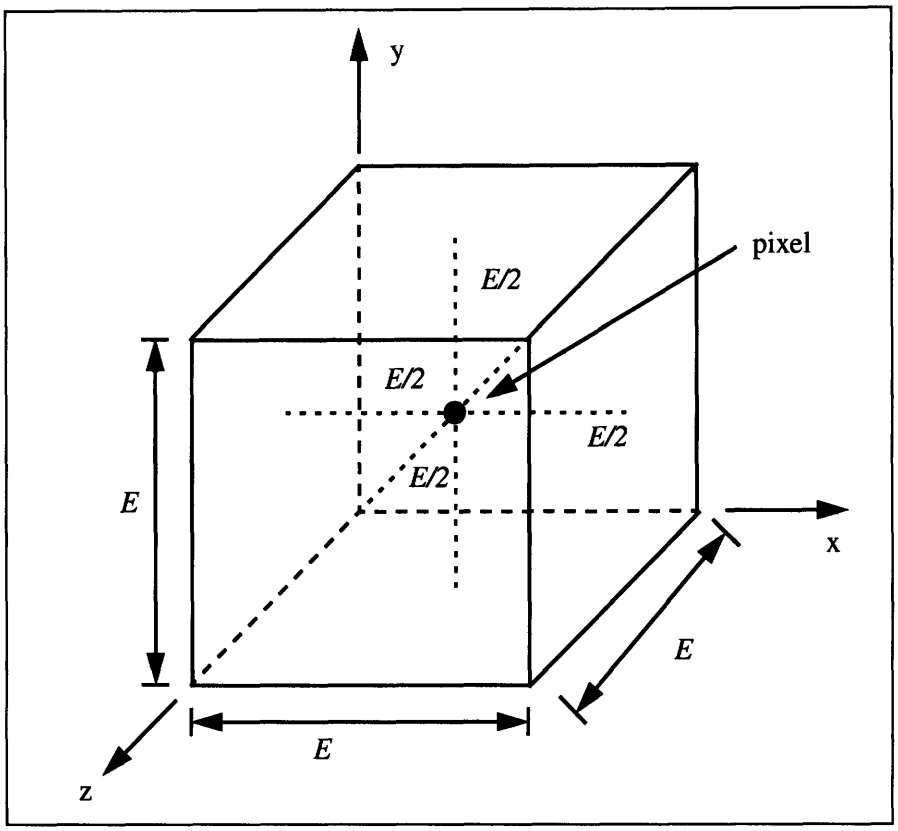

Figure 6-5: Search Cube for Cluster Finding. *E* represents the edge length along each coordinate axis.

Using the octant tree, all pixels within the bounding box are extracted. Then, these pixels are filtered to find all of the pixels which lie in the same sector and row as the base pixel. This is done due to the many irregularities between padrows and sector orientations in the TPC detector pad arrays. Once filtered, a list of pixels is obtained which represents a two-dimensional matrix of pixels differing only in pad number and TDC. The size of the matrix is determined by the range of TDC values and pad numbers of the extracted and filtered pixels. Note that many matrix elements may be empty if there is no pixel there.

Starting at the peak pixel, the cluster finder launches a recursive search of that pixel's neighbors. This search expands, checking other neighboring pixels. The search checks each neighboring pixel to see if its ADC value is less than its own. If the neighbor's ADC value is greater, then that neighbor is not added to the cluster and the search terminates along that path. Or, if there is no pixel in a neighboring position, then the search terminates along that path.

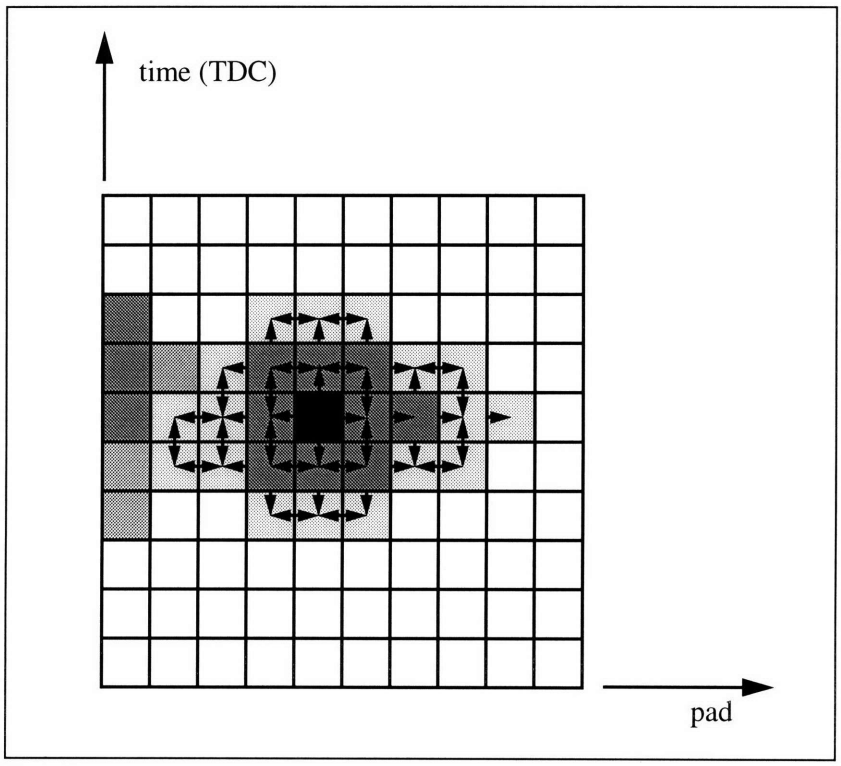

Figure 6-6: Cluster Finder Progression. In this figure, darker shading indicates a higher ADC value for the pixel at that matrix element. The arrows indicate the direction that the cluster-finding routine follows in adding neighboring pixels. Note that the algorithm proceeds outwards from the base pixel to add pixels with lower ADC values.

The pseudocode for the recursive algorithm to find the pixels in a cluster follows. Note: *direction* is one of {+TIME, -TIME, +PAD, -PAD}, corresponding to the four possible directions in TDC-pad space. The algorithm is initiated by four calls to *Check-Neighbor* - one for each of the base pixel's four neighbors.

```
(1)     Check-Neighbor (+TIME, t_pixel + 1)
(2)     Check-Neighbor (-TIME, t_pixel - 1)
(3)     Check-Neighbor (+PAD, p_pixel + 1)
(4)     Check-Neighbor (-PAD, p_pixel - 1)
(5)
(6)     Check-Neighbor (direction, pad p, time t, last_adc)
(7)         adc ← pixel[p][t]
(8)         If (pixel[p][t] is clustered or adc > last_adc) then Return
(9)
(10)        Add pixel[p][t] to cluster
(11)
(12)        If (direction = +TIME) then
(13)            Check-Neighbor (+TIME, t+1, adc)
(14)            Check-Neighbor (+PAD, p+1, adc)
(15)            Check-Neighbor (-PAD, p-1, adc)
(16)        If (direction = -TIME) then
(17)            Check-Neighbor (-TIME, t-1, adc)
(18)            Check-Neighbor (+PAD, p+1, adc)
(19)            Check-Neighbor (-PAD, p-1, adc)
(20)        If (direction = +PAD)
(21)            Check-Neighbor (+PAD, p+1, adc)
(22)            Check-Neightbor (+TIME, t+1, adc)
(23)            Check-Neighbor (-TIME, t-1, adc)
(24)        If (direction = -PAD)
(25)            Check-Neighbor (-PAD, p-1, adc)
(26)            Check-Neighbor (+TIME, t+1, adc)
(27)            Check-Neighbor (-TIME, t-1, adc)
```

This simplistic clustering algorithm errs on the side of generating too many clusters. Basically, any local maximum will generate a new cluster. Nevertheless, the approach works quite well with the cluster deconvolution procedure described in the Cluster Deconvolution section.

6.6 Proto-Track Finder

The proto-track finder is the module which locates and creates new tracks. The proto-track finder operates in cluster space, and views clusters only as individual centroids. To form a new track, the procedure looks for three clusters which satisfy certain criteria. If the three chosen clusters do satisfy the criteria, then a new track is formed and its first three centroids are added to it.

The correctness of the proto-track finder is crucial to the overall performance of the tracker. If bad proto tracks are chosen for a new track, then there is nothing that the track extender can reasonably do to correct for it. Furthermore, there is a trade-off between having a high percentage of good tracks, but not identifying true tracks (false negatives) and having a large number of track, but including more proto tracks that are not actually the beginning of a true track (false positives). We choose to err on the side of more false positives, since the proto-track finder will filter them out when these tracks cannot be extended.

Another issue concerning the quality of the proto-track finder can also directly affect the performance of the track extender. Consider the case where the proto-track finder does not identify a particular track. When this unidentified track intersects other tracks, then it will throw off the segment extending of the other track since the track extender has no idea that there is actually another track contending for clusters. If the track has been successfully identified, however, then the track extender can act accordingly to handle any contention between the two for clusters.

6.6.1 Combinatorics

The original approach to creating new tracks was to form a master list of all clusters eligible for creating new tracks, choose each possible combination of three, and perform various tests on it. These tests determine whether or not the cluster triplet is a viable proto-track. This approach is a $\Theta(n^3)$ algorithm because

$$\binom{n}{3} = \Theta(n^3) .$$

This may have been acceptable if the number of clusters being considered were small. Nevertheless, for each iteration of the tracker routine, there are approximately 1500 clusters eligible to become new tracks. Thus, due to the extremely large number of possible proto-tracks (~456 million), the proto-track finder caused the tracker to run prohibitively slowly.

To solve this problem, the octant tree was applied very similarly to the way it was applied in the cluster finder module. The intuition here is that only clusters which are sufficiently close to one another can possibly be part of the same proto-track, so only test the cluster triplets in the volume around a cluster.

First the octant tree of clusters must be constructed, which takes $\Theta(n \log_8 n)$ time. Then each of the n clusters in the master line is used as the base cluster and all c nearby clusters are extracted from the octant tree. Note that extracting c clusters from the cluster tree requires $\Theta(c \log_8 n)$ time. Finally, all pairs of clusters of the c extracted clusters can be tested along with the base cluster to see if they form a valid proto-track. Thus, for each of the n base clusters, only $\Theta(c^2)$ work is required.

The total work of the improved proto-track finder, given a master list of n clusters, is $\Theta(n \log_8 n) + n * (\Theta(c \log_8 n) + \Theta(c^2)) = \Theta(n \log_8 n) + \Theta(n * (c \log_8 n + c^2))$ where $c << n$ (since only a small subset of the n clusters is extracted from the octant tree for each base cluster). Using the same figure for the number of clusters in the proto-track finder shell queue as before, 1500, we can see how the octant tree approach significantly reduces the amount of time required for the proto-track finder. To calculate this, however, the number of extracted clusters, c, must be estimated. This figure obviously depends on the size of the search volume surrounding the base cluster. Nevertheless, using the parameter settings in the current prototype and a shell queue of approximately 1500 clusters, c has turned out to be between 10 and 20. Disregarding the constants,

$$n \log_8 n + n * (c \log_8 n + c^2) = 1500*3.52 + 1500*(20*3.52 + 400) = 7.11\text{x}10^5,$$

far better than $4.56\text{x}10^8$ as in the original version. In practice, the realized running time is approximately 100-200 times faster than the original version, which is not far from the predicted improvement calculated above.

In general, the proto-track finder works in more than one shell at a time. This is because more than one shell thickness may be required to obtain three clusters which lie in the same track (and therefore would be well-correlated).

Below is the pseudocode for the proto-track finder:

```
(1)     Create master list of unclaimed clusters in the shell queue
(2)     Create octant tree of clusters (based on cluster centroids in master list)
(3)     Sort master list in decreasing order by ρ of the cluster centroids
(4)     For each cluster in the master list:
(5)         Find all clusters near base_cluster using octant tree
(6)         For each possible pair of extracted clusters (where base_centroid1->ρ > centroid2-
            >ρ > centroid3->ρ):
(7)             Perform tests on the three clusters
(8)                 If fail, then proceed to next pair
(9)             Create a new track
(10)            Add cluster centroids to the new track
(11)            Add new track to active track list
```

First, all clusters in the proto-track finder's shell queue are checked. Each that has not yet been claimed by a track is added to a master cluster list. This simple filtering greatly cuts down on the amount of processing necessary to find new tracks since time is not wasted testing more cluster triplets than necessary only to find that one of them has already been claimed.

There are four conditions that a set of three clusters must satisfy in order to pass as a new proto-track. (See Figure 6-7 for illustrated examples.)

1) The distance of the cluster centroids from the center of the TPC must be decreasing (centroid1->ρ > centroid2->ρ > centroid3->ρ). This ensures that the track is aimed basically in the direction of the interaction center.

2) Each cluster must contain at least p pixels. p is generally set to be quite small (4 or 5), and is in place to eliminate "noise" clusters from being used to begin new tracks.

3) The distance between the first and second centroids must be less than a certain parameter d. Likewise, the distance between the second and third centroids must be less than d. This test ensures that the sequence of clusters are within a reasonable distance of each other that they could conceivably be from the same track.

4) The angle (1-2-3) between the three centroids must be within *n* degrees of a straight line (180-degree angle). This colinearity test is used in recognition of the fast that consecutive clusters over a short interval of a track should be roughly in a straight line.

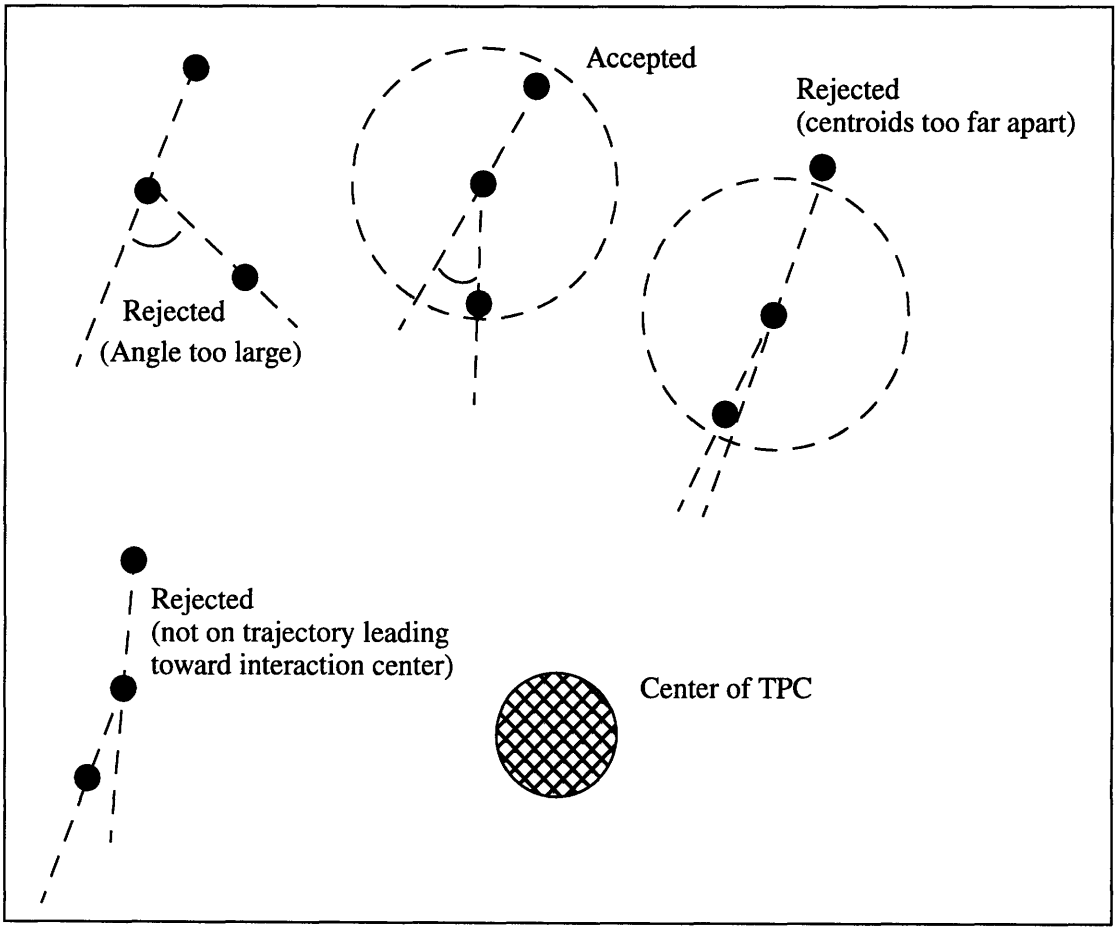

Figure 6-7: Tests for Candidate Proto-Tracks.

6.7 Track Extender

The track extender takes a pre-existing track (consisting of at least three centroids) and attempts to extend it to the next shell boundary. The inputs to the track extender are: the current shell (including the pixels and cluster list for that shell) and the active track list. Below is the pseudocode for the track extender module:

```
(1)     For each track:
(2)         Fit straight line to last two centroids of track
(3)         Calculate search cone angle
(4)         Calculate search cone bounding box
(5)         Extract pixels from octant tree which lie in bounding box
(6)         Filter extracted pixels for those lying in search cone
(7)         Find all clusters represented by the pixels in the search cone
(8)         Foreach cluster:
(9)             Decide whether cluster should be claimed by track
(10)            If so, add cluster to track's claimed cluster list
(11)    Deconvolve clusters
(12)    For each track:
(13)        If no clusters were claimed, increment track's num_shells_wo_new_centroid counter
(14)        Calculate a new centroid for the track from the claimed clusters
(15)        Label all pixels claimed by the track
```

Recall that a track is represented by a list of centroids. The proto-track finder uses these centroids to guide its search in the shell of pixels. In particular, a straight line is fitted to the last n centroids of the track (where *n* is a parameter specified by the programmer, currently set to 5). This line then forms the axis of a search cone. (See Figure 6-8 below.)

The search cone itself is described by two parameters: the base radius and the cone angle. A cone is used as opposed to a cylinder, because when extending a track, one would expect that the clusters nearest the last point in the track to be reasonably in-line with the track. As distance from the track increases, however, a larger region should be considered since the track trajectory is not known exactly. Furthermore, the track trajectories are helixes and therefore pixels farther from the last centroid in the track will not lie on the straight line fitted to the track centroids.

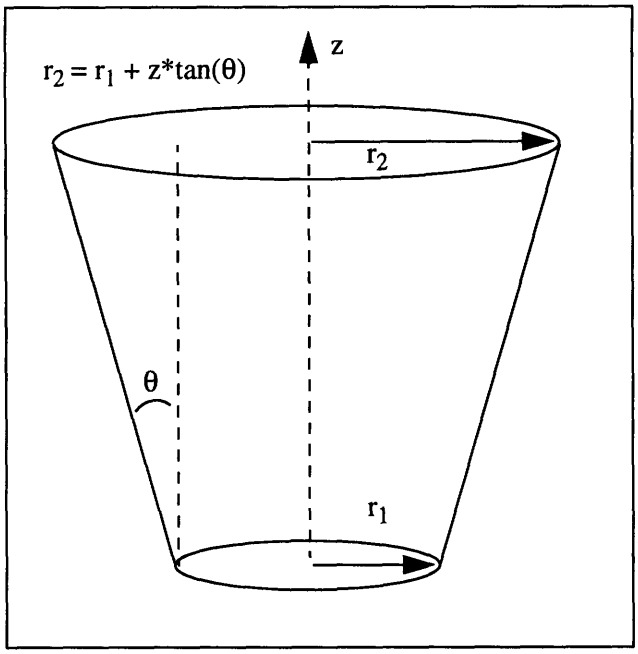

Figure 6-8: Track Extender Search Cone. The search cone is defined by two parameters: the base radius, r_1, and the angle, θ.

As more centroids are added to a track (i.e. the track is lengthened), it becomes more likely that a true track has, in fact, been found. To incorporate this intuition into the track extender, the cone angle θ is adjusted based on the total number of centroids in the track. The formula for the adjusted cone angle is a simple decaying exponential:

$$\theta = \theta_o \cdot e^{-an}$$

where θ_0 is the original cone angle, n is the number of centroids in the track, and a is a constant parameter set by the programmer. The effect of this function is to narrow the search cone as the track is extended. Notice that as n becomes large, θ approaches zero. This means that $r_1 \cong r_2$, and the search volume is therefore essentially a cylinder.

The bounding box is calculated by finding the minimum and maximum extents of the search cone along each of the coordinate axes. Recall that the octant tree representation used for this implementa-

tion only allows a search volume in the shape of a rectangular box. By extracting the pixels in the bounding box, the search time for pixels is decreased dramatically.

Once the pixels in the bounding box have been extracted, another procedure takes these pixels and tests each one to determine whether or not it lies within the search cone. Note that this procedure would function correctly on the entire shell of pixels, but would be prohibitively slow since every pixel in the shell would have to be tested.

To expand on this statement, as shown before, each extraction of p pixels takes $\Theta(p^*\log_8 n)$ time. The bounding box approach limits the number of pixels that are extracted to $\Theta(p)$ (where $p << n$), thereby preserving the $\Theta(p^*\log_8 n)$ total extraction time for these pixels. Following that, the pixel filtering procedure processes each pixel, so the total time for extracting and filtering pixels for one track extension is $\Theta(p) + \Theta(p^*\log_8 n) = \Theta(p^*\log_8 n)$, showing that the pixel filtering procedure does not increase the asymptotic running time of the track extender. Contrast this to filtering the entire set of n pixels for each track, which would require $\Theta(n)$ time per track. Thus it can be seen why the octant tree is a vital component for the track extender.

One it is known which pixels lie within the track's search cone, a list of the unique clusters represented by those pixels is generated. For example, if a total of 10 pixels were in the search cone, and 3 of them where part of cluster A, 5 were part of cluster B, and the remaining 2 were part of cluster C, then the cluster list would contain the set {A, B, C}.

The next step is to decide whether the track should actually claim the cluster. This is done by summing the ADC weights of the pixels from each cluster and comparing it to the total ADC weight of the cluster (which was calculated when the cluster was created by the cluster finder). If the ratio of ADC weights is greater than w (where w is an adjustable parameter), then the cluster is considered claimed. The intuition behind this test is that if a track's search cone barely encloses just a small edge or corner of a cluster, then the cluster probably should not be added to the track. On the other hand, if the cluster falls largely within the search cone, then it should be added.

If a cluster is claimed, then a pointer to the claiming track is added to the cluster's track list. This is done for the cluster deconvolution step, so it is known how many and which tracks have claimed a particular cluster.

6.8 Cluster Deconvolution

As shown in Figure 3-1, when two tracks are near each other, the response measured by the detector pads in the overlapping region is roughly the sum of the responses that would have been measured by the tracks individually. The result of this is a cluster of pixels that could be deconvolved.

The overall goal of deconvolution is to better guide the track extension. For the prototype tracker, a simple cluster deconvolver is being used which simply divides the ADC values of the pixels in a cluster by the number of tracks that claim that cluster. It will be argued why this is a highly effective, though not perfect, cluster deconvolver.

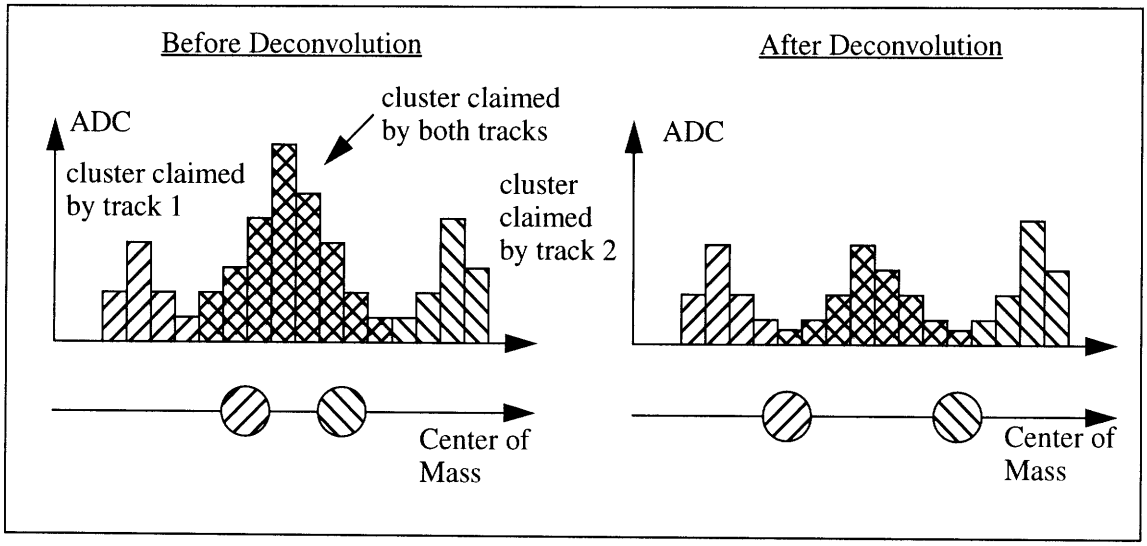

Figure 6-9: Cluster Deconvolution. The weight of a cluster claimed by multiple tracks is scaled down by the number of tracks that claim it.

Notice from Figure 6-9 above that the effect of the cluster deconvolution is to separate the centers of mass of the clusters for each track. The center cluster is produced by two partially-overlapping tracks. Without using the knowledge of how many tracks claim the middle cluster, the centers of mass

that would be calculated for the tracks' next centroids would be too close together. The goal, recall, is to follow the track of a particle exactly as it was when it passed through the TPC. When the cluster weight is resized based on the number of claiming tracks, the centers of mass are separated farther, thereby more accurately guiding each track.

Although there is no complicated deconvolution occurring, this approach accomplishes the goal of separating the calculated centroids of tracks that partially overlap. Since the cluster finder errs on the side of producing extra clusters (small valleys between peaks are considered separations between clusters), this simplistic approach works well. In fact, the clusters formed by the cluster finder can only have one basic shape - a peak with trailing edges (i.e. no other local peaks or valleys). Thus, there is no need to deconvolve this shape. Nevertheless, cluster deconvolution is certainly an area that could be improved in the prototype tracker.

6.9 Adjustable Parameters

This section is devoted to explaining the many parameters which can be adjusted by the programmer. Most of these parameters have been at least alluded to throughout the discussion of the tracker implementation. Nevertheless, for completeness, all parameters are listed below along with an explanation for each. More importantly, however, the inter-dependencies between these parameters will be discussed.

The following parameters are used in the main tracker routine:

NUM_CLUSTER_FINDER_SHELLS: specifies the maximum number of shells that should be in the cluster finder's shell queue. This parameter should not be set to any value less than 1.

NUM_PROTO_TRACK_FINDER_SHELLS: specifies the maximum number of shells that should be in the proto-track finder's shell queue.

SHELL_THICKNESS: specifies the thickness of each spherical shell. This parameter is one of the most crucial of all, and upon which many other parameters are gauged. The shell thickness reasonably should be set somewhere between 1.0 and 10.0 cm. Keep in mind that the shell thickness determines how many iterations of the main tracker loop are required to process all of the data.

The thinner the shells, the more shells that will be required for the cluster finder and proto-track finder since the size of clusters and extent of proto-tracks are properties of the TPC data itself and is not affected by the setting of any tracker parameter. Thinner shells, however, permit a more accurate tracking in general since the tracks do not need to be extended as far each time.

The following parameters are used in the cluster finder module:

CLUSTER_FINDER_SEARCH_CUBE_EDGE_LENGTH: specifies the length of each edge of the cube centered on each base cluster for the cluster finder (see Figure 6-5). This parameter can be estimated by looking at the data. Given a distribution of cluster sizes, the parameter should be chosen such that at least 95% of the clusters will be fully enclosed by the search cube. It is evident that if the edge length is set too low, then a single cluster may be divided into several smaller and irregularly-shaped clusters. The only complication that arises when the edge length is set too high is that the cluster finder's running time increases proportionally since more pixels need to be extracted and filtered each time.

MIN_NUM_PIXELS: specifies the minimum number of pixels required in a cluster in order for that cluster to be used as part of a proto-track. This parameter should be set to at least 1, but probably not above 10. This parameter's setting is important for filtering out "noise" clusters so they do not obstruct the proto-track finder from finding better candidate proto tracks. If set too high, however, then fewer proto-tracks will be formed, which means that a greater number of legitimate proto-tracks will be missed. As mentioned previously, it is better to allow the proto-track finder to err on the side of pro-

ducing extra (false positive) tracks, since the track extender is effective at filtering those tracks out quickly.

MAX_INTERPOINT_DISTANCE: specifies the maximum distance between two consecutive cluster centroids for them to be considered as candidates for a proto-track. The intuition behind this parameter is that the further apart two clusters are, the less likely is it that they were generated by the same track.

COLLINEARITY_THRESHOLD: specifies the angle between the line from centroid1 through centroid2 and the line from centroid2 through centroid3. (See Figure 6-7.) This parameter can be estimated from knowledge of how much tracks are expected to curve (which depends on the particle's charge, its velocity, and the magnetic field strength in the TPC). Set too low, this parameter causes the proto-track finder to overlook some valid tracks. And set too high, this parameter allows more bad tracks to be generated.

The following parameters are used in the track extender module:

NUM_LINEFIT_CENTROIDS: specifies the number of track centroids used to fit the straight line for track extending. This parameter must be at least 2 (takes at least 2 points to define a line), but shouldn't be set too large since the curvature of the track may throw off the track extension procedure. Note: this parameter is not currently being used.

NO_CENTROID_SHELL_LIMIT: specifies the number of shells a track may continue without finding any new clusters. This parameter should be at least 1, but could degrade tracker performance if set too large. This is because a track which has strayed from its true track path for one reason or another (perhaps a decay) could continue to claim clusters if it is permitted to search long enough (by intersecting another track, for example). Note that this parameter also depends on the shell thickness. For a thicker shell, this parameter should be set lower.

CLUSTER_PAINTING_THRESHOLD: specifies the percentage of a cluster's total ADC weight which must be enclosed by a track's search cone in order to claim it. If this parameter is set too high, then a track will be able to claim fewer clusters since they must lie more completely within its search cone. On the other hand, if the threshold is set too low, then any cluster that the search cone overlaps even by little will be added to the track.

The cluster-painting threshold is a parameter for which there is little to use as reference for gauging it. Therefore, this threshold is one of the most important parameters when conducting a search of the tracker's parameter space to find the combination of them which leads to the best performance.

SEARCH_CONE_BASE_RADIUS: specifies the radius of the track extender's search cone at the base. (See Figure 6-8.) This parameter can also be estimated from the data. The goal is to find the average width of a track as it is spread across multiple detector pads. If this radius is set too low, then fewer clusters will claimed by the track during extension. However, worse yet, if the radius is too large then the track could claim clusters from other nearby tracks, possibly throwing off the tracking.

Notice the inter-dependence between the cluster painting threshold and the search cone base radius. They both partially determine when clusters will be claimed by a track. And, like the cluster painting threshold, the search cone radius is another parameter which should be explored thoroughly when optimizing the tracker efficiency.

SEARCH_CONE_INITIAL_ANGLE: specifies the search cone angle from its longitudinal axis (measured from where the cone surface converges to a point on this axis). (See Figure 6-8.) In other words, this angle specifies how much the search cone widens as a function of distance from the search cone base. A larger angle would indicate less certainty in the trajectory of a new track. Note that if the angle were set to zero, the search volume would be a cylinder. As with the search cone base radius, if the cone angle is too large, then tracks will tend to claim clusters that do not belong to them. If the angle is too small, however, then a new track may be doomed to failure by not capturing the needed clusters to guide it along its trajectory.

SEARCH_CONE_ANGLE_TAPER_COEF: specifies the coefficient used for calculating the search cone angle as a function of the number of centroids in a track and the initial cone angle. Since the function is a decaying exponential, the taper coefficient determines how quickly the cone angle will decrease as a track is extended. Intuitively, the cone angle taper coefficient is a measure of how the confidence level of correctly following a track varies as the track is extended. If this coefficient is set too low, then the cone angle will not shrink significantly as a track is extended toward the interaction center. This becomes especially problematic as the track is being extended through the inner shells, where the track density is high, so it is especially important to only claim the clusters in the correct track. If the coefficient is set too high, then the decay curve will be steep, meaning that the angle will decrease quickly as the track is extended. This, in turn, could cause the track to stray from the correct trajectory since it might not capture the needed clusters to guide it along the right track.

6.10 Debugging and Testing

This section discusses the general methods that were used to test and debug the tracker implementation. Note that this does not include tuning parameters or even checking that parameters were set correctly. Instead, we are more interested in the correctness of the code.

Most of the basic debugging was performed using the gdb debugging utility. Running a program in the gdb environment permits the programmer to see where errors occur. This is extremely helpful for identifying and isolating problems.

When developing the program, the first step was to verify the correctness of each module independently. This was accomplished by supplying input to the module and checking the output to see if the results were as expected. For example, the cluster finder module was tested by supplying pixel data, then viewing the topology of the clusters that were produced.

Once black-box and glass-box testing was conducted on each module, the modules were integrated, and integration testing was conducted. Since the modules are largely independent of one

another, integration testing was quite simple since there are no few or no extra interactions between the modules.

Chapter 7

Simulation Data

7.1 Producing Test Data

Since RHIC and the STAR detector are not yet operational, the data used for developing, testing, and tuning the tracking algorithm had to be generated using an event simulator. There are three steps to generating such data.

First, a physics model is used to describe the collision. This model specifies how many and what type of particles will be produced in a typical collision. With this information, a utility called GEANT is used to simulate each particle's trajectory through a particular material. For STAR, this material would be the quark-gluon plasma. GEANT simulates particle decays, multiple scattering, and energy depositing. The output of the GEANT engine is energy for each track as a function of x, y, and z.

The output of GEANT is a general description of what would happen when the given particles react with one another. At this point, another simulator is used to model the TPC itself. This so-called *slow simulator* takes into account the physical construction of the TPC, the avalanche on the wire, pad images, and the electronics response.

The output of the slow simulator is the same as will be the output from the actual detector when it is functioning. Each detector pad that is activated produces an ADC value for that time slice. The ADC value is between 0 and 1023, and represents the magnitude of the pad's response, which is proportional to the ion's charge. Thus, a highly-charged particle will produce a large ADC value on the pad it meets.

One deficiency of the simulator is that each pixel can only be associated with one track in the output data. Internally, the simulator finds the track which contributes most to a pixel's ADC value and assigns to the pixel the ID of that track. As a result, when analyzing the performance of the tracker, tracks will tend to be less homogeneous in the pixels claimed by them.

7.2 Modeling High-Multiplicity Events

To test the tracking algorithm's response over a range of multiplicities, sets of low-multiplicity data can be generated (each containing approximately 1000 tracks). Then, to test the algorithm on a 1000-track event, only one of the data sets is used. Next, to obtain a 2000-track event, two 1000-track events can be combined, and so on. This technique can be used to explore the efficiency of the tracker over a range of multiplicities.

It is reasonable to combine events in this way since the events can, to a large degree, be considered independent for the purposes of testing the tracking algorithm. In actuality, more particles would interact more with one another at higher multiplicities, thereby leading to an increased number of decays. Nevertheless, these extra interactions are more important from a physics standpoint than from a tracking standpoint.

7.3 Data Preparation and Conversion

The output of the simulator is three text files: `tppad.asc`, `tppixel.asc`, and `tpmcpix.asc`. The `tppad` file is typically much smaller than the other two files, and each line is in the format:

index, number of time sequences, pad number, sector-row number

where *pad number* is the number of the pad. The *sector-row number* is a single number which indicates both the sector and row and is calculated as:

sector-row number = 100 * *sector number* + *row number*.

Next, *index* is an index into the `tppixel` and `tpmcpix` files which indicates the line number on which the pixels recorded for this sector, row, and pad begin. The *number of time sequences* indicates the number of sequences when the pad was occupied during consecutive time slices. This field is not needed for converting the simulator data nor is it used by the tracking program.

The `tppixel` file contains the ADC and TDC values for each pixel recorded during the event. Therefore, the number of lines in this file equals the total number of pixels recorded during the simu-

lated event. The `tppixel` data files used for this thesis represented the ADC and TDC values as a single "packed" number, from which the ADC and TDC values could be extracted.

The `tpmcpix` file contains the Monte-Carlo track ID's, one per line. Each line of this file corresponds with a line in the `tppixel` file. Thus each pixel has a Monte-Carlo track ID associated with it.

To prepare the data for the tracker, these files were first processed to form a single large text file (~150 MB), with each line representing one pixel recorded during an event. Reading a line from the `tppad` file gives the sector, row, and pad number for the next pixels that will be read. The index on that line gives the beginning index into the `tppixel` and `tpmcpix` files for pixels recorded for that particular pad. By reading ahead to the next line, the conversion routine can tell how many lines to read from each file. For example, if the beginning index is 57 and the next beginning index is 102, then lines 57 through 101 belong to the current pad. (See Appendix A.1 for the implementation of this routine.)

The goal of this conversion is to format each line as follows;

Sector, Row, Pad, TDC, ADC, True Track ID

where *Sector* is the detector sector number (1-24) where the pixel was registered, *Row* is the pad row of the pixel (1-45), *Pad* is the pad number (1-184, depending on which padrow), *TDC* is the time value (0-511), *ADC* is the magnitude of the pixel (0-1023), and *True Track ID* is the Monte Carlo track ID for that pixel.

The only difference between the simulator data and the real data that will be collected from the STAR detector when it becomes operational is that the real data will, of course, not have the true track ID labelled, since it is not known. (It is the job of the tracker to find such a designation for each pixel.)

To prepare the data for analysis by the tracking program, it first needs to be sorted in descending order by ρ for each pixel so that the data can be read efficiently in spherical shells beginning with the outermost shells. To this end, a simple routine is used to read in each line of the data file, calculate ρ for that pixel, then write the same data to an output file (including ρ). In order to calculate ρ, the pixel's

(sector, row, pad, tdc) values must first be converted into cartesian coordinates. This is accomplished using a custom-made routine which calculates the pixel's position in cartesian coordinates based on the measurements and specifications of the detector pad layouts and dimensions of the TPC (see Appendix A.2).

At this point, a new data file will have been generated with the format:

ρ, Sector, Row, Pad, TDC, ADC, Track_ID

The next step is to sort the new data file based on ρ. This can be accomplished using the standard UNIX *sort* command. To save time, the file can be split into numerous smaller files and each sorted independently. Then, the UNIX *sort* command can be used again to merge these data files together. The final outcome is a large data file sorted in descending order based on ρ. (For the reader's reference, these commands are given in Appendix A.3.)

Chapter 8

Evaluating the Tracker

8.1 Measuring Tracker Efficiency

Measuring the efficiency of the tracker is a difficult task. Ideally, one would like to calculate a single number that encapsulates all of the information about how well the tracker is doing. Nevertheless, since tracks are rarely 100% correctly or incorrectly found, the efficiency cannot be calculated simply as the percentage of correctly found tracks. This notion of track *pureness* is discussed in the next section (i.e. partial false positives and partial false negatives). To get an overall efficiency number, one could apply a threshold which determines whether a track was successfully found or not.

8.2 Diagnostics and Visualization

In order to evaluate the performance of the tracker and help tune the tracker parameters, the first step is to run the tracker on simulated data. The simulated data includes the true track ID's, which permits the experimenter to determine the efficiency of the tracker.

In debugging, one is searching for *errors* in the program. The diagnostic routines considered here serve a different purpose from those used for debugging in that they aid a programmer in evaluating how *well* a program is performing. Depending on the results of these diagnostics, the programmer may decide to modify some parameters in the tracking program or even modify the approach to tracking.

One simple yet very effective diagnostic is visualizing the results. The output of the tracking program is the same as the input, with an additional data field added - the found track ID. All pixels with the same found track ID are the ones that the tracker decided are in the same track. To check this against the *true* track, one can plot the true track and the centroids of the corresponding found track on top of the true track (preferably using different colors). If the two match well, then the tracker performed well on this track.

In addition to visualization, we also found it useful to evaluate tracker performance on a finer scale. A diagnostic program was developed to show the *pureness* of each track. This program processes the pixel output file from the tracker and can be keyed on either the true track ID or the found track ID. When keyed on the true track ID, the program considers each true track separately and displays all found tracks that contain pixels from that true track ID. For example:

```
True track ID: 1538
Found track ID          Portion of Pixels
14                      60%
5                       20%
76                      10%
Unclaimed               10%
```

This result would mean that Monte Carlo track #1538 from the simulator was partially claimed by three found tracks: 14, 5, and 76 and that 10% of the specified true track's pixels were left unclaimed by any found track.

For instance, the ideal result is one such as:

```
True track ID: 7834
Found track ID          Portion of Pixels
25                      100%
```

which means that all pixels from the true track were claimed by a single found track.

It is more common for a true track's pixels to be divided amongst several found tracks rather than vice-versa. This is because a true track is often segmented, causing the tracker to halt track extension along that track and begin a new track. If a true track's pixels are spread amongst many found tracks, it is useful to extract the associated pixels from the output data file and plot them. For example, in the first example above, one would extract pixels from true track 1538, and those from found tracks 14, 5, and 76. By plotting the pixels from each track in a different color, one can quickly determine what happened.

When keyed on the found track ID, the program produces a very similarly-formatted output:

```
Found track ID: 43
True track ID          Portion of Pixels
6729                   75%
1538                   20%
noise                  5%
```

which indicates that found track #43 consists of pixels from true tracks 6729 and 1538. A found track should not, in general, include large portions of pixels from other true tracks. Therefore, when this happens, it is important to plot the tracks to see what is happening. It will be common, however, for a found track to contain small portions of other true tracks, since it may claim these pixels when the true track it is following intersects another true track.

By combining the diagnostic program described above with a conventional 3-D viewer (e.g. Matlab®), the researcher can focus on particular true and found tracks which indicate poor tracking performance.

By examining these specific cases, modifications can be made to the tracker parameters or to the algorithm itself in order to strengthen its ability to handle such cases.

Chapter 9

Future Development

9.1 Improved Track Extrapolation

The current prototype only uses the last two centroids in a track to calculate the line which is the basis of calculating the search cone orientation. This approach has the favorable characteristics of being fast and simple to implement. However, using only the last two centroids of a track may be problematic due to the large variation of the centroids' positions. This means that the track may be extended in a skewed direction.

To fix this problem, a better approach would be to use the last n centroids of the track. A straight-line fit to these centroids would be a good start. An improvement on this would be to fit the sequence of centroids to a circular shape in the bend plane.

Using more than the last two centroids of a track has two primary benefits. The first is that the "noise" factor is reduced for track extrapolation. The trajectory of the track will therefore be more stable and gently varying (as it should). Secondly, using more of the track's history will better guide the track through small-angle intersections with other tracks. The intuition here is that when only a small portion of a track that is used for guidance, the track has only a small "memory" of where it has been and will therefore be more easily redirected to follow a different trajectory.

9.2 Detection and Halting of Multiply-Found Tracks

The previous section discussed using more of a track's history to guide it better. One of the goals of this is to help a found track follow the right true track when faced with a track intersection. We now view this from a different perspective - what to do if two found tracks happen to be following the same true track (a "multiply-found" track).

Multiply-found tracks are mostly harmless as far as the accuracy of the tracking goes, except for cases where the cluster deconvolution in important. Recall that cluster deconvolution uses the knowledge of how many tracks are claiming the same cluster. If, for example, three found tracks are following the same true track and there is an intersection with another true track, the second true track will not get its fair share of the multiply-claimed clusters.

Another potential complication that a multiply-found track presents arises from the fact that a pixel can only be assigned to one found track. If a pixel is claimed by more than one track, then the found track ID assigned to it is the one which is assigned last in the sequential order of the tracking program. As an artifact of the tracker's design, a multiply-claimed track's pixels will actually be assigned to only one particular track - the track that was formed most recently. If the tracker is parallelized, however, these pixels could ultimately be assigned any one of the ID's of the claiming tracks. This poses a serious problem for analyzing the results of the tracking program from the output pixel data.

Multiply-found tracks can either be formed when two tracks intersect at a small angle (as mentioned previously) or even from the very beginning of a track (proto-track). Proto-tracks can follow the same track since the proto-track finder has no way of knowing if this is happening. The proto-track finder merely finds triplets of clusters which appear to form a valid track. Thus an additional test needs to be constructed in the track extender module to detect and halt all but one found tracks following the same true track.

Multiply-found tracks can be detected by comparing the claimed clusters between pairs of tracks. If two tracks claim exactly the same clusters from shell to shell, then there is a good chance that they are both following the same track. The number of shells over which tracks need to be compared depends primarily on the track extrapolation method used for guiding the track extension. For instance, if only the last two centroids of the track are used to calculate the line around which the next search

cone will be formed, then if a pair of tracks claim exactly the same clusters over two shells, then they are necessarily following the same trajectory.

9.3 Intelligent Segment Joining

Rarely is an entire track followed from beginning to end by the tracker. For numerous reasons (noise, decays, TPC sector boundaries, intersecting tracks, etc.), one track may have been divided into numerous smaller segments during the tracking process. For this reason, a segment joining algorithm is needed to rejoin these segments into complete tracks. The expectation is that the segment joiner will increase the overall efficiency of the algorithm as well as aid the diagnostic procedures.

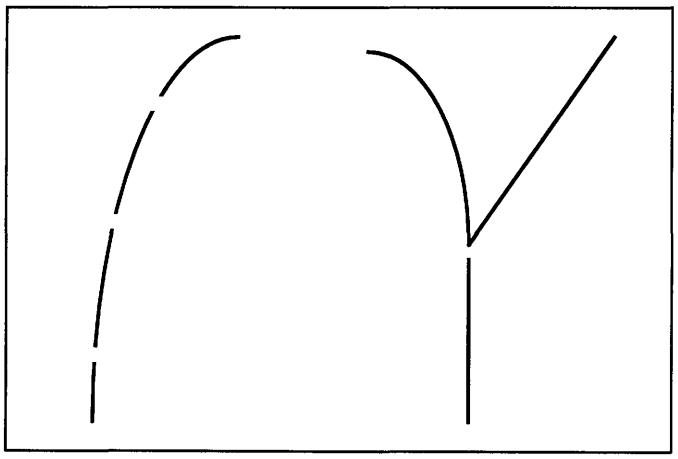

Figure 9-1: Segmented Tracks. Tracks can be segmented by TPC sector boundaries, decays, malfunctioning detector pads, etc.

By applying techniques borrowed from computer vision and object recognition research, an intelligent segment joining algorithm can be developed. The basic strategy is to characterize constraints on how segments can be joined. Examples of such constraints are the angles between two segments and the segment slopes and curvatures. Next, a tree is constructed representing all of the possibilities for joining the segments. Most of these possibilities can quickly be pruned out of the tree by applying the specified constraints. The remaining ones are then extended to include the possibilities of joining with

the next segment. The (desired) final result is one sequence of segments which, joined together, form a complete track. This stage of the overall tracking method embodies the more global view of the data. See [2] for more background on this approach to segment-joining.

9.4 Parallel Execution on a Multiprocessor Machine

9.4.1 Parallelization of the Tracker

Once all the optimizations possible have been made, the tracker may still require a long time to process each event. Fortunately, the tracking algorithm is well-suited to parallelization. Therefore, in the interest of processing events more quickly, a new version of the tracker program could be developed which would run in parallel on a multiprocessor machine. The following sections describe one approach to parallelizing the tracker.

9.4.2 Cilk - A Multithreaded Parallel Language

A multi-threaded language called *Cilk*[6] is a C-like programming language which allows a programmer to easily convert a C program into a parallel program. This language was developed primarily at the MIT Laboratory for Computer Science (LCS) under the guidance of Professor Charles Leiserson. Cilk is still evolving and is currently at version 5.

The parallelism in Cilk is at the procedural level, which is the level of parallelism that can be most-easily exploited by the tracker. The three most important functions in Cilk are: *cilk*, *spawn*, and *sync*. *cilk* designates a function as a Cilk function, which means that it can be run in parallel. *spawn* is the command which launches a new thread to run a procedure in parallel. And *sync* causes all threads running in parallel to synchronize at that point.

The Cilk language has the serial semantics of C. By removing the Cilk keywords, an ordinary C program remains which can be run serially. Compare the following two implementations of a procedure to calculate the Fibonacci of an argument n.

Cilk code:

```
(1)     cilk int fib (int n)
(2)     {
(3)         if (n<2) return (n);
(4)         else
(5)         {
(6)             int x,y;
(7)             x = spawn fib (n-1);
(8)             y = spawn fib (n-2);
(9)             sync;
(10)            return (x+y);
(11)        }
(12)    }
```

The corresponding C code would then be:

```
(1)     int fib (int n)
(2)     {
(3)         if (n<2) return (n);
(4)         else
(5)         {
(6)             int x,y;
(7)             x = fib (n-1);
(8)             y = fib (n-2);
(9)             return (x+y);
(10)        }
(11)    }
```

9.4.3 Multithreaded Computation

In multithreaded computation, once a procedure is called, the parent procedure continues to execute until a synchronization is encountered. Thus, the computation unfolds dynamically as a DAG of threads embedded in a tree of procedures. (See Figure 9-2 below.)

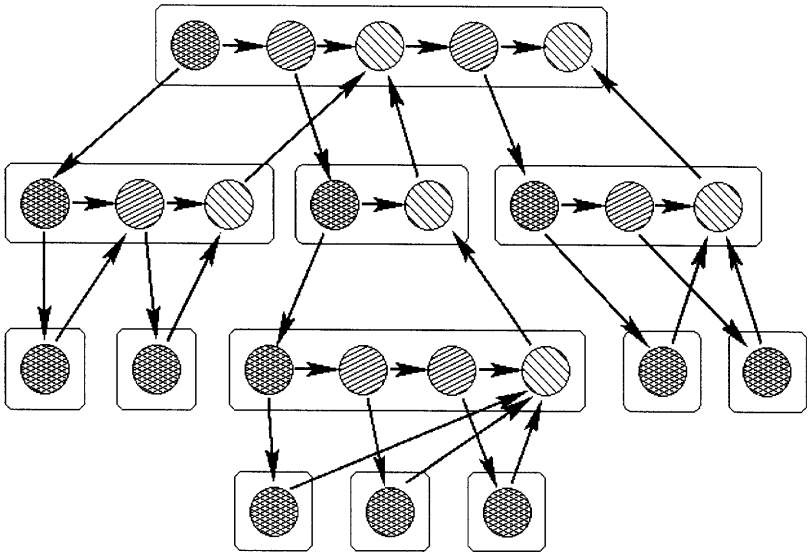

Figure 9-2: Multithreaded Computation DAG. In this diagram, procedures are represented as rounded rectangles and threads are shown as round nodes. Downward edges are spawns, horizontal edges represent sequential dependencies within a procedure, and returns are shown as upward arrows.

Key ideas:

- A *procedure* is a sequence of atomic *threads*.
- A procedure can *spawn* other procedures.
- Procedures are organized into a *tree hierarchy.*
- Return values induce additional *dependencies* among the threads.

An example of an algorithm that lends itself to multithreaded execution is the general Divide and Conquer algorithm (e.g. merge sort):

```
(1)     DivideAndConquer ( A )
(2)     {
(3)         if ( A is small ) return ( f(A) );
(4)         A1, A2 = divide ( A );
(5)
(6)         S1 = spawn DivideAndConquer ( A1 );
(7)         S2 = spawn DivideAndConquer ( A2 );
(8)         sync;
(9)
(10)        S = merge ( S1, S2 );
(11)        return ( S );
(12)    }
```

The first line handles the base case of the algorithm and is executed at a leaf of the dag and returns. If the base case does not apply, then the problem is divided into two sub problems (preferably of equal size). Next, a new thread is spawned to begin working recursively on the first subproblem. The difference between serial and multithreaded programs such as a Cilk program is that the parent continues executing. As a result, another thread is spawned to work on the second subproblem in parallel with the first. When the *sync* is reached, the parent waits for its spawned children to return. Once all of the children at that level returned, the parent can continue execution.

9.4.4 Compiling a Cilk Program

Cilk code is compiled in a 2-step process. First, a source-to-source translator (`cilk2c`) transforms the Cilk code into common C code. Once in C, a traditional C compiler (e.g. `gcc`) is used to compile the program into object code.

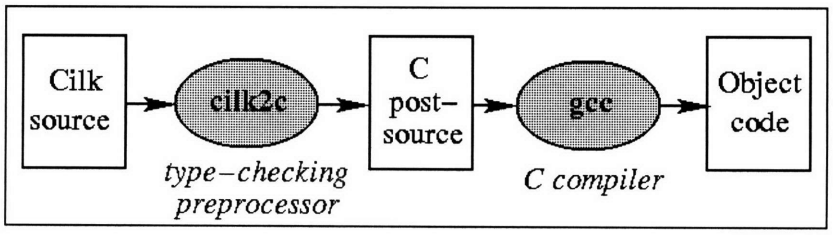

Figure 9-3: Compiling Cilk. After being translated to C, a conventional compiler such as gcc can be used to compile the code to form the executable.

9.5 Post-Mortem Processing

Another general approach to improving the overall performance of the tracker is to process the undesignated pixels from the output of the tracker. In the output data files, these pixels can be identified because their found track ID's have not been modified since they were initialized to zero. Routines that process this data could focus on tracks that the proposed tracker is weak on identifying.

Chapter 10

Conclusions

The prototype of the tracking program described in this thesis has been completed and is fully operational. At the time of this writing, however, a suitable data set from the simulator had not yet been obtained. As soon as this data is available, the tracking program can process it and the results can be analyzed using the diagnostic routines described in Chapter 8.

The proposed tracker meets the running time requirements necessary to analyze a data set in a reasonable amount of time. This tracker also features the characteristics needed to meet the efficiency requirements for identifying the quark-gluon plasma. With time and experimentation, the tracker parameters will be properly tuned to achieve optimal efficiency as well as adapt to various tracking conditions. The improvements suggested in Chapter 9, when implemented, will make the tracker even more effective. Even without these, however, the current prototype represents many improvements over traditional tracking algorithms and will undoubtedly be the model for the tracker which first confirms the existence of the quark-gluon plasma.

Appendix A
Data Preparation

A.1 Slow Simulator Output Format to Track Input File Format Conversion

generate_data_file.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <math.h>

#define FALSE 0
#define TRUE 1
#define MAX_LINE_LENGTH 200                                          10
#define XYZ_TO_RHO(x,y,z)  sqrt((x*x) + (y*y) + (z*z))

extern void srpt2xyz (int sector, int row, int pad, int tdc,
                      double *x, double *y, double *z);

void read_pixels_write_output (FILE *tppixel, FILE *tpmcpix, FILE *output,
                               int num_pixels, int sector, int row, int pad);


                                                                     20
main()
{
  char line[MAX_LINE_LENGTH], *line_ptr;
  FILE *tppad, *tppixel, *tpmcpix, *output;
  char token_chars[] = " \t\n";
  int sector, row, pad, sector_row, num_seq;
  int prev_sector, prev_row, prev_pad;
  int index, prev_index, num_pixels;

  tppad = fopen ("tppad.asc", "r");                                  30
  tppixel = fopen ("tppixel.asc", "r");
  tpmcpix = fopen ("tpmcpix.asc", "r");
  output = fopen ("/extra/mjduff/star_data.srpt", "w");

  prev_index = 1;

  while (1)
    {
      if (! fgets (line, MAX_LINE_LENGTH, tppad))  /* reached EOF */  40
          {
            read_pixels_write_output (tppixel, tpmcpix, output, INT_MAX,
                                      prev_sector, prev_row, prev_pad);
            break;
          }

      line_ptr = strtok (line, token_chars);
      index = atoi (line_ptr);

      line_ptr = strtok (NULL, token_chars);                         50
      num_seq = atoi (line_ptr);                  /* not using */

      line_ptr = strtok (NULL, token_chars);
      pad = atoi (line_ptr);

      line_ptr = strtok (NULL, token_chars);
      sector_row = atoi (line_ptr);

      row = sector_row % 100;
```

```
    sector = (sector_row - row) / 100;

    num_pixels = index - prev_index;

    read_pixels_write_output (tppixel, tpmcpix, output, num_pixels,
                              prev_sector, prev_row, prev_pad);

    prev_sector = sector;
    prev_row = row;
    prev_pad = pad;
    prev_index = index;
  }

 fclose (tppad);
 fclose (tppixel);
 fclose (tpmcpix);
 fclose (output);
}


void read_pixels_write_output (FILE *tppixel, FILE *tpmcpix, FILE *output,
                               int num_pixels, int sector, int row, int pad)
{
 register int i;
 char line[MAX_LINE_LENGTH], *line_ptr;
 char token_chars[] = " \t\n";
 long ldatum, nseqpix;
 int tdc, adc, tid;
 double x, y, z, rho;

 long nseq_factor = 0x100000;
 long tdc_factor = 0x400;

 for (i = 0; i < num_pixels; i++)
  {
    if (! fgets (line, MAX_LINE_LENGTH, tppixel))
        {
          printf ("possible problem!\n");
          break;
        }

    line_ptr = strtok (line, token_chars);
    ldatum = strtol (line_ptr, (char **) NULL, 10);

    nseqpix = ldatum / nseq_factor;
    tdc = (int) (ldatum - (nseq_factor * nseqpix)) / tdc_factor;
    adc = (int) ldatum - (nseq_factor * nseqpix) - (tdc_factor * tdc);

    if (! fgets (line, MAX_LINE_LENGTH, tpmcpix))
        {
          printf ("possible problem!\n");
          break;
        }

    line_ptr = strtok (line, token_chars);
    tid = atoi (line_ptr);

    srpt2xyz (sector, row, pad, tdc, &x, &y, &z);

    /* calculate rho for the pixel: */
    rho = XYZ_TO_RHO (x, y, z);

    fprintf (output, "%lf\t%d\t%d\t%d\t%d\t%d\t%d\n",
             rho, sector, row, pad, tdc, adc, tid);
  }

}
```

A.2 SRPT Format to Global Cartesian Coordinates Conversion

coord.c

```
#include <stdio.h>
#include <math.h>

#include "global.h"

/*************************************************************************
 *                          coord.c
 *
 * Padrow numbering: The padrows are numbered 1 to 45 starting from the
 *                inside.  Padrow n indicates the center of the nth
 *                padrow.
 *
 * Pad numbering: The pads in each padrow are numbered from left to right
 *              looking from outside the TPC starting with pad 1.  Pad n
 *              is the center of the nth pad.
 *
 *************************************************************************/

#define NUM_SECTORS 12
#define TPC_HALF_LENGTH 210.0      /* centimeters */

#define BASE_RADIUS1 60.0
#define BASE_RADIUS2 98.8          /* 60.0 + (7 * 4.8) + 5.2 */
#define BASE_RADIUS3 127.195
#define RADIAL_SPACING1 4.8
#define RADIAL_SPACING2 5.2
#define RADIAL_SPACING3 2.0
#define CROSS_SPACING1 0.335
#define CROSS_SPACING2 0.335
#define CROSS_SPACING3 0.67

int UNIT_VECTORS_INITIALIZED = FALSE;
double UNIT_LENGTH = TPC_HALF_LENGTH / 512;

/* unit vectors 30 degrees apart */
static double x_unit_vector[NUM_SECTORS];   /* 12 sectors */
static double y_unit_vector[NUM_SECTORS];   /* 12 sectors */

static int num_pads_per_row[] = {
  88, 96, 104, 112, 118, 126, 134, 142, 150, 156, 166, 174, 184, 98, 100,
  102, 104, 106, 106, 108, 110, 112, 112, 114, 116, 118, 120, 122, 122, 124,
  126, 128, 128, 130, 132, 134, 136, 136, 138, 140, 142, 144, 144, 144, 144
};

void initialize_unit_vectors (void);


/*************************************************************************
 *                          srpt2xyz
 *
 * Converts position given in terms of (sector, row, pad, tdc) into local
 * coordinates (x, y, z).
 *
 * All dimensions are in cm.
 *
 *************************************************************************/

void srpt2xyz (int sector, int row, int pad, int tdc,
               double *x, double *y, double *z)
{
  double ux, uy, nx, ny;   /* ux = unit_vector, nx = normal_vector */
  double rc, xc;
  int sector_index, neg_z_axis;

  if (UNIT_VECTORS_INITIALIZED == FALSE)
    initialize_unit_vectors();
```

```
/* Need better algorithm for calculating z */
if (sector <= 12)        /* z > 0 */
  *z = (511 - tdc) * UNIT_LENGTH;
else                     /* z < 0 */
  *z = (tdc - 511) * UNIT_LENGTH;

neg_z_axis = FALSE;
if (sector > 12)
  {
    sector -= 12;
    neg_z_axis = TRUE;
  }

sector_index = sector - 1;

ux = x_unit_vector[sector_index];
uy = y_unit_vector[sector_index];
nx = x_unit_vector[(sector_index + 3) % NUM_SECTORS];
ny = y_unit_vector[(sector_index + 3) % NUM_SECTORS];


if (row <= 8)              /* rows 1-8 */
  {
    rc = BASE_RADIUS1 + (row - 1) * RADIAL_SPACING1;
    xc = ((pad - num_pads_per_row[row - 1] / 2) * CROSS_SPACING1) -
         (CROSS_SPACING1 / 2);
  }

else if (row <= 13)        /* rows 9-13 */
  {
    rc = BASE_RADIUS2 + (row - 9) * RADIAL_SPACING2;
    xc = ((pad - num_pads_per_row[row - 1] / 2) * CROSS_SPACING2) -
         (CROSS_SPACING2 / 2);
  }

else                       /* rows 14-45 */
  {
    rc = BASE_RADIUS3 + (row - 14) * RADIAL_SPACING3;
    xc = ((pad - num_pads_per_row[row - 1] / 2) * CROSS_SPACING3) -
         (CROSS_SPACING3 / 2);
  }

/* vector = rc * [ux,uy] + xc * [nx, ny] */
if (neg_z_axis)
  *x = - ((rc * ux) + (xc * nx));
else
  *x = (rc * ux) + (xc * nx);

*y = (rc * uy) + (xc * ny);
}



/*****************************************************************************
 *                        initialize_unit_vectors                            *
 *                                                                           *
 *                                                                           *
 *****************************************************************************/

void initialize_unit_vectors (void)
{
 register int i;
 double theta;       /* radians counter-clockwise from positive x axis */

 UNIT_VECTORS_INITIALIZED = TRUE;

 for (i = 0; i < NUM_SECTORS; i++)
  {
   /* sectors numbered as clock (0 degrees is in sector 3) */
   theta = (2 - i) * (2 * M_PI) / NUM_SECTORS;
   x_unit_vector[i] = cos (theta);
```

```
    y_unit_vector[i] = sin (theta);
  }
}
```

A.3 Sorting Tracker Input File by ρ

Once main data file has been generated, split it into numerous smaller files:

```
split -l 100000 in_filename
rm in_filename
```

After splitting into smaller files, the original file (*in_filename*) is no longer needed and can be removed.

Next, use a simple csh shell script to sort each of the smaller files individually. sort's "-n" flag causes it to sort numerically as opposed to alphabetically. The "-r" flag means sort in descending order. And the "+0 -1" flags indicate that the sort should be carried out on the first column of the file only.

```
foreach file (x*)
    echo ``sorting $file...''
    sort -nr +0 -1 -o $file $file
end
```

Once each small file is sorted, merge the presorted smaller files:

```
sort -mnr +0 -1 -o out_filename x*
rm x*
```

Appendix B
Tracker Source Code

B.1 Global Parameters and Data Structures

global.h

```
#define TRUE 1
#define FALSE 0
#define MAX_LINE_LENGTH 200

#define DEFAULT_NUM_CLUSTER_FINDER_SHELLS 4
#define DEFAULT_NUM_PROTO_TRACK_FINDER_SHELLS 3
#define DEFAULT_NUM_LINEFIT_CENTROIDS 5
#define DEFAULT_NO_CENTROID_SHELL_LIMIT 3
#define DEFAULT_SHELL_THICKNESS 3.0              /* Centimeters */
#define DEFAULT_SEARCH_CONE_BASE_RADIUS 5.0
#define DEFAULT_SEARCH_CONE_INITIAL_ANGLE 0.785    /* 0.785 rad ~= 45 deg */
#define DEFAULT_SEARCH_CONE_ANGLE_TAPER_COEF 0.2
#define DEFAULT_CLUSTER_PAINTING_THRESHOLD 0.50      /* 50% */

#define DEFAULT_PTF_SEARCH_CONE_BASE_RADIUS 20.0
#define DEFAULT_PTF_SEARCH_CONE_ANGLE 0.785


#define DEFAULT_CLUSTER_FINDER_SEARCH_CUBE_EDGE_LENGTH 8.0

#define DEFAULT_MIN_NUM_PIXELS 5
#define DEFAULT_MAX_INTERPOINT_DISTANCE 8.0
#define DEFAULT_COLLINEARITY_THRESHOLD 0.35       /* 0.35 rad ~= 20 deg */

#define OCTANT1 1
#define OCTANT2 2
#define OCTANT3 3
#define OCTANT4 4
#define OCTANT5 5
#define OCTANT6 6
#define OCTANT7 7
#define OCTANT8 8

/* Useful macros: */
#define SQR(x) (x*x)
#define MAG(x,y,z) sqrt((x*x) + (y*y) + (z*z))
#define XYZ_TO_RHO(x,y,z)  sqrt((x*x) + (y*y) + (z*z))
#define DOT_PRODUCT(x1,y1,z1,x2,y2,z2) ((x1*x2) + (y1*y2) + (z1*z2))
#define DISTANCE(x1,y1,z1, x2,y2,z2) sqrt(((x2-x1)*(x2-x1)) + ((y2-y1)*(y2-y1)) + ((z2-z1)*(z2-z1)))


typedef struct PIXEL PIXEL;
typedef struct PIXEL_LIST PIXEL_LIST;
typedef struct CLUSTER CLUSTER;
typedef struct CLUSTER_LIST CLUSTER_LIST;
typedef struct CENTROID CENTROID;
typedef struct CENTROID_LIST CENTROID_LIST;
typedef struct TRACK TRACK;
typedef struct TRACK_LIST TRACK_LIST;
typedef struct SHELL SHELL;
typedef struct SHELL_QUEUE SHELL_QUEUE;
typedef struct OCTANT_TREE_NODE OCTANT_TREE_NODE;
typedef struct CLUSTER_TREE_NODE CLUSTER_TREE_NODE;


struct PIXEL {
  int true_tid, found_tid;
  double rho;
  double x, y, z;
```

```
  short int sector, row, pad, tdc;
  short int adc;
  CLUSTER *cluster;
};

struct PIXEL_LIST {
  int num_pixels;                                                    70
  PIXEL **pixel_array;
};

struct CLUSTER {
  short int claimed_by_track;
  int track_adc_weight;
  int total_adc_weight;
  CENTROID *centroid;
  PIXEL_LIST *pixel_list;                                            80
  TRACK_LIST *claiming_tracks;
};

struct CLUSTER_LIST {
  int num_clusters;
  CLUSTER **cluster_array;
};

                                                                     90
struct CENTROID {
  double rho;
  double x, y, z;
};

struct CENTROID_LIST {
  int num_centroids;
  CENTROID **centroid_array;
};                                                                  100

/* A track consists of multiple centroids */
struct TRACK {
  int found_tid;
  int num_shells_wo_new_centroid;
  CENTROID_LIST *centroid_list;
  CLUSTER_LIST *claimed_clusters;
};
                                                                    110
struct TRACK_LIST {
  int num_tracks;
  TRACK **track_array;
};

struct SHELL {
  double rho_min, rho_max;
  PIXEL_LIST *pixel_list;                                           120
  CLUSTER_LIST *cluster_list;
  SHELL *next_shell;
  SHELL *next_shell_in_queue;
};

/* A shell queue functions as a FIFO queue – with push and pop operations */

struct SHELL_QUEUE {
  int num_shells;                                                   130
  SHELL *first_shell;
  SHELL *last_shell;
};

/*
 * octant tree invariant:
 * If pixel != NULL, then the node is an interior node.
 * If pixel == NULL, then the node is a leaf of the tree and represents
```

```
 * a pixel.                                                                    140
 */

struct OCTANT_TREE_NODE {
  OCTANT_TREE_NODE *oct1, *oct2, *oct3, *oct4, *oct5, *oct6, *oct7, *oct8;
  double x_min, x_max, y_min, y_max, z_min, z_max;
  PIXEL *pixel;
};

struct CLUSTER_TREE_NODE {                                                    150
  CLUSTER_TREE_NODE *oct1, *oct2, *oct3, *oct4, *oct5, *oct6, *oct7, *oct8;
  double x_min, x_max, y_min, y_max, z_min, z_max;
  CLUSTER *cluster;
};
```

B.2 Main Tracking Routine

tracker.c

```
#include <stdio.h>

#include "global.h"
#include "cluster.h"
#include "pixel.h"
#include "shell.h"
#include "track.h"

extern void cluster_finder (SHELL_QUEUE *shell_queue, int eof);

extern void track_extender (SHELL *shell,
                            TRACK_LIST* active_track_list,
                            int num_linefit_centroids,
                            int no_centroid_shell_limit,
                            double search_cone_base_radius,
                            double search_cone_initial_angle,
                            double search_cone_angle_taper_coef,
                            double cluster_painting_threshold);

void proto_track_finder (SHELL_QUEUE *shell_queue,
                         TRACK_LIST *active_track_list);


void main()
{
  FILE *input_file_ptr, *pixel_data_file_ptr, *track_data_file_ptr;
  double rho_min, rho_max;
  char filename[100], answer[100];
  int total_pixels_read, shell_count, eof;

  double rho_step = DEFAULT_SHELL_THICKNESS;
  int num_linefit_centroids = DEFAULT_NUM_LINEFIT_CENTROIDS;
  int no_centroid_shell_limit = DEFAULT_NO_CENTROID_SHELL_LIMIT;
  int num_cluster_finder_shells = DEFAULT_NUM_CLUSTER_FINDER_SHELLS;
  int num_proto_track_finder_shells = DEFAULT_NUM_PROTO_TRACK_FINDER_SHELLS;
  double search_cone_base_radius = DEFAULT_SEARCH_CONE_BASE_RADIUS;
  double search_cone_initial_angle = DEFAULT_SEARCH_CONE_INITIAL_ANGLE;
  double search_cone_angle_taper_coef = DEFAULT_SEARCH_CONE_ANGLE_TAPER_COEF;
  double cluster_painting_threshold = DEFAULT_CLUSTER_PAINTING_THRESHOLD;

  PIXEL_LIST *shell_pixels;

  TRACK_LIST *active_track_list;

  SHELL *new_shell, *current_shell, *old_shell, *next_shell;
  SHELL_QUEUE *cluster_finder_shell_queue, *proto_track_finder_shell_queue;

  /*********************************************************************/

  printf("\n");
  printf("Sorted data filename:  ");
  scanf("%s", filename);

  if (! (input_file_ptr = fopen (filename, "r")))
    {
      printf("Couldn't open %s for reading - exiting\n", filename);
      exit (0);
    }


  /*
  printf("Output data base filename: ");
  scanf("%s", filename);
  */

  strcpy (filename, "output_data.pixel");          /* Temporary */
```

```
if (! (pixel_data_file_ptr = fopen (filename, "w")))
  {
    printf("Couldn't open %s for writing - exiting\n", filename);
    exit (0);
  }

strcpy (filename, "output_data.track");              /* Temporary */
                                                                                    80
if (! (track_data_file_ptr = fopen (filename, "w")))
  {
    printf("Couldn't open %s for writing - exiting\n", filename);
    exit (0);
  }


printf("\n");
printf("For the following prompts, press 'Return' for the default value:");        90
printf("\n\n");

gets (answer);   /* purge any character(s) from the stdin buffer */


/*
printf("Spherical shell thickness (default = %lf): ",
       DEFAULT_SHELL_THICKNESS);
gets (answer);                                                                     100
if (strlen (answer) > 0)
  rho_step = atof (answer);
  */


/*
printf("Number of centroids to use for track extending (default = %d): ",
         DEFAULT_NUM_LINEFIT_CENTROIDS);
gets (answer);
if (strlen (answer) > 0)                                                           110
  num_linefit_centroids = atoi (answer);
  */


/*
printf("No centroid shell limit (default = %d): ",
       DEFAULT_NO_CENTROID_SHELL_LIMIT);
gets (answer);
if (strlen (answer) > 0)
  no_centroid_shell_limit = atoi (answer);                                         120
  */


/*
printf("Number of cluster-finder shells (default = %lf): ",
       DEFAULT_NUM_CLUSTER_FINDER_SHELLS);
gets (answer);
if (strlen (answer) > 0)
  num_cluster_finder_shells = atoi (answer);
  */                                                                               130


/*
printf("Number of proto-track-finder shells (default = %lf): ",
       DEFAULT_NUM_PROTO_TRACK_FINDER_SHELLS);
gets (answer);
if (strlen (answer) > 0)
  num_proto_track_finder_shells = atoi (answer);
  */
                                                                                   140

/*
printf("Search cone base radius (default = %lf): ",
       DEFAULT_SEARCH_CONE_BASE_RADIUS);
gets (answer);
if (strlen (answer) > 0)
  search_cone_base_radius = atof (answer);
```

```
  */

/*
printf("Search cone initial angle (default = %lf degrees): ",
      DEFAULT_SEARCH_CONE_INITIAL_ANGLE);
gets (answer);
if (strlen (answer) > 0)
  search_cone_initial_angle = atof (answer);
  */

/*
printf("Search cone angle taper coefficient (default = %lf): ",
      DEFAULT_SEARCH_CONE_ANGLE_TAPER_COEF);
gets (answer);
if (strlen (answer) > 0)
  search_cone_angle_taper_coef = atof (answer);
  */

/*
printf("Threshold for cluster painting: (default = %lf): ",
      DEFAULT_CLUSTER_PAINTING_THRESHOLD);
gets (answer);
if (strlen (answer) > 0)
  cluster_painting_threshold = atof (answer);
  */

/*************************************************************************/

/* Create needed lists and queues: */
active_track_list = create_track_list();

cluster_finder_shell_queue = create_shell_queue();
proto_track_finder_shell_queue = create_shell_queue();

/* Initialize rho_max to rho of the first data file entry: */
read_data (input_file_ptr, 0.0, &rho_max);
rho_min = rho_max - rho_step;

printf ("rho_max = %lf\n", rho_max);                    /* Temporary */
total_pixels_read = 0;
shell_count = 1;
eof = FALSE;

/* Each iteration of the loop below corresponds to one spherical shell:
   (rho_min .. rho_max)  This loop continues until all data has been
   processed. */

while (1) {
  printf("shell %d (%lf ..  %lf)\n", shell_count, rho_min, rho_max);

  /* Read in next shell of data */
  if (shell_pixels = read_data (input_file_ptr, rho_min, NULL))
    {
        total_pixels_read += shell_pixels->num_pixels;
        printf ("num pixels read:  %d (%d total)\n",
                shell_pixels->num_pixels, total_pixels_read);

        new_shell = create_shell (rho_min, rho_max, shell_pixels);

        if (cluster_finder_shell_queue->last_shell != NULL)
          cluster_finder_shell_queue->last_shell->next_shell = new_shell;

        push_onto_shell_queue (cluster_finder_shell_queue, new_shell);
    }
  else
    eof = TRUE;      /* reached end of data file */

  if (cluster_finder_shell_queue->num_shells > num_cluster_finder_shells ||
```

```
        eof == TRUE)
    {
        current_shell = pop_off_shell_queue (cluster_finder_shell_queue);

        if (current_shell == NULL)
          break;        /* there are no further shells to process */                    230

        cluster_unclustered_pixels (current_shell);
    }
  else
    current_shell = NULL;


  if (cluster_finder_shell_queue->num_shells > 0)
    /* Find pixel clusters and calculate their centroids: */
    cluster_finder (cluster_finder_shell_queue, eof);                                  240


  if (current_shell == NULL)      /* no shells available to process yet */
    {
        rho_min -= rho_step;
        rho_max -= rho_step;
        shell_count++;
        printf ("\n");
        continue;
    }                                                                                  250

  /* track extender operates on the first shell after cluster finder
     is finished. */

  /* Run track extender */
  track_extender (current_shell, active_track_list,
                  num_linefit_centroids, no_centroid_shell_limit,
                  search_cone_base_radius, search_cone_initial_angle,
                  search_cone_angle_taper_coef, cluster_painting_threshold);           260

  active_track_list =
    remove_inactive_tracks (track_data_file_ptr, active_track_list,
                            no_centroid_shell_limit);


  /* Add new shell to front of proto-track-finder's shell queue: */
  push_onto_shell_queue (proto_track_finder_shell_queue, current_shell);

  if (proto_track_finder_shell_queue->num_shells >                                     270
        num_proto_track_finder_shells)
    {
        old_shell = pop_off_shell_queue (proto_track_finder_shell_queue);

        /* Write shell's pixel data to pixel output file: */
        printf ("writing pixel data to output file...\n");
        write_pixel_data_to_file (pixel_data_file_ptr, old_shell->pixel_list);

        printf ("freeing shell...\n");              /* Temporary */
        free_shell (old_shell);                                                        280
    }


  /* Run proto-track finder on the remaining pixels and/or clusters.
     Attempts to create new tracks.  Adds new tracks to active_track_list */
  proto_track_finder (proto_track_finder_shell_queue, active_track_list);


  rho_min -= rho_step;                                                                 290
  rho_max -= rho_step;
  shell_count++;
  printf ("\n");
}


/* Write remaining shells to pixel output file: */
current_shell = proto_track_finder_shell_queue->first_shell;

while (current_shell != NULL)                                                          300
  {
```

```
    /* Write current_shell's pixel data to pixel output file: */
    printf ("writing pixel data to output file...\n");
    write_pixel_data_to_file (pixel_data_file_ptr,
                              current_shell->pixel_list);

    next_shell = current_shell->next_shell_in_queue;
    free_shell (current_shell);
    current_shell = next_shell;
  }                                                              310

 /* Write track data to track output file: */
 printf ("writing track data to output file...\n");
 write_track_data_to_file (track_data_file_ptr, active_track_list);

 fclose (pixel_data_file_ptr);
 fclose (track_data_file_ptr);
 fclose (input_file_ptr);
                                                                320
 free_track_list (active_track_list);

 free_shell_queue (cluster_finder_shell_queue);
 free_shell_queue (proto_track_finder_shell_queue);
}
```

B.3 Cluster Finder Module

cluster_finder.c

```
#include <stdio.h>
#include <math.h>

#include "global.h"
#include "cluster.h"
#include "pixel.h"
#include "octant_tree.h"

int MinTdc, MaxTdc, MinPad, MaxPad;

static struct { int dpad; int dtdc; } glomm_next[] = {
  {-1, 0}, { 0, 1}, { 1, 0},
  { 0, 1}, { 1, 0}, { 0,-1},
  { 1, 0}, { 0,-1}, {-1, 0},
  { 0,-1}, {-1, 0}, { 0, 1}
};

static struct { int dpad; int dtdc; } neighbors[] = {
  {-1, 0}, { 1, 0},
  { 0,-1}, { 0, 1},
  {-1,-1}, { 1, 1},
  {-1, 1}, { 1,-1}
};

CLUSTER* find_cluster (PIXEL_LIST *padrow_pixels, PIXEL *pixel);

PIXEL_LIST* glomm (PIXEL ***pixel_matrix, CLUSTER *cluster, int pad, int tdc);

void glomm_internal_recurse (PIXEL ***pixel_matrix, CLUSTER *cluster,
                             PIXEL_LIST *pixel_list,
                             int pad, int tdc, int last_adc, int direction);


/*****************************************************************************
 *                          cluster_finder
 *
 * Note that it is perfectly acceptable for a cluster to consist of only a
 * single pixel.
 *
 *****************************************************************************/

void cluster_finder (SHELL_QUEUE *shell_queue, int eof)
{
  register int i;
  double half_edge = DEFAULT_CLUSTER_FINDER_SEARCH_CUBE_EDGE_LENGTH / 2;
  double sqrt_3_times_half_edge = sqrt(3) * half_edge;
  SHELL *innermost_shell, *current_shell;
  PIXEL *pixel;
  PIXEL **pixel_array;
  PIXEL_LIST *pixel_list, *found_pixels, *filtered_pixels;
  PIXEL_LIST *master_pixel_list;
  CLUSTER *new_cluster;
  OCTANT_TREE_NODE *octant_tree;
  int num_new_clusters, total_num_pixels;

  printf ("entering cluster_finder...\n");

  /* note that last_shell is the inner-most shell in the TPC: */
  innermost_shell = shell_queue->last_shell;

  if (innermost_shell->cluster_list == NULL)
    innermost_shell->cluster_list = create_cluster_list();

  /* Create master list of unclustered pixels for all of the shells: */
  master_pixel_list = create_pixel_list();
```

```
total_num_pixels = 0;
current_shell = shell_queue->first_shell;

while (current_shell != NULL)
  {
    pixel_list = current_shell->pixel_list;
    pixel_array = pixel_list->pixel_array;
                                                                          80
    for (i = 0; i < pixel_list->num_pixels; i++)
        if (pixel_array[i]->cluster == NULL)
          add_to_pixel_list (master_pixel_list, pixel_array[i]);

    total_num_pixels += pixel_list->num_pixels;
    current_shell = current_shell->next_shell_in_queue;
  }

/* Temporary: */                                                          90
printf ("total number of pixels in cluster-finder shell queue:  %d\n",
         total_num_pixels);

printf ("total number of unclustered pixels before cluster-finder:  %d\n",
         master_pixel_list->num_pixels);

if (master_pixel_list->num_pixels < DEFAULT_MIN_NUM_PIXELS)
  {
    free_pixel_list (master_pixel_list);
    return;                                                               100
  }

printf ("building cluster-finder octant tree (%d pixels)...\n",
         master_pixel_list->num_pixels);

octant_tree = build_octant_tree (master_pixel_list);

printf ("done\n");

                                                                          110
/* sort entire pixel list by ADC values (decreasing order): */
sort_pixel_list_by_adc (master_pixel_list);


printf ("searching for new clusters...\n");

num_new_clusters = 0;

/* Go through each pixel and extract nearby pixels: */
for (i = 0; i < master_pixel_list->num_pixels; i++)                       120
  {
    pixel = master_pixel_list->pixel_array[i];

    if (pixel->cluster != NULL)
        continue;       /* pixel has recently become clustered */

    if (! eof &&       /* don't apply this constraint after data file EOF */
          (pixel->rho - sqrt_3_times_half_edge) < innermost_shell->rho_min)
        continue;  /* Search cube could be partially inside the innermost
                      shell of the shell queue, so wait until next shell    130
                      is added. */

    found_pixels = create_pixel_list();

    extract_pixels (octant_tree, found_pixels,
                    pixel->x - half_edge, pixel->x + half_edge,
                    pixel->y - half_edge, pixel->y + half_edge,
                    pixel->z - half_edge, pixel->z + half_edge);

    filtered_pixels = same_row_and_sector_filter (found_pixels, pixel);   140
    free_pixel_list (found_pixels);


    /* run conventional cluster-finder algorithm: */
    new_cluster = find_cluster (filtered_pixels, pixel);
    free_pixel_list (filtered_pixels);
```

```
    if (new_cluster != NULL)
        {
          place_cluster_in_shell_queue (shell_queue, new_cluster);
          num_new_clusters++;
        }
  }

 free_pixel_list (master_pixel_list);

 printf ("found %d new clusters\n", num_new_clusters);


 free_octant_tree (octant_tree);

 printf ("leaving cluster_finder...\n");
}



/*****************************************************************************
 *                          find_cluster
 *
 * Roughly functionally equivalent to MAL's mountainfinder
 *
 *****************************************************************************/

CLUSTER* find_cluster (PIXEL_LIST *pixel_list, PIXEL *base_pixel)
{
 register int i;
 int num_pads, num_tdc;
 int dpad, dtdc, pass_test;
 PIXEL *current_pixel, *matrix_pixel;
 PIXEL ***pixel_matrix;
 PIXEL_LIST *cluster_pixels;
 CLUSTER *new_cluster = NULL;


 /* Find the maximum & minimum for the tdc and pad #'s: */
 MaxTdc = 0; MinTdc = 512; MaxPad = 0; MinPad = 144;

 for (i = 0; i < pixel_list->num_pixels; i++)
   {
    current_pixel = pixel_list->pixel_array[i];

    MaxTdc = (current_pixel->tdc > MaxTdc) ? current_pixel->tdc : MaxTdc;
    MinTdc = (current_pixel->tdc < MinTdc) ? current_pixel->tdc : MinTdc;
    MaxPad = (current_pixel->pad > MaxPad) ? current_pixel->pad : MaxPad;
    MinPad = (current_pixel->pad < MinPad) ? current_pixel->pad : MinPad;
   }


 num_pads = (MaxPad - MinPad) + 1;
 num_tdc = (MaxTdc - MinTdc) + 1;

 /* Allocate memory for the pixel matrix: */
 pixel_matrix = (PIXEL***) calloc (num_pads, sizeof (PIXEL**));

 for (i = 0; i < num_pads; i++)
   pixel_matrix[i] = (PIXEL**) calloc (num_tdc, sizeof (PIXEL*));


 /* Assign pixels from the given pixel list to the pixel matrix: */
 for (i = 0; i < pixel_list->num_pixels; i++)
   {
    current_pixel = pixel_list->pixel_array[i];

    pixel_matrix[current_pixel->pad - MinPad][current_pixel->tdc - MinTdc] =
        current_pixel;
   }


 /* check neighboring pixels for higher ADC values: */
 pass_test = TRUE;

 for (i = 0; i < 8; i++)
```

```
  {
    dpad = neighbors[i].dpad;
    dtdc = neighbors[i].dtdc;

    if (base_pixel->pad + dpad < MinPad || base_pixel->pad + dpad > MaxPad ||
          base_pixel->tdc + dtdc < MinTdc || base_pixel->tdc + dtdc > MaxTdc)
        continue;

    matrix_pixel =
        pixel_matrix[base_pixel->pad - MinPad + dpad][base_pixel->tdc - MinTdc + dtdc];

    if (matrix_pixel != NULL &&
          matrix_pixel->cluster == NULL &&
          matrix_pixel->adc > base_pixel->adc)
        {
          pass_test = FALSE;
          break;
        }
  }

 if (pass_test == FALSE)
   return NULL;


 new_cluster = create_cluster (NULL);    /* create empty cluster */

 cluster_pixels = glomm (pixel_matrix, new_cluster,
                         base_pixel->pad, base_pixel->tdc);

 free_cluster (new_cluster);

 /* Temporary: */
 /* unparse_adc_matrix (pixel_matrix, num_pads, num_tdc, pixel, cluster_pixels); */

 new_cluster = create_cluster (cluster_pixels);

 /* free pixel matrix: */
 for (i = 0; i < num_pads; i++)
   free (pixel_matrix[i]);
 free (pixel_matrix);

 return new_cluster;
}


/*****************************************************************************
 *                              glomm
 *
 *
 *****************************************************************************/

PIXEL_LIST* glomm (PIXEL ***pixel_matrix, CLUSTER *cluster, int pad, int tdc)
{
 register int direction;
 int pad_index = pad - MinPad;
 int tdc_index = tdc - MinTdc;
 PIXEL_LIST *pixel_list;

 pixel_list = create_pixel_list();

 /* not already assigned */
 if (pixel_matrix[pad_index][tdc_index]->cluster == NULL)
   {
     add_to_pixel_list (pixel_list, pixel_matrix[pad_index][tdc_index]);
     pixel_matrix[pad_index][tdc_index]->cluster = cluster;

     /* Explore the neighboring pixels: */
     for (direction = 0; direction < 4; direction++)
         glomm_internal_recurse (pixel_matrix, cluster, pixel_list,
```

```
                                pad + glomm_next[3 * direction + 1].dpad,
                                tdc + glomm_next[3 * direction + 1].dtdc,
                                pixel_matrix[pad_index][tdc_index]->adc,
                                direction);
    }

  return pixel_list;
}                                                                          310


/*****************************************************************************
 *                      glomm_internal_recurse
 *
 *
 *****************************************************************************/
                                                                           320
void glomm_internal_recurse (PIXEL ***pixel_matrix, CLUSTER *cluster,
                             PIXEL_LIST *pixel_list,
                             int pad, int tdc, int last_adc, int direction)
{
  register int i;
  int pad_index = pad - MinPad;
  int tdc_index = tdc - MinTdc;

  if (pad < MinPad || pad > MaxPad || tdc < MinTdc || tdc > MaxTdc)        330
    return;     /* If run into pixel matrix boundary --> complete
                   (search area assumed to be large enough to
                   encapsulate the cluster completely) */

  if (pixel_matrix[pad_index][tdc_index] != NULL &&        /* pixel exists */
      pixel_matrix[pad_index][tdc_index]->cluster == NULL && /* not assigned */
      pixel_matrix[pad_index][tdc_index]->adc <= last_adc) /* descending adc */
    {
      add_to_pixel_list (pixel_list, pixel_matrix[pad_index][tdc_index]);  340
      pixel_matrix[pad_index][tdc_index]->cluster = cluster;

      /* Explore the neighboring pixels recursively: */
      for (i = 0; i < 4; i++)
          glomm_internal_recurse (pixel_matrix, cluster, pixel_list,
                                  pad + glomm_next[3 * direction + i].dpad,
                                  tdc + glomm_next[3 * direction + i].dtdc,
                                  pixel_matrix[pad_index][tdc_index]->adc,
                                  direction);
    }                                                                      350

  return;
}
```

B.4 Proto-Track Finder Module

proto_track_finder.c

```
#include <stdio.h>
#include <math.h>

#include "global.h"
#include "cluster.h"
#include "cluster_tree.h"
#include "track.h"
#include "centroid.h"

double distance (CENTROID *centroid1, CENTROID *centroid2);

double triplet_angle (CENTROID *centroid1, CENTROID *centroid2,
                      CENTROID *centroid3);



/*****************************************************************************
 *                        proto_track_finder                                 *
 *                                                                           *
 *                                                                           *
 *****************************************************************************/

void proto_track_finder (SHELL_QUEUE *shell_queue, TRACK_LIST *track_list)
{
  register int i;
  int num_new_tracks;
  SHELL *current_shell;
  CLUSTER_TREE_NODE *cluster_tree;
  CLUSTER *current_cluster, **cluster_array;
  CLUSTER *base_cluster, *cluster2, *cluster3;
  CLUSTER_LIST *eligible_clusters, *cluster_list;
  CLUSTER_LIST *found_clusters;
  CENTROID *base_centroid, *centroid2, *centroid3;
  CENTROID *new_track_centroid;
  TRACK *new_track;
  double x0, y0, z0, vx, vy, vz;
  double px0, py0, pz0;
  double x_min, x_max, y_min, y_max, z_min, z_max;
  int num_clusters, cluster_index2, cluster_index3;

  double queue_rho_min = shell_queue->last_shell->rho_min;
  double queue_rho_max = shell_queue->first_shell->rho_max;


  printf ("entering proto_track_finder (%lf .. %lf) ...\n",
          queue_rho_min, queue_rho_max);


  /* Create a list of all clusters from the shells in the queue which
     have not yet been claimed by a track: */
  eligible_clusters = create_cluster_list();

  current_shell = shell_queue->first_shell;

  while (current_shell != NULL)
    {
      cluster_list = current_shell->cluster_list;

      /* Clear out the clusters already claimed by tracks.  This is
         done so the proto-track-finder doesn't use these clusters for
         creating new tracks. */
      for (i = 0; i < cluster_list->num_clusters; i++)
          {
            current_cluster = cluster_list->cluster_array[i];

            if (current_cluster->claimed_by_track == FALSE &&
                current_cluster->pixel_list->num_pixels >= DEFAULT_MIN_NUM_PIXELS)
              add_to_cluster_list (eligible_clusters, current_cluster);
          }
```

```
    current_shell = current_shell->next_shell_in_queue;
  }

printf ("%d eligible clusters for new tracks\n",
        eligible_clusters->num_clusters);

                                                                          80

cluster_tree = build_cluster_tree (eligible_clusters);

num_new_tracks = 0;

for (i = 0; i < eligible_clusters->num_clusters; i++)
  {
    base_cluster = eligible_clusters->cluster_array[i];
    base_centroid = base_cluster->centroid;
                                                                          90
    x0 = base_centroid->x;
    y0 = base_centroid->y;
    z0 = base_centroid->z;

    /* Vector from cluster centroid to origin: */
    vx = -x0;
    vy = -y0;
    vz = -z0;

    if (! calculate_bounding_box (queue_rho_min, queue_rho_max,          100
                                  x0, y0, z0, vx, vy, vz,
                                  DEFAULT_PTF_SEARCH_CONE_BASE_RADIUS,
                                  DEFAULT_PTF_SEARCH_CONE_ANGLE,
                                  &x_min, &x_max, &y_min,
                                  &y_max, &z_min, &z_max,
                                  &px0, &py0, &pz0))
        {
          printf ("this shouldn't happen!!\n");
          printf ("queue_rho:  %lf ..  %lf\n", queue_rho_min, queue_rho_max);
          printf ("(x0, y0, z0) = (%lf, %lf, %lf)\n", x0, y0, z0);      110
          printf ("Bounding box:  %lf-%lf, %lf-%lf, %lf-%lf\n",
                  x_min, x_max, y_min, y_max, z_min, z_max);
          printf ("(px0, py0, pz0) = (%lf, %lf, %lf)\n", px0, py0, pz0);
          continue;
        }

    /* Find pixels in bounding box: */
    /* This narrows the search significantly over checking each cluster
         in the entire shell queue. */                                    120
    found_clusters = create_cluster_list();

    extract_clusters (cluster_tree, found_clusters,
                      x_min, x_max, y_min, y_max, z_min, z_max);


    /* Temporary: */
    /*
    printf ("%d clusters extracted in bounding box\n",
            found_clusters->num_clusters);                               130
    unparse_cluster_list (found_clusters);
    */

    num_clusters = found_clusters->num_clusters;
    if (num_clusters < 3)
        {
          free_cluster_list (found_clusters);
          continue;
        }
                                                                         140
    /* Sort cluster list based on rho (decreasing order): */
    sort_cluster_list_by_rho (found_clusters);


    /* printf ("initializing get_next_cluster_pair...\n"); */
    cluster_index2 = 0;
```

```
/* set index to the cluster after the base_cluster: */
cluster_array = found_clusters->cluster_array;
while (cluster_array[cluster_index2++] != base_cluster);
cluster_index3 = cluster_index2;


/* Test all possible triplets of clusters: */
while (1)
    {
     cluster_index3++;

     if (cluster_index3 >= num_clusters)
       {
        cluster_index2++;
        cluster_index3 = cluster_index2 + 1;
       }

     if (cluster_index2 >= num_clusters - 1)
       break;

     cluster2 = cluster_array[cluster_index2];
     cluster3 = cluster_array[cluster_index3];

     centroid2 = cluster2->centroid;
     centroid3 = cluster3->centroid;

     /* perform various tests on the triplet: */
     if (distance (base_centroid, centroid2) >
         DEFAULT_MAX_INTERPOINT_DISTANCE)
       continue;

     if (distance (centroid2, centroid3) >
         DEFAULT_MAX_INTERPOINT_DISTANCE)
       continue;

     if (triplet_angle (base_centroid, centroid2, centroid3) >
         DEFAULT_COLLINEARITY_THRESHOLD)
       continue;

     /*
       printf ("new proto track found!\n");
       unparse_cluster (cluster1);
       unparse_cluster (cluster2);
       unparse_cluster (cluster3);
       */


     /* If the triplet passes all of the tests, then create a new track
        using the 3 centroids: */
     new_track = create_track ();
     new_track->claimed_clusters = create_cluster_list();

     /* Now add the centroids to the new track: */
     new_track_centroid = create_centroid (base_centroid->x,
                                           base_centroid->y,
                                           base_centroid->z);
     add_centroid_to_track (new_track, new_track_centroid);
     add_to_cluster_list (new_track->claimed_clusters, base_cluster);
     base_cluster->claimed_by_track = TRUE;

     new_track_centroid =
       create_centroid (centroid2->x, centroid2->y, centroid2->z);
     add_centroid_to_track (new_track, new_track_centroid);
     add_to_cluster_list (new_track->claimed_clusters, cluster2);
     cluster2->claimed_by_track = TRUE;

     new_track_centroid =
       create_centroid (centroid3->x, centroid3->y, centroid3->z);
     add_centroid_to_track (new_track, new_track_centroid);
     add_to_cluster_list (new_track->claimed_clusters, cluster3);
     cluster3->claimed_by_track = TRUE;

     /* Add the new track to the track list and assign found_tid: */
     add_new_found_track (track_list, new_track);

     label_track_pixels (new_track);
```

```
            free_cluster_list (new_track->claimed_clusters);
            new_track->claimed_clusters = NULL;

            num_new_tracks++;
          }

      free_cluster_list (found_clusters);       /* No longer needed */
    }

  free_cluster_list (eligible_clusters);

  printf ("found %d new tracks\n", num_new_tracks);

  free_cluster_tree (cluster_tree);

  printf ("leaving proto_track_finder...\n");
}



/*****************************************************************************
 *                              distance
 *
 *
 *****************************************************************************/

double distance (CENTROID *centroid1, CENTROID *centroid2)
{
  double dx = centroid1->x - centroid2->x;
  double dy = centroid1->y - centroid2->y;
  double dz = centroid1->z - centroid2->z;

  return sqrt ((dx * dx) + (dy * dy) + (dz * dz));
}



/*****************************************************************************
 *                            triplet_angle
 *
 * Returns angle between line from 1 to 2 and line from 2 to 3.
 *
 *****************************************************************************/

double triplet_angle (CENTROID *centroid1, CENTROID *centroid2,
                      CENTROID *centroid3)
{
  double dx1, dy1, dz1;
  double dx2, dy2, dz2;
  double vec1_dot_vec2, vec1_mag, vec2_mag;
  double cos_angle;

  dx1 = centroid1->x - centroid2->x;
  dy1 = centroid1->y - centroid2->y;
  dz1 = centroid1->z - centroid2->z;

  dx2 = centroid3->x - centroid2->x;
  dy2 = centroid3->y - centroid2->y;
  dz2 = centroid3->z - centroid2->z;

  vec1_dot_vec2 = DOT_PRODUCT (dx1,dy1,dz1, dx2,dy2,dz2);
  vec1_mag = MAG (dx1, dy1, dz1);
  vec2_mag = MAG (dx2, dy2, dz2);

  cos_angle = vec1_dot_vec2 / (vec1_mag * vec2_mag);

  if (cos_angle > 0.999) cos_angle = 0.999;
  if (cos_angle < -0.999) cos_angle = -0.999;

  return (M_PI - acos (cos_angle));
}
```

B.5 Track Extender Module

track_extender.c

```
#include <stdio.h>

#include "global.h"
#include "octant_tree.h"
#include "pixel.h"
#include "cluster.h"
#include "track.h"

/*****************************************************************************
 *                          track_extender
 *
 * Attempt to extend all tracks in active_track_list in the spherical
 * shell from rho_max to rho_min.
 *
 *****************************************************************************/

void track_extender (SHELL *shell,
                     TRACK_LIST* active_track_list,
                     int num_linefit_centroids, int no_centroid_shell_limit,
                     double search_cone_base_radius,
                     double search_cone_initial_angle,
                     double search_cone_angle_taper_coef,
                     double cluster_painting_threshold)
{
  register int i;
  PIXEL_LIST *found_pixels, *filtered_pixels;
  TRACK *current_track;
  CLUSTER *current_cluster;
  CLUSTER_LIST *claimed_clusters;
  CENTROID *new_centroid;
  OCTANT_TREE_NODE *octant_tree;
  double rho_min = shell->rho_min;
  double rho_max = shell->rho_max;
  double shell_thickness, mid_shell_next, mid_shell_prev;
  double search_cone_angle;
  double x0, y0, z0, vx, vy, vz;
  double px0, py0, pz0;
  double x_min, x_max, y_min, y_max, z_min, z_max;


  printf ("entering track_extender (%lf ..  %lf) ...\n",
          rho_min, rho_max);

  /* Build octant tree for current search space */
  octant_tree = build_octant_tree (shell->pixel_list);

  printf ("num pixels inserted into track-extender octant tree:  %d\n",
          shell->pixel_list->num_pixels);


  /* Recluster any pixels whose cluster was free'd: */
  cluster_unclustered_pixels (shell);


  /* Start a new claiming_tracks list for each cluster
     (in case the cluster was claimed from the next shell): */
  for (i = 0; i < shell->cluster_list->num_clusters; i++)
    {
      current_cluster = shell->cluster_list->cluster_array[i];

      if (current_cluster->claiming_tracks != NULL)
          {
            free_track_list (current_cluster->claiming_tracks);
            current_cluster->claiming_tracks = create_track_list();
          }
    }
```

```
/*
  Iterate through all existing tracks.  For each:
  - find the pixels which lie in the search space,
  */

for (i = 0; i < active_track_list->num_tracks; i++)
  {
    current_track = active_track_list->track_array[i];
    current_track->claimed_clusters = create_cluster_list();                    80

    /*
    printf("processing track %d (%d of %d)...\n",
            current_track->found_tid, i + 1, active_track_list->num_tracks);
            */

    fit_line_to_centroids (current_track->centroid_list,
                           num_linefit_centroids,
                           &x0, &y0, &z0, &vx, &vy, &vz);
                                                                                90
    /* Calculate search cone angle from number of centroids in
         track, the initial angle, and the angle taper coefficient */
    search_cone_angle =
        calculate_search_cone_angle (search_cone_initial_angle,
                                     search_cone_angle_taper_coef,
                                     current_track->centroid_list->num_centroids);

    shell_thickness = rho_max - rho_min;                                        100
    mid_shell_prev = rho_max + (shell_thickness / 2.0);
    mid_shell_next = rho_min - (shell_thickness / 2.0);

    /* Calculate rectangular bounding box around search cone: */
    if (! calculate_bounding_box (mid_shell_next, mid_shell_prev,
                                  x0, y0, z0, vx, vy, vz,
                                  search_cone_base_radius, search_cone_angle,
                                  &x_min, &x_max, &y_min,
                                  &y_max, &z_min, &z_max,
                                  &px0, &py0, &pz0))                            110
        {
          /* cannot calculate bounding box */
          /* printf ("Cannot extend track %d - continuing with next track...\n",
                  current_track->found_tid); */
          continue;
        }

    /* Find pixels in bounding box: */
    /* (search must be performed in pixel space) */                             120
    /* This narrows the search significantly over checking each pixel
         in the entire shell. */
    found_pixels = create_pixel_list();

    extract_pixels (octant_tree, found_pixels,
                    x_min, x_max, y_min, y_max, z_min, z_max);

    /* Temporary: */
    /*                                                                          130
    printf ("track %d: %d pixels extracted in bounding box\n",
            current_track->found_tid, found_pixels->num_pixels);
            */

    /* Now check individual pixels in box search volume for inclusion
         or exclusion from search cone: */
    filtered_pixels =
        filter_for_search_cone_pixels (found_pixels,
                                       px0, py0, pz0, vx, vy, vz,
                                       search_cone_base_radius,                 140
                                       search_cone_angle);

    free_pixel_list (found_pixels);       /* No longer needed */

    /* Temporary: */
    /* printf ("track %d: %d pixels in search cone\n",
```

```
            current_track->found_tid, filtered_pixels->num_pixels); */

    /* Decide which clusters overlapped by the search cone should be
         claimed by the track.  Then add these clusters to the track's
         claimed_clusters list. */
    paint_clusters (filtered_pixels, current_track,
                    cluster_painting_threshold);

    /* Temporary: */
    /* printf ("Number of clusters claimed by track %d: %d\n",
               current_track->found_tid,
               current_track->claimed_clusters->num_clusters); */

    free_pixel_list (filtered_pixels);    /* No longer needed */
  }


for (i = 0; i < active_track_list->num_tracks; i++)
  {
    current_track = active_track_list->track_array[i];
    claimed_clusters = current_track->claimed_clusters;

    if (claimed_clusters->num_clusters == 0) /* No clusters were claimed */
        {
          /* printf ("num_shells_wo_new_centroid++ for track %d\n",
                     current_track->found_tid); */

          current_track->num_shells_wo_new_centroid++;

          free_cluster_list (claimed_clusters);
          current_track->claimed_clusters = NULL;
          continue;
        }

    /* Clusters were claimed --> reset shell count to zero */
    current_track->num_shells_wo_new_centroid = 0;

    new_centroid = calculate_new_track_centroid (current_track);
    add_centroid_to_track (current_track, new_centroid);

    /* Label all pixels claimed by the track with the track's found_tid: */
    label_track_pixels (current_track);

    /* unparse_track (current_track);  */          /* Temporary */

    /* claimed_clusters no longer needed */
    free_cluster_list (claimed_clusters);
    current_track->claimed_clusters = NULL;
  }

 free_octant_tree (octant_tree);

 printf ("leaving track_extender...\n");
}
```

B.6 Pixel and Pixel List Functions

pixel.h

```
extern void srpt2xyz (int sector, int row, int pad, int tdc,
                      double *x, double *y, double *z);

PIXEL_LIST* read_data (FILE* file_ptr, double rho_min, double* rho_max);

PIXEL_LIST* read_xyz_data (FILE* file_ptr, double rho_min, double* rho_max);

PIXEL* create_pixel (double rho, double x, double y, double z,
                     int sector, int row, int pad, int tdc,                    10
                     int adc, int true_tid);

void unparse_pixel (PIXEL* pixel);

void unparse_pixel_list (PIXEL_LIST* pixel_list);

PIXEL_LIST* create_pixel_list (void);

void add_to_pixel_list (PIXEL_LIST* pixel_list, PIXEL* new_pixel);
                                                                              20
void free_pixel_list (PIXEL_LIST* pixel_list);

void free_pixel_array (PIXEL_LIST* pixel_list);

PIXEL_LIST* same_row_and_sector_filter (PIXEL_LIST *pixel_list, PIXEL *pixel);

void sort_pixel_list_by_adc (PIXEL_LIST *pixel_list);

void write_pixel_data_to_file (FILE *file_ptr, PIXEL_LIST *pixel_list);
```

pixel.c

```
#include <stdio.h>
#include <string.h>
#include "global.h"
#include "pixel.h"


/*****************************************************************************
 *                          create_pixel
 *                                                                            10
 * create_pixel allocates memory for a new PIXEL, and assigns the supplied
 * values to the various fields of the PIXEL.
 *
 *****************************************************************************/

PIXEL* create_pixel (double rho, double x, double y, double z,
                     int sector, int row, int pad, int tdc,
                     int adc, int true_tid)
{
  PIXEL *pixel = (PIXEL*) malloc (sizeof (PIXEL));                            20

  pixel->true_tid = true_tid;
  pixel->found_tid = 0;

  pixel->rho = rho;
  pixel->x = x;
  pixel->y = y;
  pixel->z = z;

  pixel->sector = sector;                                                     30
  pixel->row = row;
  pixel->pad = pad;
  pixel->tdc = tdc;

  pixel->adc = adc;
```

```
  pixel->cluster = NULL;

  return pixel;
}


/*****************************************************************************
 *                          unparse_pixel
 *
 * unparse_pixel prints the values of the PIXEL struct (pointed to by
 * 'pixel') fields.
 *
 *****************************************************************************/

void unparse_pixel (PIXEL *pixel)
{
  if (pixel == NULL)
    return;

  printf("rho:   %lf\n", pixel->rho);
  printf("x:   %lf\n", pixel->x);
  printf("y:   %lf\n", pixel->y);
  printf("z:   %lf\n", pixel->z);
  printf("sector:   %d\n", pixel->sector);
  printf("row:   %d\n", pixel->row);
  printf("pad:   %d\n", pixel->pad);
  printf("tdc:   %d\n", pixel->tdc);
  printf("adc:   %d\n", pixel->adc);
  printf("true_tid:   %d\n", pixel->true_tid);
  printf("found_tid:   %d\n", pixel->found_tid);
  printf("\n\n");
}



/*****************************************************************************
 *                          unparse_pixel_list
 *
 * unparse_pixel_list calls unparse_pixel for each pixel in 'pixel_list'.
 *
 *****************************************************************************/

void unparse_pixel_list (PIXEL_LIST *pixel_list)
{
  int i;

  if (pixel_list == NULL)
    return;

  printf("\n\n");
  printf("PIXEL LIST (%d pixels)\n", pixel_list->num_pixels);
  printf("----------------------\n");

  for (i = 0; i < pixel_list->num_pixels; i++)
    unparse_pixel (pixel_list->pixel_array[i]);
}



/*****************************************************************************
 *                          read_data
 *
 * The data file lines are assumed to be in the format:
 *
 *          rho   sector   row   pad   tdc   adc   tid
 *
 * and are assumed to be sorted on rho.
 *
 *****************************************************************************/

PIXEL_LIST* read_data (FILE* file_ptr, double rho_min, double* rho_max)
```

```
{
 int eof = FALSE;
 char line[MAX_LINE_LENGTH], *line_ptr;
 char token_chars[] = " \t\n";
 double rho, x, y, z;
 int sector, row, pad, tdc, adc, tid;
 long position;
 PIXEL* pixel;
 PIXEL_LIST* pixel_list;

 if (rho_max != NULL)     /* Initialize rho_max */
  {
   fgets (line, MAX_LINE_LENGTH, file_ptr);    /* get first line of file */

   line_ptr = strtok (line, token_chars);
   *rho_max = atof (line_ptr);

   rewind (file_ptr);   /* set file pointer back to beginning of file */
   return NULL;
  }

 pixel_list = create_pixel_list ();

 while (1)
  {
   position = ftell (file_ptr);  /* file position before reading line */

   if (! fgets (line, MAX_LINE_LENGTH, file_ptr))  /* reached EOF */
       {
        eof = TRUE;
        break;
       }

   line_ptr = strtok (line, token_chars);
   rho = atof (line_ptr);

   if (rho < rho_min) { fseek (file_ptr, position, SEEK_SET); break; }

   line_ptr = strtok (NULL, token_chars);
   sector = atoi (line_ptr);

   line_ptr = strtok (NULL, token_chars);
   row = atoi (line_ptr);

   line_ptr = strtok (NULL, token_chars);
   pad = atoi (line_ptr);

   line_ptr = strtok (NULL, token_chars);
   tdc = atoi (line_ptr);

   line_ptr = strtok (NULL, token_chars);
   adc = atoi (line_ptr);

   line_ptr = strtok (NULL, token_chars);
   tid = atoi (line_ptr);

   srpt2xyz (sector, row, pad, tdc, &x, &y, &z);

   pixel = create_pixel (rho, x, y, z, sector, row, pad, tdc, adc, tid);
   add_to_pixel_list (pixel_list, pixel);
  }

 if (eof && pixel_list->num_pixels == 0)
  {
   free_pixel_list (pixel_list);
   return NULL;
  }
 else
  return pixel_list;
}
```

```
/*****************************************************************************
 *                          read_xyz_data
 *
 * The data file lines are assumed to be in the format:
 *
 *                    rho   x   y   z   adc   tid
 *
 * and are assumed to be sorted on rho.
 *
 *****************************************************************************/

PIXEL_LIST* read_xyz_data (FILE* file_ptr, double rho_min, double* rho_max)
{
  int eof = FALSE;
  char line[MAX_LINE_LENGTH], *line_ptr;
  char token_chars[] = " \t\n";
  double rho, x, y, z;
  int adc, tid;
  long position;

  PIXEL* pixel;
  PIXEL_LIST* pixel_list;


  if (rho_max != NULL)     /* Initialize rho_max */
    {
     fgets (line, MAX_LINE_LENGTH, file_ptr);    /* get first line of file */

     line_ptr = strtok (line, token_chars);
     *rho_max = atof (line_ptr);

     rewind (file_ptr);   /* set file pointer back to beginning of file */
     return NULL;
    }


  pixel_list = create_pixel_list ();


  while (1)
    {
     position = ftell (file_ptr);  /* file position before reading line */

     if (! fgets (line, MAX_LINE_LENGTH, file_ptr))  /* reached EOF */
          {
           eof = TRUE;
           break;
          }

     line_ptr = strtok (line, token_chars);
     rho = atof (line_ptr);

     if (rho < rho_min) { fseek (file_ptr, position, SEEK_SET); break; }

     line_ptr = strtok (NULL, token_chars);
     x = atof (line_ptr);

     line_ptr = strtok (NULL, token_chars);
     y = atof (line_ptr);

     line_ptr = strtok (NULL, token_chars);
     z = atof (line_ptr);

     line_ptr = strtok (NULL, token_chars);
     adc = atoi (line_ptr);

     line_ptr = strtok (NULL, token_chars);
     tid = atoi (line_ptr);


     pixel = create_pixel (rho, x, y, z, 0, 0, 0, 0, adc, tid);
     add_to_pixel_list (pixel_list, pixel);
```

```
    }

  if (eof && pixel_list->num_pixels == 0)
    {
      free_pixel_list (pixel_list);
      return NULL;
    }
  else
    return pixel_list;
}


/*****************************************************************************
 *                          create_pixel_list
 *
 * create_pixel_list allocates memory for and returns an empty pixel list
 * It sets pixel_list->num_pixels to zero, the the pixel_array to NULL.
 * free_pixel_list should be called to deallocate the memory when the
 * pixel list is no longer needed.
 *
 *****************************************************************************/
PIXEL_LIST* create_pixel_list (void)
{
  PIXEL_LIST *pixel_list = (PIXEL_LIST*) malloc (sizeof (PIXEL_LIST));

  pixel_list->num_pixels = 0;
  pixel_list->pixel_array = NULL;

  return pixel_list;
}


/*****************************************************************************
 *                          add_to_pixel_list
 *
 * add_to_pixel_list takes an existing pixel_list and a pixel, and adds
 * pixel to the pixel_list.  pixel_list->num_pixels is incremented in
 * accordance with the addition.  add_to_pixel_list will work fine on any
 * pixel_list (including an empty one).
 *
 *****************************************************************************/

void add_to_pixel_list (PIXEL_LIST *pixel_list, PIXEL *new_pixel)
{
  int num_pixels = pixel_list->num_pixels;

  pixel_list->pixel_array =
    (PIXEL**) realloc ((PIXEL**) pixel_list->pixel_array,
                       (num_pixels + 1) * sizeof (PIXEL*));

  pixel_list->pixel_array[num_pixels] = new_pixel;
  pixel_list->num_pixels++;
}


/*****************************************************************************
 *                          free_pixel_list
 *
 * free_pixel_list deallocates memory for the pixel list pointed to by
 * 'pixel_list'.  This function should be called when a pixel list is no
 * longer needed (but the pixel data should be retained).
 *
 *****************************************************************************/

void free_pixel_list (PIXEL_LIST *pixel_list)
{
  if (pixel_list == NULL)
    return;
```

```
  if (pixel_list->pixel_array != NULL)
    free (pixel_list->pixel_array);

  pixel_list->pixel_array = NULL;
  free (pixel_list);
}

/*****************************************************************************
 *                          free_pixel_array
 *
 *
 *****************************************************************************/

void free_pixel_array (PIXEL_LIST *pixel_list)
{
  register int i;

  for (i = 0; i < pixel_list->num_pixels; i++)
    free (pixel_list->pixel_array[i]);

  free (pixel_list->pixel_array);

  pixel_list->num_pixels = 0;
  pixel_list->pixel_array = NULL;
}


/*****************************************************************************
 *                       same_row_and_sector_filter
 *
 *
 *****************************************************************************/

PIXEL_LIST* same_row_and_sector_filter (PIXEL_LIST *pixel_list,
                                        PIXEL *base_pixel)
{
  register int i;
  int sector = base_pixel->sector;
  int row = base_pixel->row;
  PIXEL *current_pixel;
  PIXEL_LIST *filtered_pixels = create_pixel_list();

  for (i = 0; i < pixel_list->num_pixels; i++)
    {
      current_pixel = pixel_list->pixel_array[i];

      if (current_pixel->sector == sector && current_pixel->row == row)
          add_to_pixel_list (filtered_pixels, current_pixel);
    }

  return filtered_pixels;
}


/*****************************************************************************
 *                       sort_pixel_list_by_adc
 *
 * uses shell sort
 *
 *****************************************************************************/

void sort_pixel_list_by_adc (PIXEL_LIST *pixel_list)
{
  register int i, j, inc;
  PIXEL *temp_pixel;
  PIXEL **pixel_array = pixel_list->pixel_array;

  inc = 1;

  do
```

```
  {
   inc *= 3;
   inc++;
  } while (inc <= pixel_list->num_pixels);

 do
  {
   inc /= 3;

   for (i = inc + 1; i <= pixel_list->num_pixels; i++)
       {
        temp_pixel = pixel_array[i - 1];
        j = i;

        while (pixel_array[j - inc - 1]->adc < temp_pixel->adc)
         {
          pixel_array[j - 1] = pixel_array[j - inc - 1];
          j -= inc;
          if (j <= inc) break;
         }

        pixel_array[j - 1] = temp_pixel;
       }

  } while (inc > 1);
}



/*****************************************************************************
 *                        write_pixel_data_to_file
 *
 *
 *****************************************************************************/

void write_pixel_data_to_file (FILE *file_ptr, PIXEL_LIST *pixel_list)
{
 register int i;
 PIXEL *current_pixel;

 for (i = 0; i < pixel_list->num_pixels; i++)
  {
   current_pixel = pixel_list->pixel_array[i];

   fprintf (file_ptr, "%d\t%d\n",
            current_pixel->found_tid, current_pixel->true_tid);

   /* fprintf (file_ptr, "%d\t%lf\t%lf\t%lf\n", found_tid,
               current_pixel->x, current_pixel->y, current_pixel->z); */
  }
}
```

B.7 Cluster and Cluster List Functions

cluster.h

```
CLUSTER* create_cluster (PIXEL_LIST *pixel_list);

void add_pixel_to_cluster (CLUSTER* cluster, PIXEL* pixel);

void unparse_cluster (CLUSTER* cluster);

void free_cluster (CLUSTER* cluster);

CLUSTER_LIST* create_cluster_list (void);

void add_to_cluster_list (CLUSTER_LIST* cluster_list, CLUSTER* new_cluster);

void append_cluster_list (CLUSTER_LIST *master_cluster_list,
                          CLUSTER_LIST *new_cluster_list);

void sort_cluster_list_by_rho (CLUSTER_LIST *cluster_list);

void unparse_cluster_list (CLUSTER_LIST* cluster_list);

void free_cluster_list (CLUSTER_LIST* cluster_list);

void free_cluster_array (CLUSTER_LIST *cluster_list);

void unparse_adc_matrix (PIXEL ***pixel_matrix, int n, int m,
                         PIXEL *chosen_pixel, PIXEL_LIST *cluster_pixels);
```

cluster.c

```
#include <stdio.h>
#include "global.h"
#include "centroid.h"
#include "pixel.h"
#include "track.h"
#include "cluster.h"


/*****************************************************************************
 *                        create_cluster
 *
 *
 *****************************************************************************/

CLUSTER* create_cluster (PIXEL_LIST *pixel_list)
{
  register int i;
  PIXEL *current_pixel;
  CLUSTER *cluster = (CLUSTER*) malloc (sizeof (CLUSTER));

  cluster->claimed_by_track = FALSE;
  cluster->track_adc_weight = 0;
  cluster->centroid = NULL;
  cluster->pixel_list = pixel_list;
  cluster->claiming_tracks = create_track_list();

  cluster->total_adc_weight = 0;

  if (pixel_list == NULL)
    return cluster;

  for (i = 0; i < pixel_list->num_pixels; i++)
    {
      current_pixel = pixel_list->pixel_array[i];
      current_pixel->cluster = cluster;
      cluster->total_adc_weight += current_pixel->adc;
    }
```

```
  calculate_centroid (cluster);

  return cluster;
}


/*****************************************************************************
 *                        add_pixel_to_cluster
 *
 *
 *****************************************************************************/

void add_pixel_to_cluster (CLUSTER *cluster, PIXEL *pixel)
{
  add_to_pixel_list (cluster->pixel_list, pixel);
}


/*****************************************************************************
 *                          unparse_cluster
 *
 *
 *****************************************************************************/

void unparse_cluster (CLUSTER *cluster)
{
  int i;
  TRACK_LIST* track_list;

  if (cluster == NULL)
    return;

  if (cluster->pixel_list != NULL)
    {
      printf("Number of pixels in cluster:  %d\n",
             cluster->pixel_list->num_pixels);

      /* Temporary:
      for (i = 0;  i < cluster->pixel_list->num_pixels; i++)
          unparse_pixel (cluster->pixel_list->pixel_array[i]);
          */
    }

  if (cluster->centroid != NULL)
    unparse_centroid (cluster->centroid);

  if (cluster->claiming_tracks == NULL)
    {
      printf ("claiming_tracks == NULL\n");
      printf ("\n\n");
      return;
    }

  track_list = cluster->claiming_tracks;

  printf("Number of tracks claiming this cluster:  %d\n",
         track_list->num_tracks);

  printf("Track ID's:");

  for (i = 0; i < track_list->num_tracks; i++)
    printf(" %d", track_list->track_array[i]->found_tid);

  printf("\n\n");
}


/*****************************************************************************
 *                          free_cluster
 *
```

```
 *
 ************************************************************************/

void free_cluster (CLUSTER *cluster)
{
 register int i;

 if (cluster == NULL)
  return;


 if (cluster->centroid != NULL)
  free_centroid (cluster->centroid);

 /* Free actual pixel information */
 if (cluster->pixel_list != NULL)
  {
   free_pixel_array (cluster->pixel_list);
   free_pixel_list (cluster->pixel_list);
   cluster->pixel_list = NULL;
  }

 /* Do not free actual track information */
 if (cluster->claiming_tracks != NULL)
  free_track_list (cluster->claiming_tracks);
 cluster->claiming_tracks = NULL;

 free (cluster);
}



/************************************************************************
 *                      create_cluster_list
 *
 *
 ************************************************************************/

CLUSTER_LIST* create_cluster_list (void)
{
 CLUSTER_LIST *cluster_list = (CLUSTER_LIST*) malloc (sizeof (CLUSTER_LIST));

 cluster_list->num_clusters = 0;
 cluster_list->cluster_array = NULL;

 return cluster_list;
}



/************************************************************************
 *                      add_to_cluster_list
 *
 *
 ************************************************************************/

void add_to_cluster_list (CLUSTER_LIST *cluster_list, CLUSTER *new_cluster)
{
 int num_clusters = cluster_list->num_clusters;

 if (new_cluster == NULL)
  return;

 cluster_list->cluster_array =
  (CLUSTER**) realloc ((CLUSTER**) cluster_list->cluster_array,
                       (num_clusters + 1) * sizeof (CLUSTER*));

 cluster_list->cluster_array[num_clusters] = new_cluster;
 cluster_list->num_clusters++;
}


/************************************************************************
```

```
 *                        append_cluster_list
 *
 * Appends the clusters contained in new_cluster_list to
 * master_cluster_list.
 *
 ****************************************************************************/
                                                                          200
void append_cluster_list (CLUSTER_LIST *master_cluster_list,
                          CLUSTER_LIST *new_cluster_list)
{
 register int i;
 int master_num_clusters, total_num_clusters;
 CLUSTER **master_cluster_array;
 CLUSTER **new_cluster_array;

 total_num_clusters =
  master_cluster_list->num_clusters + new_cluster_list->num_clusters;     210

 master_cluster_list->cluster_array =
  (CLUSTER**) realloc ((CLUSTER**) master_cluster_list->cluster_array,
                       total_num_clusters * sizeof (CLUSTER*));

 master_cluster_array = master_cluster_list->cluster_array;
 new_cluster_array = new_cluster_list->cluster_array;

 master_num_clusters = master_cluster_list->num_clusters;
 for(i = 0; i < new_cluster_list->num_clusters; i++)                      220
  master_cluster_array[master_num_clusters + i] = new_cluster_array[i];

 master_cluster_list->num_clusters = total_num_clusters;
}



/****************************************************************************
 *                        sort_cluster_list_by_rho                          230
 *
 * uses shell sort
 *
 ****************************************************************************/

void sort_cluster_list_by_rho (CLUSTER_LIST *cluster_list)
{
 register int i, j, inc;
 CLUSTER **cluster_array = cluster_list->cluster_array;
 CLUSTER *temp_cluster;                                                   240

 inc = 1;

 do
  {
   inc *= 3;
   inc++;
  } while (inc <= cluster_list->num_clusters);
                                                                          250
 do
  {
   inc /= 3;

   for (i = inc + 1; i <= cluster_list->num_clusters; i++)
        {
         temp_cluster = cluster_array[i - 1];
         j = i;

         while (cluster_array[j - inc - 1]->centroid->rho <               260
                temp_cluster->centroid->rho)
          {
           cluster_array[j - 1] = cluster_array[j - inc - 1];
           j -= inc;
           if (j <= inc) break;
          }

         cluster_array[j - 1] = temp_cluster;
        }
```

```
  } while (inc > 1);
}


/*************************************************************************
 *                        unparse_cluster_list
 *
 *
 *************************************************************************/

void unparse_cluster_list (CLUSTER_LIST *cluster_list)
{
 int i;

 if (cluster_list == NULL)
  return;

 printf("Number of clusters in this cluster list:  %d\n",
         cluster_list->num_clusters);

 for (i = 0; i < cluster_list->num_clusters; i++)
  unparse_cluster (cluster_list->cluster_array[i]);
}


/*************************************************************************
 *                        free_cluster_list
 *
 *
 *************************************************************************/

void free_cluster_list (CLUSTER_LIST *cluster_list)
{
 if (cluster_list == NULL)
  return;

 if (cluster_list->cluster_array != NULL)
  free (cluster_list->cluster_array);

 cluster_list->cluster_array = NULL;
 free (cluster_list);
}


/*************************************************************************
 *                        free_cluster_array
 *
 *
 *************************************************************************/

void free_cluster_array (CLUSTER_LIST *cluster_list)
{
 register int i;

 if (cluster_list == NULL)
  return;

 if (cluster_list->cluster_array == NULL)
  return;

 for (i = 0; i < cluster_list->num_clusters; i++)
  free_cluster (cluster_list->cluster_array[i]);

 free (cluster_list->cluster_array);

 cluster_list->num_clusters = 0;
 cluster_list->cluster_array = NULL;
}
```

```
/*****************************************************************************
 *                          unparse_adc_matrix                                  350
 *
 * pixel_array is an n x m matrix of pixel pointers
 *
 *****************************************************************************/

void unparse_adc_matrix (PIXEL ***pixel_matrix, int n, int m,
                         PIXEL *chosen_pixel, PIXEL_LIST *cluster_pixels)
{
 register int i, j, k;
 int found;                                                                   360

 printf ("matrix of pixel adc values:\n\n");

 for (i = 0; i < n; i++)
  {
   for (j = 0; j < m; j++)
       {
        if (pixel_matrix[i][j] == NULL)
          printf ("*\t");
        else if (pixel_matrix[i][j] == chosen_pixel)                          370
          printf ("#%d#\t", pixel_matrix[i][j]->adc);
        else
          {
           found = FALSE;

           for (k = 0; k < cluster_pixels->num_pixels; k++)
                if (pixel_matrix[i][j] == cluster_pixels->pixel_array[k])
                  {
                   found = TRUE;
                   break;                                                     380
                  }

           if (found)
                printf ("<%d>\t", pixel_matrix[i][j]->adc);
           else
                printf ("%d\t", pixel_matrix[i][j]->adc);
          }
       }

   printf ("\n");                                                             390
  }

}
```

B.8 Centroid and Centroid List Functions

centroid.h

```
CENTROID* create_centroid (double x, double y, double z);

void unparse_centroid (CENTROID *centroid);

void free_centroid (CENTROID *centroid);

CENTROID_LIST* create_centroid_list (void);

void add_to_centroid_list (CENTROID_LIST *centroid_list,
                           CENTROID *new_centroid);

void unparse_centroid_list (CENTROID_LIST *centroid_list);

void free_centroid_list (CENTROID_LIST *centroid_list);

void calculate_centroid (CLUSTER* cluster);
```

centroid.c

```
#include <stdio.h>
#include <math.h>
#include "global.h"
#include "centroid.h"


/*****************************************************************************
 *                          create_centroid
 *
 *
 *****************************************************************************/

CENTROID* create_centroid (double x, double y, double z)
{
  CENTROID *centroid = (CENTROID*) malloc (sizeof (CENTROID));

  centroid->x = x;
  centroid->y = y;
  centroid->z = z;
  centroid->rho = XYZ_TO_RHO (x, y, z);

  return centroid;
}


/*****************************************************************************
 *                          unparse_centroid
 *
 *
 *****************************************************************************/

void unparse_centroid (CENTROID *centroid)
{
  printf("rho:  (x, y, z) = %lf:  (%lf, %lf, %lf)\n",
         centroid->rho, centroid->x, centroid->y, centroid->z);
}


/*****************************************************************************
 *                          free_centroid
 *
 *
```

```
 ***************************************************************************/

void free_centroid (CENTROID *centroid)                                        50
{
  if (centroid != NULL)
    free (centroid);
}


/***************************************************************************
 *                        create_centroid_list                              60
 *
 *
 ***************************************************************************/

CENTROID_LIST* create_centroid_list (void)
{
  CENTROID_LIST *centroid_list =
    (CENTROID_LIST*) malloc (sizeof (CENTROID_LIST));

  centroid_list->num_centroids = 0;                                            70
  centroid_list->centroid_array = NULL;

  return centroid_list;
}


/***************************************************************************
 *                        add_to_centroid_list                              80
 *
 *
 ***************************************************************************/

void add_to_centroid_list (CENTROID_LIST *centroid_list,
                           CENTROID *new_centroid)
{
  int num_centroids = centroid_list->num_centroids;

  centroid_list->centroid_array =                                              90
    (CENTROID**) realloc ((CENTROID**) centroid_list->centroid_array,
                          (num_centroids + 1) * sizeof (CENTROID*));

  centroid_list->centroid_array[num_centroids] = new_centroid;
  centroid_list->num_centroids++;
}


                                                                              100
/***************************************************************************
 *                        unparse_centroid_list
 *
 *
 ***************************************************************************/

void unparse_centroid_list (CENTROID_LIST *centroid_list)
{
  int i;
                                                                              110
  printf ("Centroid List (%d centroids):\n", centroid_list->num_centroids);
  printf ("-----------------------------\n");

  for (i = 0; i < centroid_list->num_centroids; i++)
    unparse_centroid (centroid_list->centroid_array[i]);
}


                                                                              120
/***************************************************************************
 *                        free_centroid_list
 *
 *
```

```
 ***************************************************************************/

void free_centroid_list (CENTROID_LIST *centroid_list)
{
 register int i;

 if (centroid_list == NULL)
   return;

 if (centroid_list->centroid_array == NULL)
   {
    free (centroid_list);
    return;
   }

 for (i = 0; i < centroid_list->num_centroids; i++)
   free_centroid (centroid_list->centroid_array[i]);

 free (centroid_list->centroid_array);
 free (centroid_list);
}



/***************************************************************************
 *                          calculate_centroid
 *
 *
 ***************************************************************************/

void calculate_centroid (CLUSTER *cluster)
{
 register int i;
 PIXEL *current_pixel;
 PIXEL_LIST *pixel_list;
 double adc, sum_weights;
 double x_weighted_sum, y_weighted_sum, z_weighted_sum;


 /* Calculate the weighted mean */
 pixel_list = cluster->pixel_list;

 sum_weights = 0.0;
 x_weighted_sum = 0.0;
 y_weighted_sum = 0.0;
 z_weighted_sum = 0.0;

 for (i = 0; i < pixel_list->num_pixels; i++)
   {
    current_pixel = pixel_list->pixel_array[i];
    adc = (double) current_pixel->adc;

    sum_weights += current_pixel->adc;

    x_weighted_sum += (current_pixel->x * adc);
    y_weighted_sum += (current_pixel->y * adc);
    z_weighted_sum += (current_pixel->z * adc);
   }


 cluster->centroid = create_centroid (x_weighted_sum / sum_weights,
                                      y_weighted_sum / sum_weights,
                                      z_weighted_sum / sum_weights);
}
```

B.9 Shell and Shell Queue Functions

shell.h

```
SHELL* create_shell (double rho_min, double rho_max, PIXEL_LIST *shell_pixels);

void free_shell (SHELL* shell);

SHELL_QUEUE* create_shell_queue (void);

void push_onto_shell_queue (SHELL_QUEUE* shell_queue, SHELL* new_shell);

SHELL* pop_off_shell_queue (SHELL_QUEUE* shell_queue);                          10

void free_shell_queue (SHELL_QUEUE* shell_queue);

void cluster_unclustered_pixels (SHELL *shell);

void place_cluster_in_shell_queue (SHELL_QUEUE *shell_queue,
                                   CLUSTER *new_cluster);
```

shell.c

```
#include <stdio.h>
#include "global.h"
#include "pixel.h"
#include "cluster.h"
#include "shell.h"


/*****************************************************************************
 *                          create_shell                                     10
 *
 *
 *****************************************************************************/

SHELL* create_shell (double rho_min, double rho_max, PIXEL_LIST *shell_pixels)
{
  SHELL *shell = (SHELL*) malloc (sizeof (SHELL));

  shell->rho_min = rho_min;
  shell->rho_max = rho_max;                                                     20
  shell->pixel_list = shell_pixels;
  shell->cluster_list = NULL;
  shell->next_shell = NULL;
  shell->next_shell_in_queue = NULL;

  return shell;
}


                                                                              30
/*****************************************************************************
 *                          free_shell
 *
 * Actually frees pixel data (assumes that free'ing a shell means that the
 * contained pixel data is no longer needed).
 *
 *****************************************************************************/

void free_shell (SHELL *shell)                                                40
{
  register int i, j;
  int num_freed, num_passed_down;
  double rho_min_next;
  double rho_max_next;
  int pixel_in_next_shell;   /* used as boolean */
  CLUSTER *current_cluster;
```

```
PIXEL_LIST *pixel_list;

if (shell == NULL)
  return;

if (shell->pixel_list != NULL)
  free_pixel_list (shell->pixel_list);

if (shell->cluster_list == NULL)
  return;

if (shell->next_shell == NULL)
  {
    free_cluster_array (shell->cluster_list);
    free_cluster_list (shell->cluster_list);
    return;
  }

rho_min_next = shell->next_shell->rho_min;
rho_max_next = shell->next_shell->rho_max;

num_freed = 0;
num_passed_down = 0;

for (i = 0; i < shell->cluster_list->num_clusters; i++)
  {
    current_cluster = shell->cluster_list->cluster_array[i];
    pixel_list = current_cluster->pixel_list;

    pixel_in_next_shell = FALSE;

    for (j = 0; j < pixel_list->num_pixels; j++)
        if (rho_min_next <= pixel_list->pixel_array[j]->rho &&
            pixel_list->pixel_array[j]->rho <= rho_max_next)
          {
            pixel_in_next_shell = TRUE;
            break;
          }

    if (pixel_in_next_shell)
        {
          add_to_cluster_list (shell->next_shell->cluster_list,
                               current_cluster);
          num_passed_down++;
        }
    else  /* free the cluster and its pixels */
        {
          free_cluster (current_cluster);
          num_freed++;
        }
  }

 /* Temporary: */
 printf ("clusters:  %d originally, %d freed, %d passed down\n",
          shell->cluster_list->num_clusters, num_freed, num_passed_down);

 free_cluster_list (shell->cluster_list);
}


/*************************************************************************
 *                        create_shell_queue
 *
 *
 *************************************************************************/

SHELL_QUEUE* create_shell_queue (void)
{
 SHELL_QUEUE *shell_queue = (SHELL_QUEUE*) malloc (sizeof (SHELL_QUEUE));

 shell_queue->num_shells = 0;
 shell_queue->first_shell = NULL;
```

```
  shell_queue->last_shell = NULL;

  return shell_queue;
}

                                                                          130
/*************************************************************************
 *                        push_onto_shell_queue
 *
 *
 *************************************************************************/

void push_onto_shell_queue (SHELL_QUEUE *shell_queue, SHELL *new_shell)
{
  if (new_shell == NULL)                                                  140
    return;

  new_shell->next_shell_in_queue = NULL;

  /* new_shell could be the first shell added to the queue: */
  if (shell_queue->first_shell == NULL)
    shell_queue->first_shell = new_shell;

  /* set next_shell_in_queue of the last shell in the queue to the new shell: */
  if (shell_queue->last_shell != NULL)                                    150
    shell_queue->last_shell->next_shell_in_queue = new_shell;

  /* set the last_shell of the queue to the new shell: */
  shell_queue->last_shell = new_shell;

  shell_queue->num_shells++;
}

                                                                          160
/*************************************************************************
 *                        pop_off_shell_queue
 *
 *
 *************************************************************************/

SHELL* pop_off_shell_queue (SHELL_QUEUE *shell_queue)
{
  SHELL *shell;                                                           170

  if (shell_queue == NULL)
    return NULL;

  if (shell_queue->first_shell == NULL)
    return NULL;

  shell = shell_queue->first_shell;

  /* set the second shell to be the first shell: */                       180
  shell_queue->first_shell = shell_queue->first_shell->next_shell_in_queue;

  shell_queue->num_shells--;       /* decrement number of shells in queue */

  return shell;
}

                                                                          190
/*************************************************************************
 *                        free_shell_queue
 *
 *
 *************************************************************************/

void free_shell_queue (SHELL_QUEUE *shell_queue)
{
  free (shell_queue);
}                                                                         200
```

```
/*****************************************************************************
 *                    cluster_unclustered_pixels
 *
 *
 *****************************************************************************/
                                                                                    210
void cluster_unclustered_pixels (SHELL *shell)
{
  register int i;
  int num_clusters_added;
  PIXEL *pixel;
  PIXEL_LIST *pixel_list = shell->pixel_list;
  PIXEL_LIST *cluster_pixel_list;
  CLUSTER *new_cluster;
  CLUSTER_LIST *cluster_list;
                                                                                    220
  printf ("entering cluster_unclustered_pixels...\n");  /* Temporary */

  if (shell->cluster_list == NULL)
    shell->cluster_list = create_cluster_list();

  cluster_list = shell->cluster_list;


  num_clusters_added = 0;                                                           230

  for (i = 0; i < pixel_list->num_pixels; i++)
    {
      /* already been assigned to a cluster */
      if (pixel_list->pixel_array[i]->cluster != NULL)
          continue;

      pixel = pixel_list->pixel_array[i];

      cluster_pixel_list = create_pixel_list();                                     240
      add_to_pixel_list (cluster_pixel_list, pixel);

      new_cluster = create_cluster (cluster_pixel_list);

      add_to_cluster_list (cluster_list, new_cluster);

      num_clusters_added++;
    }

                                                                                    250
  /* Temporary: */
  printf ("Found %d unclustered pixels\n", num_clusters_added);
  printf ("leaving cluster_unclustered_pixels...\n");
}



/*****************************************************************************
 *                    place_cluster_in_shell_queue                               260
 *
 *
 *****************************************************************************/

void place_cluster_in_shell_queue (SHELL_QUEUE *shell_queue,
                                   CLUSTER *new_cluster)
{
  SHELL *current_shell = shell_queue->first_shell;
  CENTROID *centroid = new_cluster->centroid;
                                                                                    270
  while (current_shell != NULL)
    {
      if (current_shell->rho_min < centroid->rho &&
          centroid->rho <= current_shell->rho_max)
        {
          add_to_cluster_list (current_shell->cluster_list, new_cluster);
          break;
```

```
        }
    current_shell = current_shell->next_shell_in_queue;
  }
}
```

B.10 Track and Track List Functions

track.h

```
#ifndef _TRACK_
#define _TRACK_

#include "entry.h"

class Track
{
  int _tid;
  int _num_entries;
  Entry* _entry_array;

public:
  Track();
  Track(int tid);
  ~Track();
  void process_pixel(int tid);
  void add_entry(int tid);
  void sort_entries();
  int total_num_pixels();
  int get_tid();
  void unparse(int true_found);
};

#endif _TRACK_
```

track.c

```
#include <stdio.h>
#include <math.h>

#include "global.h"
#include "centroid.h"
#include "cluster.h"
#include "pixel.h"
#include "track.h"

/*****************************************************************************
 *                          create_track
 *
 *
 *****************************************************************************/

TRACK* create_track (void)
{
  TRACK *track = (TRACK*) malloc (sizeof (TRACK));

  track->found_tid = 0;
  track->num_shells_wo_new_centroid = 0;
  track->centroid_list = create_centroid_list();
  track->claimed_clusters = NULL;

  return track;
}


/*****************************************************************************
 *                       add_centroid_to_track
 *
 *
 *****************************************************************************/

void add_centroid_to_track (TRACK *track, CENTROID *new_centroid)
{
```

```
  add_to_centroid_list (track->centroid_list, new_centroid);
}


/*****************************************************************************
 *                          unparse_track
 *
 *
 *****************************************************************************/

void unparse_track (TRACK *track)
{
  printf("found_tid:   %d\n", track->found_tid);
  printf("num_shells_wo_new_centroid:   %d\n",
         track->num_shells_wo_new_centroid);

  unparse_centroid_list (track->centroid_list);
  printf("\n\n");
}


/*****************************************************************************
 *                          free_track
 *
 *
 *****************************************************************************/

void free_track (TRACK *track)
{
  if (track == NULL)
    return;

  if (track->centroid_list != NULL)
    free_centroid_list (track->centroid_list);

  if (track->claimed_clusters != NULL)
    free_cluster_list (track->claimed_clusters);

  free (track);
}



/*****************************************************************************
 *                          create_track_list
 *
 *
 *****************************************************************************/

TRACK_LIST* create_track_list (void)
{
  TRACK_LIST *track_list = (TRACK_LIST*) malloc (sizeof (TRACK_LIST));
  track_list->num_tracks = 0;
  track_list->track_array = NULL;

  return track_list;
}



/*****************************************************************************
 *                          add_to_track_list
 *
 * Adds the given track (new_track) to the given track_list.
 *
 *****************************************************************************/

void add_to_track_list (TRACK_LIST *track_list, TRACK *new_track)
{
  int num_tracks = track_list->num_tracks;
```

```
  track_list->track_array =
    (TRACK**) realloc ((TRACK**) track_list->track_array,
                       (num_tracks + 1) * sizeof (TRACK*));

  track_list->track_array[num_tracks] = new_track;
  track_list->num_tracks++;
}



/*****************************************************************************
 *                        add_new_found_track
 *
 * Adds the given track (new_track) to the given track_list.  Also sets
 * the new track's found_tid.
 *
 * Modifies: new_track->found_tid
 *
 *****************************************************************************/

void add_new_found_track (TRACK_LIST *track_list, TRACK *new_track)
{
  static int found_tid = 1;      /* provides unique found_tid's */
  int num_tracks = track_list->num_tracks;

  track_list->track_array =
    (TRACK**) realloc ((TRACK**) track_list->track_array,
                       (num_tracks + 1) * sizeof (TRACK*));

  new_track->found_tid = found_tid;
  track_list->track_array[num_tracks] = new_track;
  track_list->num_tracks++;
  found_tid++;
}



/*****************************************************************************
 *                        remove_inactive_tracks
 *
 *
 *****************************************************************************/

TRACK_LIST* remove_inactive_tracks (FILE *file_ptr,
                                    TRACK_LIST *active_track_list,
                                    int no_centroid_shell_limit)
{
  register int i;
  TRACK_LIST *new_active_track_list = create_track_list();
  TRACK_LIST *inactive_track_list = create_track_list();
  TRACK *current_track;

  printf ("remove_inactive_tracks...\n");


  for (i = 0; i < active_track_list->num_tracks; i++)
    {
      current_track = active_track_list->track_array[i];

      if (current_track->num_shells_wo_new_centroid > no_centroid_shell_limit)
          {
            printf ("moving track %d to inactive\n", current_track->found_tid);
            add_to_track_list (inactive_track_list, current_track);
          }
      else
          add_to_track_list (new_active_track_list, current_track);
    }


  write_track_data_to_file (file_ptr, inactive_track_list);

  free_track_array (inactive_track_list);
  free_track_list (inactive_track_list);
```

```
  free_track_list (active_track_list);

  return new_active_track_list;
}

                                                                                200

/*************************************************************************
 *                        unparse_track_list
 *
 *
 *************************************************************************/

void unparse_track_list (TRACK_LIST *track_list)
{
  int i;                                                                        210

  if (track_list == NULL)
    return;

  printf ("Track list (%d tracks):\n", track_list->num_tracks);
  printf ("----------------------\n");

  for (i = 0; i < track_list->num_tracks; i++)
    unparse_track (track_list->track_array[i]);
}                                                                               220


/*************************************************************************
 *                        free_track_list
 *
 * Doesn't actually free the track information - only the track pointer
 * array.
 *                                                                              230
 *************************************************************************/

void free_track_list (TRACK_LIST *track_list)
{
  if (track_list == NULL)
    return;

  if (track_list->track_array != NULL)
    free (track_list->track_array);
                                                                                240
  track_list->track_array = NULL;
  free (track_list);
}


/*************************************************************************
 *                        free_track_array
 *                                                                              250
 *
 *************************************************************************/

void free_track_array (TRACK_LIST *track_list)
{
  register int i, j;
  TRACK *current_track;

  if (track_list == NULL)
    return;                                                                     260

  if (track_list->track_array == NULL)
    return;

  for (i = 0; i < track_list->num_tracks; i++)
    {
      current_track = track_list->track_array[i];

      /* Free each centroid: */
      for (j = 0; j < current_track->centroid_list->num_centroids; j++)         270
          free_centroid (current_track->centroid_list->centroid_array[j]);
```

```
    /* Free cluster list: */
    if (current_track->claimed_clusters != NULL)
        free_cluster_list (current_track->claimed_clusters);
  }

 free (track_list->track_array);
 track_list->track_array = NULL;                                          280
 track_list->num_tracks = 0;
}


/*************************************************************************
 *                          paint_clusters
 *
 * This is the only procedure that should add to a cluster's              290
 * claiming_tracks.
 *
 *************************************************************************/

void paint_clusters (PIXEL_LIST *pixel_list, TRACK *track,
                     double cluster_painting_threshold)
{
 register int i, j;
 int found;          /* used as a boolean */
 double percent_overlap;                                                  300
 PIXEL *pixel;
 CLUSTER *cluster;
 CLUSTER_LIST *temp_cluster_list = create_cluster_list();


 for (i = 0; i < pixel_list->num_pixels; i++)
  {
    pixel = pixel_list->pixel_array[i];
    cluster = pixel->cluster;
                                                                          310
    /*  Check to see if cluster has already been added: */
    found = FALSE;

    for (j = 0; j < temp_cluster_list->num_clusters; j++)
        if (temp_cluster_list->cluster_array[j] == cluster)
          {
            found = TRUE;
            break;
          }                                                               320

    /* Only add the cluster if it hasn't already been added */
    if (! found)
        {
          add_to_cluster_list (temp_cluster_list, cluster);
          cluster->track_adc_weight = 0;     /* set weight to zero */
        }

    cluster->track_adc_weight += pixel->adc;
  }                                                                       330

 /* Temporary: */
 printf ("number clusters overlapped in search cone:  %d\n",
          temp_cluster_list->num_clusters);


 /* Now check all temp_clusters against cluster-painting threshold */
 for (i = 0; i < temp_cluster_list->num_clusters; i++)
  {                                                                       340
    cluster = temp_cluster_list->cluster_array[i];

    percent_overlap =
        (double) cluster->track_adc_weight /
        (double) cluster->total_adc_weight;

    /*
```

```
    printf ("num pixels in cluster: %d\n", cluster->pixel_list->num_pixels);
    printf ("total_adc_weight: %d\n", cluster->total_adc_weight);
    printf ("percent_overlap: %lf\n\n", percent_overlap);
    */

    if (percent_overlap >= cluster_painting_threshold)
        {
         cluster->claimed_by_track = TRUE;
         add_to_track_list (cluster->claiming_tracks, track);
         add_to_cluster_list (track->claimed_clusters, cluster);
        }
   }

 free_cluster_list (temp_cluster_list);
}


/*****************************************************************************
 *                    calculate_new_track_centroid
 *
 * Assumes that the track claimed at least one cluster.
 *
 *****************************************************************************/

CENTROID* calculate_new_track_centroid (TRACK *track)
{
 register int i;
 double adjusted_total_adc_weight, sum_weights;
 double x_weighted_sum, y_weighted_sum, z_weighted_sum;
 CLUSTER *cluster;
 CENTROID *centroid;
 CLUSTER_LIST *claimed_clusters = track->claimed_clusters;

 /* Calculate the weighted mean */
 sum_weights = 0.0;
 x_weighted_sum = 0.0;
 y_weighted_sum = 0.0;
 z_weighted_sum = 0.0;


 for (i = 0; i < claimed_clusters->num_clusters; i++)
  {
   cluster = claimed_clusters->cluster_array[i];
   centroid = cluster->centroid;

   /* Divide the cluster's total_adc_weight by the number of
        tracks claiming that cluster: */
   adjusted_total_adc_weight =
       (double) cluster->total_adc_weight /
       (double) cluster->claiming_tracks->num_tracks;

   sum_weights += adjusted_total_adc_weight;
   x_weighted_sum += (centroid->x * adjusted_total_adc_weight);
   y_weighted_sum += (centroid->y * adjusted_total_adc_weight);
   z_weighted_sum += (centroid->z * adjusted_total_adc_weight);
  }

 return (create_centroid (x_weighted_sum / sum_weights,
                          y_weighted_sum / sum_weights,
                          z_weighted_sum / sum_weights));
}


/*****************************************************************************
 *                    fit_line_to_centroids
 *
 * Equation of line passing through point P0 = (x0, y0, z0) and parallel
 * to the vector V=a*i + b*j + c*k:
 *
 *          x = x0 + a*t,  y = y0 + b*t,  z = z0 + c*t
 *
```

```
 * Cartesian:        (x - x0)/a = (y - y0)/b = (z - z0)/c
 *
 ****************************************************************************/

void fit_line_to_centroids (CENTROID_LIST *centroid_list,
                            int num_linefit_centroids,
                            double *x0, double *y0, double *z0,
                            double *vx, double *vy, double *vz)
{
  int num_centroids = centroid_list->num_centroids;
  CENTROID *centroid1, *centroid2;

  /* For now, use only the last two centroids of the track: */
  centroid1 = centroid_list->centroid_array[num_centroids - 2];
  centroid2 = centroid_list->centroid_array[num_centroids - 1];

  *x0 = centroid2->x;
  *y0 = centroid2->y;
  *z0 = centroid2->z;

  *vx = centroid2->x - centroid1->x;
  *vy = centroid2->y - centroid1->y;
  *vz = centroid2->z - centroid2->z;
}


/****************************************************************************
 *                    calculate_search_cone_angle
 *
 *
 ****************************************************************************/

double calculate_search_cone_angle (double search_cone_initial_angle,
                                    double search_cone_angle_taper_coef,
                                    int num_centroids)
{
  /* Subtract 3 from num_centroids to account for proto-track centroids: */
  double coef = search_cone_angle_taper_coef * (double) (num_centroids - 3);

  return (search_cone_initial_angle * exp (-coef));
}


/****************************************************************************
 *                    calculate_bounding_box
 *
 *
 ****************************************************************************/

int calculate_bounding_box (double rho_min, double rho_max,
                            double x0, double y0, double z0,
                            double vx, double vy, double vz,
                            double search_cone_base_radius,
                            double search_cone_angle,
                            double *x_min, double *x_max, double *y_min,
                            double *y_max, double *z_min, double *z_max,
                            double *px0, double *py0, double *pz0)
{
  double px1, py1, pz1, px2, py2, pz2;
  double length, search_cone_rim_radius;
  double v_mag, dx, dy, dz;

  /* Calculate intersection (px1, py1, pz1) with middle of previous shell: */
  if (! nearest_sphere_intersection_point (rho_max,
                                           x0, y0, z0, vx, vy, vz,
                                           &px1, &py1, &pz1))
    {
      printf ("no intersection point!\n");
      return FALSE;
    }
```

```
                                                                          500
  /* Calculate intersection (px2, py2, pz2) with middle of next shell: */
  if (! nearest_sphere_intersection_point (rho_min,
                                           x0, y0, z0, vx, vy, vz,
                                           &px2, &py2, &pz2))
    {
      printf ("no intersection point!\n");
      return FALSE;
    }

  length = DISTANCE (px1,py1,pz1, px2,py2,pz2);                           510

  search_cone_rim_radius =
    search_cone_base_radius + (length * tan (search_cone_angle));


  v_mag = MAG (vx, vy, vz);
  if (v_mag < 0.001) v_mag = 0.001;

  dx = search_cone_base_radius * sin (acos (vx / v_mag));
  dy = search_cone_base_radius * sin (acos (vy / v_mag));                 520
  dz = search_cone_base_radius * sin (acos (vz / v_mag));

  *x_min = px1 - dx;  *x_max = px1 + dx;
  *y_min = py1 - dy;  *y_max = py1 + dy;
  *z_min = pz1 - dz;  *z_max = pz1 + dz;


  dx = search_cone_rim_radius * sin (acos (vx / v_mag));
  dy = search_cone_rim_radius * sin (acos (vy / v_mag));
  dz = search_cone_rim_radius * sin (acos (vz / v_mag));                  530

  *x_min = (px2 - dx < *x_min) ? px2 - dx : *x_min;
  *x_max = (px2 + dx > *x_max) ? px2 + dx : *x_max;
  *y_min = (py2 - dy < *y_min) ? py2 - dy : *y_min;
  *y_max = (py2 + dy > *y_max) ? py2 + dy : *y_max;
  *z_min = (pz2 - dz < *z_min) ? pz2 - dz : *z_min;
  *z_max = (pz2 + dz > *z_max) ? pz2 + dz : *z_max;

  *px0 = px1;
  *py0 = py1;                                                             540
  *pz0 = pz1;

  return TRUE;
}



/*****************************************************************************
 *                  nearest_sphere_intersection_point                        550
 *
 *
 *****************************************************************************/

int nearest_sphere_intersection_point (double r,
                                       double x0, double y0, double z0,
                                       double vx, double vy, double vz,
                                       double *px, double *py, double *pz)
{
  double px1, py1, pz1;                                                   560
  double px2, py2, pz2;
  double sum1, sum2, den;

  if (vy < 0.001)  vy = 0.001;

  sum1 = 2 * (vx*vx) * y0 / (vy*vy);
  sum1 -= 2 * vx * x0 / vy;
  sum1 += 2 * (vz*vz) * y0 / (vy*vy);
  sum1 -= 2 * vz * z0 / vy;
                                                                          570
  sum2 = -(vx*vx) * (z0*z0);
  sum2 += (vx*vx) * (r*r);
  sum2 -= (vx*vx) * (y0*y0);
  sum2 -= (vz*vz) * (y0*y0);
  sum2 -= (vz*vz) * (x0*x0);
```

```
  sum2 += (vz*vz) * (r*r);
  sum2 -= (x0*x0) * (vy*vy);
  sum2 -= (z0*z0) * (vy*vy);
  sum2 += (r*r) * (vy*vy);
  sum2 += 2.0 * vx * x0 * vz * z0;                                          580
  sum2 += 2.0 * vx * y0 * x0 * vy;
  sum2 += 2.0 * vz * y0 * z0 * vy;

  if (sum2 <= 0.0)
    return FALSE;     /* no intersection */

  den = 1 + (vx*vx) / (vy*vy) + (vz*vz) / (vy*vy);

  py1 = (sum1 + 2.0 * sqrt (sum2) / vy) / (2.0 * den);
  py2 = (sum1 - 2.0 * sqrt (sum2) / vy) / (2.0 * den);                      590

  px1 = (py1 - y0) * vx / vy + x0;
  pz1 = (py1 - y0) * vz / vy + z0;

  px2 = (py2 - y0) * vx / vy + x0;
  pz2 = (py2 - y0) * vz / vy + z0;

  if (DISTANCE (x0, y0, z0, px1, py1, pz1) <                                600
      DISTANCE (x0, y0, z0, px2, py2, pz2))
    {
      *px = px1;
      *py = py1;
      *pz = pz1;
    }
  else
    {
      *px = px2;
      *py = py2;                                                            610
      *pz = pz2;
    }

  /* Temporary: */
  /*
  printf ("(px1, py1, pz1) = (%lf, %lf, %lf)\n", px1, py1, pz1);
  printf ("(px2, py2, pz2) = (%lf, %lf, %lf)\n", px2, py2, pz2);
  printf ("(px, py, pz) = (%lf, %lf, %lf)\n", *px, *py, *pz);
  */
                                                                            620
  return TRUE;      /* successfully found intersection point */
}



/*****************************************************************************
 *                          label_track_pixels
 *                                                                           630
 * Modifies: pixel->found_tid for each pixel in track's cluster list.
 *
 *****************************************************************************/

void label_track_pixels (TRACK *track)
{
  register int i, j;
  PIXEL_LIST* pixel_list;

  for (i = 0; i < track->claimed_clusters->num_clusters; i++)               640
    {
      pixel_list = track->claimed_clusters->cluster_array[i]->pixel_list;

      for (j = 0; j < pixel_list->num_pixels; j++)
          pixel_list->pixel_array[j]->found_tid = track->found_tid;
    }
}

                                                                            650
```

```
/*****************************************************************************
 *                    filter_for_search_cone_pixels
 *
 *
 *****************************************************************************/

PIXEL_LIST* filter_for_search_cone_pixels (PIXEL_LIST *pixel_list,
                                           double px0, double py0, double pz0,
                                           double vx, double vy, double vz,
                                           double search_cone_base_radius,
                                           double search_cone_angle)
{
 register int i;
 double p_mag, v_mag;
 double px, py, pz;
 double h, cone_radius, cos_angle, angle;
 PIXEL *pixel;
 PIXEL_LIST *filtered_pixels = create_pixel_list();

 v_mag = MAG (vx, vy, vz);
 if (v_mag < 0.001) v_mag = 0.001;


 for(i = 0; i < pixel_list->num_pixels; i++)
  {
   pixel = pixel_list->pixel_array[i];

   /* translate to origin */
   px = pixel->x - px0;
   py = pixel->y - py0;
   pz = pixel->z - pz0;

   p_mag = MAG (px, py, pz);
   if (p_mag < 0.001) p_mag = 0.001;

   cos_angle = DOT_PRODUCT (vx, vy, vz, px, py, pz) / (v_mag * p_mag);

   if (cos_angle > 1.0) cos_angle = 1.0;
   if (cos_angle < -1.0) cos_angle = -1.0;
   angle = acos (cos_angle);

   h = p_mag * cos (angle);
   cone_radius = search_cone_base_radius + (h * tan (search_cone_angle));

   if (h >= 0.0 && p_mag * sin (angle) <= cone_radius)
     add_to_pixel_list (filtered_pixels, pixel);
  }

 return filtered_pixels;
}



/*****************************************************************************
 *                    write_track_data_to_file
 *
 *
 *****************************************************************************/

void write_track_data_to_file (FILE *file_ptr, TRACK_LIST *track_list)
{
 register int i, j;
 TRACK *current_track;
 CENTROID *centroid;

 for (i = 0; i < track_list->num_tracks; i++)
  {
   current_track = track_list->track_array[i];

   if (current_track->centroid_list->num_centroids <= 3)
       continue;       /* Don't write to file if only a proto-track */

   for (j = 0; j < current_track->centroid_list->num_centroids; j++)
       {
```

```
            centroid = current_track->centroid_list->centroid_array[j];

            fprintf (file_ptr, "%d\t%lf\t%lf\t%lf\n", current_track->found_tid,
                     centroid->x, centroid->y, centroid->z);
          }
    }
}


/*****************************************************************************
 *                    filter_for_search_cone_clusters
 *
 *
 *****************************************************************************/

CLUSTER_LIST* filter_for_search_cone_clusters (CLUSTER_LIST *cluster_list,
                                               double px0, double py0, double pz0,
                                               double vx, double vy, double vz,
                                               double search_cone_base_radius,
                                               double search_cone_angle)
{
  register int i;
  double p_mag, v_mag;
  double px, py, pz;
  double h, cone_radius, cos_angle, angle;
  CLUSTER *cluster;
  CENTROID *centroid;
  CLUSTER_LIST *filtered_clusters = create_cluster_list();

  v_mag = MAG (vx, vy, vz);
  if (v_mag < 0.001) v_mag = 0.001;

  for(i = 0; i < cluster_list->num_clusters; i++)
    {
      cluster = cluster_list->cluster_array[i];
      centroid = cluster->centroid;

      /* translate to origin */
      px = centroid->x - px0;
      py = centroid->y - py0;
      pz = centroid->z - pz0;

      p_mag = MAG (px, py, pz);
      if (p_mag < 0.001) p_mag = 0.001;

      cos_angle = DOT_PRODUCT (vx, vy, vz, px, py, pz) / (v_mag * p_mag);

      if (cos_angle > 1.0) cos_angle = 1.0;
      if (cos_angle < -1.0) cos_angle = -1.0;
      angle = acos (cos_angle);

      h = p_mag * cos (angle);
      cone_radius = search_cone_base_radius + (h * tan (search_cone_angle));

      if (h >= 0.0 && p_mag * sin (angle) <= cone_radius)
        add_to_cluster_list (filtered_clusters, cluster);
    }

  return filtered_clusters;
}
```

B.11 Octant Tree Functions

octant_tree.c

```
OCTANT_TREE_NODE* create_octant_tree_node (double x_min, double x_max,
                                           double y_min, double y_max,
                                           double z_min, double z_max);

OCTANT_TREE_NODE* build_octant_tree (PIXEL_LIST* pixel_list);

void add_pixel_to_tree (OCTANT_TREE_NODE* node, PIXEL* pixel);

void extract_pixels (OCTANT_TREE_NODE* node, PIXEL_LIST* found_pixels,                10
                     double bb_x_min, double bb_x_max, double bb_y_min,
                     double bb_y_max, double bb_z_min, double bb_z_max);

void verify_extract_pixels (PIXEL_LIST* tree_pixels, PIXEL_LIST* found_pixels,
                            double bb_x_min, double bb_x_max, double bb_y_min,
                            double bb_y_max, double bb_z_min, double bb_z_max);

void recursively_add_pixels (OCTANT_TREE_NODE* node,
                             PIXEL_LIST* found_pixels);
                                                                                      20
OCTANT_TREE_NODE* next_octant (OCTANT_TREE_NODE* node, int* sub_octant_num,
                               double x, double y, double z);

void place_new_octant_tree_node (OCTANT_TREE_NODE* node, int sub_octant_num,
                                 OCTANT_TREE_NODE* new_node);

void sub_octant_boundaries (OCTANT_TREE_NODE* node, int sub_octant_num,
                            double *x_min, double *x_max,
                            double *y_min, double *y_max,
                            double *z_min, double *z_max);                            30

void unparse_octant_tree (OCTANT_TREE_NODE* node);

void free_octant_tree (OCTANT_TREE_NODE* node);

int number_of_nodes (OCTANT_TREE_NODE* node);

int number_of_nodes_real (OCTANT_TREE_NODE* node, int new);

int num_pixels_in_tree (OCTANT_TREE_NODE* node);                                      40

int num_pixels_in_tree_real (OCTANT_TREE_NODE* node, int new);

void find_min_max_coordinates (PIXEL_LIST *pixel_list,
                               double *x_min, double *x_max, double *y_min,
                               double *y_max, double *z_min, double *z_max);
```

octant_tree.h

```
#include <stdio.h>
#include "global.h"
#include "pixel.h"
#include "octant_tree.h"


/*****************************************************************************
 *                    create_octant_tree_node
 *                                                                            10
 *
 *****************************************************************************/

OCTANT_TREE_NODE* create_octant_tree_node (double x_min, double x_max,
                                           double y_min, double y_max,
                                           double z_min, double z_max)
{
```

```
 OCTANT_TREE_NODE *octant_tree_node =
  (OCTANT_TREE_NODE*) malloc (sizeof (OCTANT_TREE_NODE));

 octant_tree_node->oct1 = NULL;
 octant_tree_node->oct2 = NULL;
 octant_tree_node->oct3 = NULL;
 octant_tree_node->oct4 = NULL;
 octant_tree_node->oct5 = NULL;
 octant_tree_node->oct6 = NULL;
 octant_tree_node->oct7 = NULL;
 octant_tree_node->oct8 = NULL;

 octant_tree_node->x_min = x_min;
 octant_tree_node->x_max = x_max;
 octant_tree_node->y_min = y_min;
 octant_tree_node->y_max = y_max;
 octant_tree_node->z_min = z_min;
 octant_tree_node->z_max = z_max;

 octant_tree_node->pixel = NULL;

 return octant_tree_node;
}


/*****************************************************************************
 *                        build_octant_tree
 *
 *
 *****************************************************************************/

OCTANT_TREE_NODE* build_octant_tree (PIXEL_LIST *pixel_list)
{
 register int i;
 double x_min, x_max, y_min, y_max, z_min, z_max;
 OCTANT_TREE_NODE *octant_tree_root;


 if (pixel_list == NULL)
  return NULL;

 find_min_max_coordinates (pixel_list,
                           &x_min, &x_max, &y_min, &y_max, &z_min, &z_max);

 octant_tree_root =
  create_octant_tree_node (x_min, x_max, y_min, y_max, z_min, z_max);

 for (i = 0; i < pixel_list->num_pixels; i++)
  add_pixel_to_tree (octant_tree_root, pixel_list->pixel_array[i]);

 return octant_tree_root;
}


/*****************************************************************************
 *                        add_pixel_to_tree
 *
 *
 *****************************************************************************/

void add_pixel_to_tree (OCTANT_TREE_NODE* node, PIXEL* pixel)
{
 OCTANT_TREE_NODE *sub_octant_ptr;
 PIXEL *temp_pixel;
 int sub_octant_num;
 double x_min, x_max, y_min, y_max, z_min, z_max;


  /* Must insert an interior node and decide which octant the pixel
     belongs to. */

 if (node->pixel != NULL)          /* node is a leaf */
  {
```

```
    temp_pixel = node->pixel;
    node->pixel = NULL;          /* node is no longer a leaf */

    /* Call recursively on temp_pixel and pixel so they can be placed: */
    add_pixel_to_tree (node, temp_pixel);
    add_pixel_to_tree (node, pixel);                                    100
    return;
   }

  /* node is an interior node: */

  sub_octant_ptr =
   next_octant (node, &sub_octant_num, pixel->x, pixel->y, pixel->z);

                                                                        110
  if (sub_octant_ptr == NULL)
   {
    sub_octant_boundaries (node, sub_octant_num, &x_min, &x_max,
                           &y_min, &y_max, &z_min, &z_max);

    /* Create new leaf: */
    sub_octant_ptr =
        create_octant_tree_node (x_min, x_max, y_min, y_max, z_min, z_max);

    sub_octant_ptr->pixel = pixel;                                      120

    /* Attach the new leaf to the tree: */
    place_new_octant_tree_node (node, sub_octant_num, sub_octant_ptr);

    return;
   }

  /* Otherwise, recursively traverse tree to find insertion point: */
  add_pixel_to_tree (sub_octant_ptr, pixel);
}                                                                       130



/*************************************************************************
 *                          extract_pixels
 *
 * Extract the pixels from the octant tree which lie in the desired
 * conical search volume.
 *                                                                       140
 *************************************************************************/

void extract_pixels (OCTANT_TREE_NODE *octant, PIXEL_LIST *found_pixels,
                     double bb_x_min, double bb_x_max, double bb_y_min,
                     double bb_y_max, double bb_z_min, double bb_z_max)
{
  int axis_overlap_count;
  double oct_x_min, oct_x_max, oct_y_min, oct_y_max, oct_z_min, oct_z_max;
  PIXEL *pixel;
                                                                        150
  if (octant == NULL)
   return;

  oct_x_min = octant->x_min;  oct_x_max = octant->x_max;
  oct_y_min = octant->y_min;  oct_y_max = octant->y_max;
  oct_z_min = octant->z_min;  oct_z_max = octant->z_max;

  pixel = octant->pixel;

                                                                        160
  /* If reach a leaf, test the pixel to see if it should be included */
  if (pixel != NULL)
   {
    if (bb_x_min <= pixel->x && pixel->x <= bb_x_max &&
        bb_y_min <= pixel->y && pixel->y <= bb_y_max &&
        bb_z_min <= pixel->z && pixel->z <= bb_z_max)
       add_to_pixel_list (found_pixels, pixel);
    /* check pixel against search volume */

    return;                                                             170
   }
```

```
/* Otherwise, must be an interior node... */

/* If octant is completely enclosed by the bounding box, then recursively
   add all pixels in the subregions */
if (bb_x_min <= oct_x_min && oct_x_min <= bb_x_max &&
    bb_x_min < oct_x_max && oct_x_max < bb_x_max &&
    bb_y_min <= oct_y_min && oct_y_min <= bb_y_max &&
    bb_y_min < oct_y_max && oct_y_max < bb_y_max &&
    bb_z_min <= oct_z_min && oct_z_min <= bb_z_max &&
    bb_z_min < oct_z_max && oct_z_max < bb_z_max)
  {
    recursively_add_pixels (octant, found_pixels);
    return;
  }

axis_overlap_count = 0;

/* Test for overlap along x axis: */
if (((oct_x_min <= bb_x_min && bb_x_min < oct_x_max) ||
     (oct_x_min <= bb_x_max && bb_x_max < oct_x_max)) ||
    ((bb_x_min <= oct_x_min && oct_x_min <= bb_x_max) ||
     (bb_x_min < oct_x_max && oct_x_max < bb_x_max)))
  axis_overlap_count++;

/* Test for overlap along y axis: */
 if (((oct_y_min <= bb_y_min && bb_y_min < oct_y_max) ||
     (oct_y_min <= bb_y_max && bb_y_max < oct_y_max)) ||
    ((bb_y_min <= oct_y_min && oct_y_min <= bb_y_max) ||
     (bb_y_min < oct_y_max && oct_y_max < bb_y_max)))
  axis_overlap_count++;

 /* Test for overlap along z axis: */
if (((oct_z_min <= bb_z_min && bb_z_min < oct_z_max) ||
     (oct_z_min <= bb_z_max && bb_z_max < oct_z_max)) ||
    ((bb_z_min <= oct_z_min && oct_z_min <= bb_z_max) ||
     (bb_z_min < oct_z_max && oct_z_max < bb_z_max)))
  axis_overlap_count++;

/* If there is overlap between the octant and the bounding box, then call
   extract_pixels recursively on each of the subregions */
if (axis_overlap_count == 3)       /* All 3 axes overlap */
  {
    extract_pixels (octant->oct1, found_pixels, bb_x_min, bb_x_max,
                    bb_y_min, bb_y_max, bb_z_min, bb_z_max);

    extract_pixels (octant->oct2, found_pixels, bb_x_min, bb_x_max,
                    bb_y_min, bb_y_max, bb_z_min, bb_z_max);

    extract_pixels (octant->oct3, found_pixels, bb_x_min, bb_x_max,
                    bb_y_min, bb_y_max, bb_z_min, bb_z_max);

    extract_pixels (octant->oct4, found_pixels, bb_x_min, bb_x_max,
                    bb_y_min, bb_y_max, bb_z_min, bb_z_max);

    extract_pixels (octant->oct5, found_pixels, bb_x_min, bb_x_max,
                    bb_y_min, bb_y_max, bb_z_min, bb_z_max);

    extract_pixels (octant->oct6, found_pixels, bb_x_min, bb_x_max,
                    bb_y_min, bb_y_max, bb_z_min, bb_z_max);

    extract_pixels (octant->oct7, found_pixels, bb_x_min, bb_x_max,
                    bb_y_min, bb_y_max, bb_z_min, bb_z_max);

    extract_pixels (octant->oct8, found_pixels, bb_x_min, bb_x_max,
                    bb_y_min, bb_y_max, bb_z_min, bb_z_max);
  }
}
```

```
/*****************************************************************************
 *                        verify_extract_pixels
 *                                                                           250
 *
 *****************************************************************************/

void verify_extract_pixels (PIXEL_LIST *tree_pixels, PIXEL_LIST *found_pixels,
                            double bb_x_min, double bb_x_max, double bb_y_min,
                            double bb_y_max, double bb_z_min, double bb_z_max)
{
 register int i;
 PIXEL *pixel;
                                                                           260
 for (i = 0; i < tree_pixels->num_pixels; i++)
   {
    pixel = tree_pixels->pixel_array[i];

    if (bb_x_min <= pixel->x && pixel->x <= bb_x_max &&
        bb_y_min <= pixel->y && pixel->y <= bb_y_max &&
        bb_z_min <= pixel->z && pixel->z <= bb_z_max)
       add_to_pixel_list (found_pixels, pixel);
   }
}                                                                          270


/*****************************************************************************
 *                        recursively_add_pixels
 *
 *
 *****************************************************************************/
                                                                           280
void recursively_add_pixels (OCTANT_TREE_NODE *node, PIXEL_LIST *found_pixels)
{
 if (node == NULL)
   return;

 if (node->pixel != NULL)        /* leaf node */
   {
    add_to_pixel_list (found_pixels, node->pixel);
    return;
   }                                                                       290

 recursively_add_pixels (node->oct1, found_pixels);
 recursively_add_pixels (node->oct2, found_pixels);
 recursively_add_pixels (node->oct3, found_pixels);
 recursively_add_pixels (node->oct4, found_pixels);
 recursively_add_pixels (node->oct5, found_pixels);
 recursively_add_pixels (node->oct6, found_pixels);
 recursively_add_pixels (node->oct7, found_pixels);
 recursively_add_pixels (node->oct8, found_pixels);
}                                                                          300


/*****************************************************************************
 *                          next_octant
 *
 *
 *****************************************************************************/
                                                                           310
OCTANT_TREE_NODE* next_octant (OCTANT_TREE_NODE *node, int *sub_octant_num,
                               double x, double y, double z)
{
 double x_mid, y_mid, z_mid;

 x_mid = (node->x_min + node->x_max) / 2.0;
 y_mid = (node->y_min + node->y_max) / 2.0;
 z_mid = (node->z_min + node->z_max) / 2.0;

                                                                           320
 if (x >= x_mid)
   {                                    /* O2, O3, O6, O7 */
    if (y >= y_mid)
```

```
        {                                       /* O2, O6 */
          if (z >= z_mid)                           /* O2 */
            { *sub_octant_num = OCTANT2; return (node->oct2); }
          else                                    /* O6 */
            { *sub_octant_num = OCTANT6; return (node->oct6); }
        }
      else                                                                    330
        {                                       /* O3, O7 */
          if (z >= z_mid)                           /* O3 */
            { *sub_octant_num = OCTANT3; return (node->oct3); }
          else                                    /* O7 */
            { *sub_octant_num = OCTANT7; return (node->oct7); }
        }
    }

  else
    {                                       /* O1, O4, O5, O8 */              340
      if (y >= y_mid)
        {                                       /* O1, O5 */
          if (z >= z_mid)                           /* O1 */
            { *sub_octant_num = OCTANT1; return (node->oct1); }
          else                                    /* O5 */
            { *sub_octant_num = OCTANT5; return (node->oct5); }
        }
      else
        {                                       /* O4, O8 */
          if (z >= z_mid)                           /* O4 */                  350
            { *sub_octant_num = OCTANT4; return (node->oct4); }
          else                                    /* O8 */
            { *sub_octant_num = OCTANT8; return (node->oct8); }
        }
    }
}


                                                                              360
/*****************************************************************************
 *                      place_new_octant_tree_node
 *
 *
 *****************************************************************************/

void place_new_octant_tree_node (OCTANT_TREE_NODE *node, int sub_octant_num,
                                 OCTANT_TREE_NODE *new_node)
{
  switch (sub_octant_num)                                                     370
    {
    case OCTANT1:
      node->oct1 = new_node;
      break;

    case OCTANT2:
      node->oct2 = new_node;
      break;

    case OCTANT3:                                                             380
      node->oct3 = new_node;
      break;

    case OCTANT4:
      node->oct4 = new_node;
      break;

    case OCTANT5:
      node->oct5 = new_node;
      break;                                                                  390

    case OCTANT6:
      node->oct6 = new_node;
      break;

    case OCTANT7:
      node->oct7 = new_node;
      break;
```

```
    case OCTANT8:                                                          400
      node->oct8 = new_node;
      break;
    }
}



/*****************************************************************************
 *                        sub_octant_boundaries                              410
 *
 *
 *****************************************************************************/

void sub_octant_boundaries (OCTANT_TREE_NODE *node, int sub_octant_num,
                            double *x_min, double *x_max,
                            double *y_min, double *y_max,
                            double *z_min, double *z_max)
{
  double x_mid, y_mid, z_mid;                                              420

  x_mid = (node->x_min + node->x_max) / 2.0;
  y_mid = (node->y_min + node->y_max) / 2.0;
  z_mid = (node->z_min + node->z_max) / 2.0;


  switch (sub_octant_num)
    {
    case OCTANT1:
      *x_min = node->x_min; *x_max = x_mid;                                430
      *y_min = y_mid; *y_max = node->y_max;
      *z_min = z_mid; *z_max = node->z_max;
      break;

    case OCTANT2:
      *x_min = x_mid; *x_max = node->x_max;
      *y_min = y_mid; *y_max = node->y_max;
      *z_min = z_mid; *z_max = node->z_max;
      break;
                                                                           440
    case OCTANT3:
      *x_min = x_mid; *x_max = node->x_max;
      *y_min = node->y_min; *y_max = y_mid;
      *z_min = z_mid; *z_max = node->z_max;
      break;

    case OCTANT4:
      *x_min = node->x_min; *x_max = x_mid;
      *y_min = node->y_min; *y_max = y_mid;
      *z_min = z_mid; *z_max = node->z_max;                                450
      break;

    case OCTANT5:
      *x_min = node->x_min; *x_max = x_mid;
      *y_min = y_mid; *y_max = node->y_max;
      *z_min = node->z_min; *z_max = z_mid;
      break;

    case OCTANT6:
      *x_min = x_mid; *x_max = node->x_max;                                460
      *y_min = y_mid; *y_max = node->y_max;
      *z_min = node->z_min; *z_max = z_mid;
      break;

    case OCTANT7:
      *x_min = x_mid; *x_max = node->x_max;
      *y_min = node->y_min; *y_max = y_mid;
      *z_min = node->z_min; *z_max = z_mid;
      break;
                                                                           470
    case OCTANT8:
      *x_min = node->x_min; *x_max = x_mid;
      *y_min = node->y_min; *y_max = y_mid;
      *z_min = node->z_min; *z_max = z_mid;
      break;
```

```
    }
}

/*****************************************************************************
 *                        unparse_octant_tree
 *
 * unparse_octant_tree recursively traverses the pixel tree, and calls
 * unparse_pixel to print the values of the PIXEL struct fields.
 *
 *****************************************************************************/

void unparse_octant_tree (OCTANT_TREE_NODE *node)
{
  if (node == NULL)
    return;

  if (node->pixel != NULL)          /* leaf of the octant tree */
    {
      unparse_pixel (node->pixel);
      return;
    }

  /* Otherwise, must be an interior node: */

  printf("Interior node\n");
  printf("-------------\n");
  printf("x_min:  %lf\n", node->x_min);
  printf("x_max:  %lf\n", node->x_max);
  printf("y_min:  %lf\n", node->y_min);
  printf("y_max:  %lf\n", node->y_max);
  printf("z_min:  %lf\n", node->z_min);
  printf("z_max:  %lf\n", node->z_max);
  printf("\n\n");

  unparse_octant_tree (node->oct1);
  unparse_octant_tree (node->oct2);
  unparse_octant_tree (node->oct3);
  unparse_octant_tree (node->oct4);
  unparse_octant_tree (node->oct5);
  unparse_octant_tree (node->oct6);
  unparse_octant_tree (node->oct7);
  unparse_octant_tree (node->oct8);
}


/*****************************************************************************
 *                        free_octant_tree
 *
 * free_octant_tree recursively deallocates memory for the tree pointed to
 * by 'node'.  Note that this procedure does not free the individual
 * pixels in the tree.
 *
 *****************************************************************************/

void free_octant_tree (OCTANT_TREE_NODE *node)
{
  if (node == NULL)
    return;

  free_octant_tree (node->oct1);
  free_octant_tree (node->oct2);
  free_octant_tree (node->oct3);
  free_octant_tree (node->oct4);
  free_octant_tree (node->oct5);
  free_octant_tree (node->oct6);
  free_octant_tree (node->oct7);
  free_octant_tree (node->oct8);
  free (node);
}
```

```
/*****************************************************************************
 *                          number_of_nodes
 *
 * number_of_nodes recursively traverses the pixel tree pointed to by
 * 'node'.  As it reaches each node, 'num_nodes' is incremented by 1.
 * This function is useful for testing how well-balanced the pixel tree is.
 *
 *****************************************************************************/

int number_of_nodes (OCTANT_TREE_NODE *node)
{
  if (node != NULL)
    return (number_of_nodes_real (node, TRUE));
  else
    return 0;
}


int number_of_nodes_real (OCTANT_TREE_NODE *node, int new)
{
  static int num_nodes = 0;

  if (node == NULL)
    return num_nodes;

  if (new == TRUE)
    num_nodes = 0;

  num_nodes++;

  number_of_nodes_real (node->oct1, FALSE);
  number_of_nodes_real (node->oct2, FALSE);
  number_of_nodes_real (node->oct3, FALSE);
  number_of_nodes_real (node->oct4, FALSE);
  number_of_nodes_real (node->oct5, FALSE);
  number_of_nodes_real (node->oct6, FALSE);
  number_of_nodes_real (node->oct7, FALSE);
  number_of_nodes_real (node->oct8, FALSE);

  return num_nodes;
}


/*****************************************************************************
 *                          num_pixels_in_tree
 *
 *
 *****************************************************************************/

int num_pixels_in_tree (OCTANT_TREE_NODE *node)
{
  if (node != NULL)
    return (num_pixels_in_tree_real (node, TRUE));
  else
    return 0;
}


int num_pixels_in_tree_real (OCTANT_TREE_NODE *node, int new)
{
  static int num_pixels = 0;

  if (node == NULL)
    return num_pixels;

  if (new == TRUE)
    num_pixels = 0;

  if (node->pixel != NULL)   /* reached a leaf --> increment num_pixels */
    {
```

```
    num_pixels++;
    return num_pixels;
   }

  num_pixels_in_tree_real (node->oct1, FALSE);
  num_pixels_in_tree_real (node->oct2, FALSE);
  num_pixels_in_tree_real (node->oct3, FALSE);
  num_pixels_in_tree_real (node->oct4, FALSE);
  num_pixels_in_tree_real (node->oct5, FALSE);
  num_pixels_in_tree_real (node->oct6, FALSE);
  num_pixels_in_tree_real (node->oct7, FALSE);
  num_pixels_in_tree_real (node->oct8, FALSE);

  return num_pixels;
}


/*****************************************************************************
 *                      find_min_max_coordinates
 *
 *
 *****************************************************************************/

void find_min_max_coordinates (PIXEL_LIST *pixel_list,
                               double *x_min, double *x_max, double *y_min,
                               double *y_max, double *z_min, double *z_max)
{
  register int i;
  PIXEL *pixel;

  if (pixel_list == NULL)
    return;

  pixel = pixel_list->pixel_array[0];

  *x_min = pixel->x;  *x_max = pixel->x;
  *y_min = pixel->y;  *y_max = pixel->y;
  *z_min = pixel->z;  *z_max = pixel->z;

  for (i = 0; i < pixel_list->num_pixels; i++)
    {
      pixel = pixel_list->pixel_array[i];

      *x_min = (pixel->x < *x_min) ? pixel->x : *x_min;
      *x_max = (pixel->x > *x_max) ? pixel->x : *x_max;
      *y_min = (pixel->y < *y_min) ? pixel->y : *y_min;
      *y_max = (pixel->y > *y_max) ? pixel->y : *y_max;
      *z_min = (pixel->z < *z_min) ? pixel->z : *z_min;
      *z_max = (pixel->z > *z_max) ? pixel->z : *z_max;
    }

}
```

B.12 Cluster Tree Functions

cluster_tree.c

```
CLUSTER_TREE_NODE* create_cluster_tree_node (double x_min, double x_max,
                                             double y_min, double y_max,
                                             double z_min, double z_max);

CLUSTER_TREE_NODE* build_cluster_tree (CLUSTER_LIST *cluster_list);

void add_cluster_to_tree (CLUSTER_TREE_NODE* node, CLUSTER* cluster);

void extract_clusters (CLUSTER_TREE_NODE *node,                                  10
                       CLUSTER_LIST *found_clusters,
                       double bb_x_min, double bb_x_max, double bb_y_min,
                       double bb_y_max, double bb_z_min, double bb_z_max);

void verify_extract_clusters (CLUSTER_LIST *tree_clusters,
                              CLUSTER_LIST *found_clusters,
                              double bb_x_min, double bb_x_max,
                              double bb_y_min, double bb_y_max,
                              double bb_z_min, double bb_z_max);
                                                                                 20
void recursively_add_clusters (CLUSTER_TREE_NODE *node,
                               CLUSTER_LIST *found_clusters);

CLUSTER_TREE_NODE* next_cluster_tree_octant (CLUSTER_TREE_NODE *node,
                                             int *sub_octant_num,
                                             double x, double y, double z);

void place_new_cluster_tree_node (CLUSTER_TREE_NODE *node, int sub_octant_num,
                                  CLUSTER_TREE_NODE *new_node);
                                                                                 30
void sub_cluster_tree_boundaries (CLUSTER_TREE_NODE *node, int sub_octant_num,
                                  double *x_min, double *x_max,
                                  double *y_min, double *y_max,
                                  double *z_min, double *z_max);

void find_cluster_list_min_max_coordinates (CLUSTER_LIST *cluster_list,
                                            double *x_min, double *x_max,
                                            double *y_min, double *y_max,
                                            double *z_min, double *z_max);
                                                                                 40
void free_cluster_tree (CLUSTER_TREE_NODE *node);
```

cluster_tree.h

```
#include <stdio.h>
#include "global.h"
#include "cluster.h"
#include "cluster_tree.h"


/*****************************************************************************
 *                    create_cluster_tree_node
 *                                                                              10
 *
 *****************************************************************************/

CLUSTER_TREE_NODE* create_cluster_tree_node (double x_min, double x_max,
                                             double y_min, double y_max,
                                             double z_min, double z_max)
{
  CLUSTER_TREE_NODE *cluster_tree_node =
    (CLUSTER_TREE_NODE*) malloc (sizeof (CLUSTER_TREE_NODE));
                                                                                 20
  cluster_tree_node->oct1 = NULL;
  cluster_tree_node->oct2 = NULL;
```

```
  cluster_tree_node->oct3 = NULL;
  cluster_tree_node->oct4 = NULL;
  cluster_tree_node->oct5 = NULL;
  cluster_tree_node->oct6 = NULL;
  cluster_tree_node->oct7 = NULL;
  cluster_tree_node->oct8 = NULL;

  cluster_tree_node->x_min = x_min;                                          30
  cluster_tree_node->x_max = x_max;
  cluster_tree_node->y_min = y_min;
  cluster_tree_node->y_max = y_max;
  cluster_tree_node->z_min = z_min;
  cluster_tree_node->z_max = z_max;

  cluster_tree_node->cluster = NULL;

  return cluster_tree_node;
}                                                                            40


/*****************************************************************************
 *                        build_cluster_tree
 *
 *
 *****************************************************************************/
                                                                             50
CLUSTER_TREE_NODE* build_cluster_tree (CLUSTER_LIST *cluster_list)
{
  register int i;
  double x_min, x_max, y_min, y_max, z_min, z_max;
  CLUSTER_TREE_NODE *cluster_tree_root;


  if (cluster_list == NULL)
    return NULL;
                                                                             60
  find_cluster_list_min_max_coordinates (cluster_list,
                                         &x_min, &x_max,
                                         &y_min, &y_max,
                                         &z_min, &z_max);


  cluster_tree_root =
    create_cluster_tree_node (x_min, x_max, y_min, y_max, z_min, z_max);
                                                                             70
  for (i = 0; i < cluster_list->num_clusters; i++)
    add_cluster_to_tree (cluster_tree_root, cluster_list->cluster_array[i]);

  return cluster_tree_root;
}



/*****************************************************************************  80
 *                        add_cluster_to_tree
 *
 *
 *****************************************************************************/

void add_cluster_to_tree (CLUSTER_TREE_NODE* node, CLUSTER* cluster)
{
  CLUSTER_TREE_NODE *sub_octant_ptr;
  CENTROID *centroid;
  CLUSTER *temp_cluster;                                                     90
  int sub_octant_num;
  double x_min, x_max, y_min, y_max, z_min, z_max;


  /* Must insert an interior node and decide which cluster the cluster
     belongs to. */

  if (node->cluster != NULL)          /* node is a leaf */
    {
      temp_cluster = node->cluster;                                         100
```

```
    node->cluster = NULL;          /* node is no longer a leaf */

    /* Call recursively on temp_cluster and cluster so they can be placed: */
    add_cluster_to_tree (node, temp_cluster);
    add_cluster_to_tree (node, cluster);
    return;
   }

 /* node is an interior node: */                                          110
 centroid = cluster->centroid;

 sub_octant_ptr =
   next_cluster_tree_octant (node, &sub_octant_num,
                             centroid->x, centroid->y, centroid->z);

 if (sub_octant_ptr == NULL)
   {
    sub_cluster_tree_boundaries (node, sub_octant_num, &x_min, &x_max,   120
                                 &y_min, &y_max, &z_min, &z_max);

    /* Create new leaf: */
    sub_octant_ptr =
        create_cluster_tree_node (x_min, x_max, y_min, y_max, z_min, z_max);

    sub_octant_ptr->cluster = cluster;

    /* Attach the new leaf to the tree: */
    place_new_cluster_tree_node (node, sub_octant_num, sub_octant_ptr);  130

    return;
   }

 /* Otherwise, recursively traverse tree to find insertion point: */
 add_cluster_to_tree (sub_octant_ptr, cluster);
}

                                                                         140
/*************************************************************************
 *                        extract_clusters
 *
 * Extract the clusters from the cluster tree which lie in the desired
 * conical search volume.
 *
 *************************************************************************/

void extract_clusters (CLUSTER_TREE_NODE *node,                          150
                       CLUSTER_LIST *found_clusters,
                       double bb_x_min, double bb_x_max, double bb_y_min,
                       double bb_y_max, double bb_z_min, double bb_z_max)
{
 int axis_overlap_count;
 double oct_x_min, oct_x_max, oct_y_min, oct_y_max, oct_z_min, oct_z_max;
 CLUSTER *cluster;
 CENTROID *centroid;

 if (node == NULL)                                                       160
   return;

 oct_x_min = node->x_min;  oct_x_max = node->x_max;
 oct_y_min = node->y_min;  oct_y_max = node->y_max;
 oct_z_min = node->z_min;  oct_z_max = node->z_max;

 cluster = node->cluster;

 /* If reach a leaf, test the cluster to see if it should be included */
 if (cluster != NULL)                                                    170
   {
    centroid = cluster->centroid;

    if (bb_x_min <= centroid->x && centroid->x <= bb_x_max &&
        bb_y_min <= centroid->y && centroid->y <= bb_y_max &&
        bb_z_min <= centroid->z && centroid->z <= bb_z_max)
       add_to_cluster_list (found_clusters, cluster);
```

```
    /* check cluster against search volume */

    return;                                                                    180
  }

/* Otherwise, must be an interior node... */

/* If cluster is completely enclosed by the bounding box, then recursively
   add all clusters in the subregions */
if (bb_x_min <= oct_x_min && oct_x_min <= bb_x_max &&
    bb_x_min < oct_x_max && oct_x_max < bb_x_max &&                            190
    bb_y_min <= oct_y_min && oct_y_min <= bb_y_max &&
    bb_y_min < oct_y_max && oct_y_max < bb_y_max &&
    bb_z_min <= oct_z_min && oct_z_min <= bb_z_max &&
    bb_z_min < oct_z_max && oct_z_max < bb_z_max)
  {
    recursively_add_clusters (node, found_clusters);
    return;
  }

                                                                               200
axis_overlap_count = 0;

/* Test for overlap along x axis: */
if (((oct_x_min <= bb_x_min && bb_x_min < oct_x_max) ||
     (oct_x_min <= bb_x_max && bb_x_max < oct_x_max)) ||
    ((bb_x_min <= oct_x_min && oct_x_min <= bb_x_max) ||
     (bb_x_min < oct_x_max && oct_x_max < bb_x_max)))
  axis_overlap_count++;

/* Test for overlap along y axis: */                                           210
if (((oct_y_min <= bb_y_min && bb_y_min < oct_y_max) ||
     (oct_y_min <= bb_y_max && bb_y_max < oct_y_max)) ||
    ((bb_y_min <= oct_y_min && oct_y_min <= bb_y_max) ||
     (bb_y_min < oct_y_max && oct_y_max < bb_y_max)))
  axis_overlap_count++;

/* Test for overlap along z axis: */
if (((oct_z_min <= bb_z_min && bb_z_min < oct_z_max) ||
     (oct_z_min <= bb_z_max && bb_z_max < oct_z_max)) ||
    ((bb_z_min <= oct_z_min && oct_z_min <= bb_z_max) ||                       220
     (bb_z_min < oct_z_max && oct_z_max < bb_z_max)))
  axis_overlap_count++;

/* If there is overlap between the cluster and the bounding box, then call
   extract_clusters recursively on each of the subregions */
if (axis_overlap_count == 3)       /* All 3 axes overlap */
  {
    extract_clusters (node->oct1, found_clusters, bb_x_min, bb_x_max,
                      bb_y_min, bb_y_max, bb_z_min, bb_z_max);                 230

    extract_clusters (node->oct2, found_clusters, bb_x_min, bb_x_max,
                      bb_y_min, bb_y_max, bb_z_min, bb_z_max);

    extract_clusters (node->oct3, found_clusters, bb_x_min, bb_x_max,
                      bb_y_min, bb_y_max, bb_z_min, bb_z_max);

    extract_clusters (node->oct4, found_clusters, bb_x_min, bb_x_max,
                      bb_y_min, bb_y_max, bb_z_min, bb_z_max);
                                                                               240
    extract_clusters (node->oct5, found_clusters, bb_x_min, bb_x_max,
                      bb_y_min, bb_y_max, bb_z_min, bb_z_max);

    extract_clusters (node->oct6, found_clusters, bb_x_min, bb_x_max,
                      bb_y_min, bb_y_max, bb_z_min, bb_z_max);

    extract_clusters (node->oct7, found_clusters, bb_x_min, bb_x_max,
                      bb_y_min, bb_y_max, bb_z_min, bb_z_max);

    extract_clusters (node->oct8, found_clusters, bb_x_min, bb_x_max,          250
                      bb_y_min, bb_y_max, bb_z_min, bb_z_max);
  }
}
```

```
/*****************************************************************************
 *                      verify_extract_clusters
 *                                                                           260
 *
 *****************************************************************************/

void verify_extract_clusters (CLUSTER_LIST *tree_clusters,
                              CLUSTER_LIST *found_clusters,
                              double bb_x_min, double bb_x_max,
                              double bb_y_min, double bb_y_max,
                              double bb_z_min, double bb_z_max)
{
 register int i;                                                           270
 CLUSTER *cluster;
 CENTROID *centroid;

 for (i = 0; i < tree_clusters->num_clusters; i++)
   {
    cluster = tree_clusters->cluster_array[i];
    centroid = cluster->centroid;

    if (bb_x_min <= centroid->x && centroid->x <= bb_x_max &&
        bb_y_min <= centroid->y && centroid->y <= bb_y_max &&                280
        bb_z_min <= centroid->z && centroid->z <= bb_z_max)
       add_to_cluster_list (found_clusters, cluster);
   }
}



/*****************************************************************************
 *                      recursively_add_clusters                             290
 *
 *
 *****************************************************************************/

void recursively_add_clusters (CLUSTER_TREE_NODE *node,
                               CLUSTER_LIST *found_clusters)
{
 if (node == NULL)
   return;
                                                                           300
 if (node->cluster != NULL)        /* leaf node */
   {
    add_to_cluster_list (found_clusters, node->cluster);
    return;
   }

 recursively_add_clusters (node->oct1, found_clusters);
 recursively_add_clusters (node->oct2, found_clusters);
 recursively_add_clusters (node->oct3, found_clusters);
 recursively_add_clusters (node->oct4, found_clusters);                    310
 recursively_add_clusters (node->oct5, found_clusters);
 recursively_add_clusters (node->oct6, found_clusters);
 recursively_add_clusters (node->oct7, found_clusters);
 recursively_add_clusters (node->oct8, found_clusters);
}



/*****************************************************************************                                                                           320
 *                      next_cluster_tree_octant
 *
 *
 *****************************************************************************/

CLUSTER_TREE_NODE* next_cluster_tree_octant (CLUSTER_TREE_NODE *node,
                                             int *sub_octant_num,
                                             double x, double y, double z)
{
```

```
 double x_mid, y_mid, z_mid;                                                    330

 x_mid = (node->x_min + node->x_max) / 2.0;
 y_mid = (node->y_min + node->y_max) / 2.0;
 z_mid = (node->z_min + node->z_max) / 2.0;


 if (x >= x_mid)
   {                                        /* O2, O3, O6, O7 */
    if (y >= y_mid)
         {                                    /* O2, O6 */                     340
          if (z >= z_mid)                       /* O2 */
            { *sub_octant_num = OCTANT2; return (node->oct2); }
          else                                  /* O6 */
            { *sub_octant_num = OCTANT6; return (node->oct6); }
         }
    else
         {                                    /* O3, O7 */
          if (z >= z_mid)                       /* O3 */
            { *sub_octant_num = OCTANT3; return (node->oct3); }
          else                                  /* O7 */                         350
            { *sub_octant_num = OCTANT7; return (node->oct7); }
         }
   }

 else
   {                                        /* O1, O4, O5, O8 */
    if (y >= y_mid)
         {                                    /* O1, O5 */
          if (z >= z_mid)                       /* O1 */
            { *sub_octant_num = OCTANT1; return (node->oct1); }                 360
          else                                  /* O5 */
            { *sub_octant_num = OCTANT5; return (node->oct5); }
         }
    else
         {                                    /* O4, O8 */
          if (z >= z_mid)                       /* O4 */
            { *sub_octant_num = OCTANT4; return (node->oct4); }
          else                                  /* O8 */
            { *sub_octant_num = OCTANT8; return (node->oct8); }
         }                                                                      370
   }
}



/*****************************************************************************
 *                    place_new_cluster_tree_node
 *
 *                                                                            380
 *****************************************************************************/

void place_new_cluster_tree_node (CLUSTER_TREE_NODE *node, int sub_octant_num,
                                  CLUSTER_TREE_NODE *new_node)
{
 switch (sub_octant_num)
   {
   case OCTANT1:
     node->oct1 = new_node;
     break;                                                                     390

   case OCTANT2:
     node->oct2 = new_node;
     break;

   case OCTANT3:
     node->oct3 = new_node;
     break;

   case OCTANT4:                                                                400
     node->oct4 = new_node;
     break;

   case OCTANT5:
```

```
    node->oct5 = new_node;
    break;

  case OCTANT6:
    node->oct6 = new_node;
    break;

  case OCTANT7:
    node->oct7 = new_node;
    break;

  case OCTANT8:
    node->oct8 = new_node;
    break;
  }
}


/*****************************************************************************
 *                    sub_cluster_tree_boundaries
 *
 *
 *****************************************************************************/

void sub_cluster_tree_boundaries (CLUSTER_TREE_NODE *node, int sub_octant_num,
                                  double *x_min, double *x_max,
                                  double *y_min, double *y_max,
                                  double *z_min, double *z_max)
{
 double x_mid, y_mid, z_mid;

 x_mid = (node->x_min + node->x_max) / 2.0;
 y_mid = (node->y_min + node->y_max) / 2.0;
 z_mid = (node->z_min + node->z_max) / 2.0;

 switch (sub_octant_num)
  {
  case OCTANT1:
    *x_min = node->x_min; *x_max = x_mid;
    *y_min = y_mid; *y_max = node->y_max;
    *z_min = z_mid; *z_max = node->z_max;
    break;

  case OCTANT2:
    *x_min = x_mid; *x_max = node->x_max;
    *y_min = y_mid; *y_max = node->y_max;
    *z_min = z_mid; *z_max = node->z_max;
    break;

  case OCTANT3:
    *x_min = x_mid; *x_max = node->x_max;
    *y_min = node->y_min; *y_max = y_mid;
    *z_min = z_mid; *z_max = node->z_max;
    break;

  case OCTANT4:
    *x_min = node->x_min; *x_max = x_mid;
    *y_min = node->y_min; *y_max = y_mid;
    *z_min = z_mid; *z_max = node->z_max;
    break;

  case OCTANT5:
    *x_min = node->x_min; *x_max = x_mid;
    *y_min = y_mid; *y_max = node->y_max;
    *z_min = node->z_min; *z_max = z_mid;
    break;

  case OCTANT6:
    *x_min = x_mid; *x_max = node->x_max;
    *y_min = y_mid; *y_max = node->y_max;
    *z_min = node->z_min; *z_max = z_mid;
    break;

  case OCTANT7:
```

```
    *x_min = x_mid; *x_max = node->x_max;
    *y_min = node->y_min; *y_max = y_mid;
    *z_min = node->z_min; *z_max = z_mid;
    break;

  case OCTANT8:
    *x_min = node->x_min; *x_max = x_mid;
    *y_min = node->y_min; *y_max = y_mid;
    *z_min = node->z_min; *z_max = z_mid;
    break;
  }
}


/*****************************************************************************
 *                find_cluster_list_min_max_coordinates
 *
 *
 *****************************************************************************/

void find_cluster_list_min_max_coordinates (CLUSTER_LIST *cluster_list,
                                            double *x_min, double *x_max,
                                            double *y_min, double *y_max,
                                            double *z_min, double *z_max)
{
 register int i;
 CENTROID *centroid;

 if (cluster_list == NULL)
   return;

 centroid = cluster_list->cluster_array[0]->centroid;

 *x_min = centroid->x;  *x_max = centroid->x;
 *y_min = centroid->y;  *y_max = centroid->y;
 *z_min = centroid->z;  *z_max = centroid->z;

 for (i = 0; i < cluster_list->num_clusters; i++)
   {
    centroid = cluster_list->cluster_array[i]->centroid;

    *x_min = (centroid->x < *x_min) ? centroid->x : *x_min;
    *x_max = (centroid->x > *x_max) ? centroid->x : *x_max;
    *y_min = (centroid->y < *y_min) ? centroid->y : *y_min;
    *y_max = (centroid->y > *y_max) ? centroid->y : *y_max;
    *z_min = (centroid->z < *z_min) ? centroid->z : *z_min;
    *z_max = (centroid->z > *z_max) ? centroid->z : *z_max;
   }

}



/*****************************************************************************
 *                        free_cluster_tree
 *
 * free_cluster_tree recursively deallocates memory for the tree pointed
 * to by 'node'.  Note that this procedure does not free the individual
 * clusters in the tree.
 *
 *****************************************************************************/

void free_cluster_tree (CLUSTER_TREE_NODE *node)
{
 if (node == NULL)
   return;

 free_cluster_tree (node->oct1);
 free_cluster_tree (node->oct2);
 free_cluster_tree (node->oct3);
 free_cluster_tree (node->oct4);
```

```
  free_cluster_tree (node->oct5);
  free_cluster_tree (node->oct6);                                    560
  free_cluster_tree (node->oct7);
  free_cluster_tree (node->oct8);
  free (node);
}
```

Appendix C
Diagnostic Routines

C.1 Main Diagnostic Routine

diagnostic.h

```
#ifndef _DIAGNOSTIC_
#define _DIAGNOSTIC_

#define TRUE_TRACK_STATS 1
#define FOUND_TRACK_STATS 2

#endif
```

diagnostic.cc

```
#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include <fstream.h>

#include "diagnostic.h"
#include "entry.h"
#include "track.h"
#include "table.h"

// simple diagnostics for track finder dump file

main()
{
 char filename[200];

 cout << "Enter pixel dump filename:  ";
 cin >> filename;
 cout << endl;

 ifstream in(filename);

 if(!in)
  {
   cout << "Couldn't open " << filename << ".    Exiting..." << endl;
   exit(0);
  }

 /*
 cout << "Enter diagnostic results output filename: ";
 cin >> filename;
 cout << endl;

 ofstream out(filename);

 if(!out)
  {
   cout << "Couldn't open " << filename << ".  Exiting..." << endl;
   exit(0);
  }
 */

 int true_found;
 cout << "Specify (1) True tid or (2) Found tid:  ";
 cin >> true_found;
 cout << endl;
```

```
if (true_found != TRUE_TRACK_STATS && true_found != FOUND_TRACK_STATS)
  {                                                                          50
    cout << "Did not enter 1 or 2.     Exiting." << endl;
    exit(0);
  }

// go through dump file on line at a time and build up statistics:
// get a line, then get the true tid and the found tid.

Table table;
char buffer[500];
                                                                             60
// dump file format:
// found_tid  true_tid  rho  theta  phi  {module_id}

cout << "Processing dump file..." << endl;

while (!in.getline(buffer, 500).eof())
  {
    char token_chars[] = " \t\n";

    char* ptr = strtok(buffer, token_chars); // extract true track ID          70
    int found_tid = (int) atof(ptr);

    ptr = strtok(NULL, token_chars);         // extract found track ID
    int true_tid = (int) atof(ptr);

    table.process_pixel(true_found, found_tid, true_tid);

    //ccout << "****************************" << endl;
    //ttable.unparse(true_found);
    //ccout << "****************************" << endl;                        80
  }

cout << "Finished.     Displaying results:" << endl;

cout << endl << endl;
table.unparse(true_found);
cout << endl << endl;
}
                                                                             90
```

C.2 Track Functions

track.h

```
#ifndef _TRACK_
#define _TRACK_

#include "entry.h"

class Track
{
  int _tid;
  int _num_entries;
  Entry* _entry_array;

public:
  Track();
  Track(int tid);
  ~Track();
  void process_pixel(int tid);
  void add_entry(int tid);
  void sort_entries();
  int total_num_pixels();
  int get_tid();
  void unparse(int true_found);
};

#endif _TRACK_
```

track.cc

```
#include <iostream.h>
#include "diagnostic.h"
#include "entry.h"
#include "track.h"

Track::Track()
{
  _tid = 0;
  _num_entries = 0;
  _entry_array = NULL;
}

Track::Track(int tid)
{
  _tid = tid;
  _num_entries = 0;
  _entry_array = NULL;
}

Track::~Track()
{
  //if(_entry_array != NULL)
  //   delete [] _entry_array;
}

void Track::process_pixel(int tid)
{
  //  Go through list of entries and look for one that
  //  has the same tid:

  for(int i = 0; i < _num_entries; i++)
    if(_entry_array[i].get_tid() == tid)
      {
          _entry_array[i].add_pixel();
```

```
        return;
    }

  // If the tid wasn't found, then a new entry needs to be added:
  add_entry(tid);
}

void Track::add_entry(int tid)
{
  Entry* temp_entry_array = new Entry[_num_entries + 1];

  // copy existing entries
  for(int i = 0; i < _num_entries; i++)
    temp_entry_array[i] = _entry_array[i];

  if(_entry_array != NULL)
    delete [] _entry_array;

  _entry_array = temp_entry_array;

  _entry_array[_num_entries] = Entry(tid);
  _num_entries++;
}

void Track::sort_entries()
{
  // use merge sort
  // cout << "sort_entries()..." << endl;

  // temporary - insertion sort (runs in n^2 time):
  for(int i = _num_entries - 1; i >= 0; i--)
    {
      for(int j = 0; j < i; j++)
          if(_entry_array[i].get_num_pixels() > _entry_array[j].get_num_pixels())
            {
              Entry temp_entry = _entry_array[i];
              _entry_array[i] = _entry_array[j];
              _entry_array[j] = temp_entry;
            }
    }
}

int Track::get_tid()
{
  return _tid;
}

int Track::total_num_pixels()
{
  int tot_pixels = 0;

  for(int i = 0; i < _num_entries; i++)
    tot_pixels += _entry_array[i].get_num_pixels();

  return tot_pixels;
}

void Track::unparse(int true_found)
{
  sort_entries();

  cout.setf(ios::left, ios::adjustfield);
  cout.setf(ios::fixed, ios::floatfield);
  cout.setf(ios::showpoint);
  cout.precision(1);

  if(true_found == TRUE_TRACK_STATS)
    {
      cout << "True track ID: " << _tid << endl;
      cout << "\t";
```

```
cout.width(16);
cout << "Found track ID";

cout.width(19);
cout << "Portion of pixels";

cout << "# of pixels" << endl;
cout << "\t";

cout.width(16);
cout << "--------------";

cout.width(19);
cout << "------------------";

cout << "-----------" << endl;

int tot_pixels = total_num_pixels();
int num_unfound_pixels = 0;
int num_other_entries = 0;
int num_other_pixels = 0;

for(int i = 0; i < _num_entries; i++)
    {
     int num_pixels = _entry_array[i].get_num_pixels();

     if(_entry_array[i].get_tid() == 0)  //  found_tid == 0 --> unfound
       num_unfound_pixels += num_pixels;

     else if(((float) num_pixels / (float) tot_pixels) < 0.05)
       //  change to user-setable value
       {
         num_other_entries++;
         num_other_pixels += num_pixels;
       }

     else
       {
         cout << "\t";
         cout.width(16);
         cout << _entry_array[i].get_tid();

         float portion = ((float) num_pixels / (float) tot_pixels) * 100.0;

         if(portion < 10.0)  //  since it is rounded, use 9.95 (?)
              cout << " ";
         if(portion < 100.0)
              cout << " ";
         cout << portion;

         cout.width(14);
         cout << "%";

         cout << num_pixels << endl;
       }
    }

if(num_other_entries > 0)
    {
     cout << "\tother:   ";
     cout.width(9);
     cout << num_other_entries;

     float portion = ((float) num_other_pixels / (float) tot_pixels) * 100.0;

     if(portion < 10.0)  //  since it is rounded, use 9.95 (?)
       cout << " ";
     if(portion < 100.0)
       cout << " ";
     cout << portion;

     cout.width(14);
     cout << "%";

     cout << num_other_pixels << endl;
    }

if(num_unfound_pixels > 0)
```

```
        {
          cout << "\t";
          cout.width(16);
          cout << "Unfound";

          float portion = ((float) num_unfound_pixels / (float) tot_pixels) * 100.0;

          if(portion < 10.0)  //  since it is rounded, use 9.95 (?)                    200
            cout << " ";
          if(portion < 100.0)
            cout << " ";
          cout << portion;

          cout.width(14);
          cout << "%";

          cout << num_unfound_pixels << endl;
        }                                                                             210

    cout << "\t";

    cout.width(16);
    cout << "--------------";

    cout.width(19);
    cout << "-----------------";

    cout << "-----------" << endl;                                                   220

    cout << "Totals:\t";

    cout.width(16);
    cout << _num_entries;

    cout.width(19);
    cout << "100.0%";

    cout << tot_pixels << endl << endl << endl;                                     230
    return;
  }

else if(true_found == FOUND_TRACK_STATS)
  {
    cout << "Found track ID: " << _tid << endl;
    cout << "\t";

    cout.width(15);                                                                 240
    cout << "True track ID";

    cout.width(19);
    cout << "Portion of pixels";

    cout << "# of pixels" << endl;
    cout << "\t";

    cout.width(15);
    cout << "-------------";                                                        250

    cout.width(19);
    cout << "-----------------";

    cout << "-----------" << endl;

    int tot_pixels = total_num_pixels();
    int num_noise_pixels = 0;
    int num_other_entries = 0;
    int num_other_pixels = 0;                                                       260

    for(int i = 0; i < _num_entries; i++)
        {
          int num_pixels = _entry_array[i].get_num_pixels();

          if(_entry_array[i].get_tid() == 0)  //  found_tid == 0 --> noise
            num_noise_pixels += num_pixels;

          else if(((float) num_pixels / (float) tot_pixels) < 0.05)
            //  change to user-setable value                                       270
            {
```

```
            num_other_entries++;
            num_other_pixels += num_pixels;
          }

        else
          {
            cout << "\t";
            cout.width(15);
            cout << _entry_array[i].get_tid();

            float portion = ((float) num_pixels / (float) tot_pixels) * 100.0;

            if(portion < 10.0)  //  since it is rounded, use 9.95 (?)
                 cout << "  ";
            if(portion < 100.0)
                 cout << "  ";
            cout << portion;

            cout.width(14);
            cout << "%";

            cout << num_pixels << endl;
          }
      }

if(num_other_entries > 0)
     {
      cout << "\tother:   ";
      cout.width(8);
      cout << num_other_entries;

      float portion = ((float) num_other_pixels / (float) tot_pixels) * 100.0;

      if(portion < 10.0)  //  since it is rounded, use 9.95 (?)
        cout << "  ";
      if(portion < 100.0)
        cout << "  ";
      cout << portion;

      cout.width(14);
      cout << "%";

      cout << num_other_pixels << endl;
     }

if(num_noise_pixels > 0)
     {
      cout << "\t";
      cout.width(15);
      cout << "Noise";

      float portion = ((float) num_noise_pixels / (float) tot_pixels) * 100.0;

      if(portion < 10.0)  //  since it is rounded, use 9.95 (?)
        cout << "  ";
      if(portion < 100.0)
        cout << "  ";
      cout << portion;

      cout.width(14);
      cout << "%";

      cout << num_noise_pixels << endl;
     }

cout << "\t";

cout.width(15);
cout << "-------------";

cout.width(19);
cout << "------------------";

cout << "-----------" << endl;

cout << "Totals:\t";

cout.width(15);
```

```
    cout << _num_entries;                                                    350

    cout.width(19);
    cout << "100.0%";

    cout << tot_pixels << endl << endl << endl;
    return;
   }

  else
   {                                                                         360
    cout << "Error in Track::unparse!" << endl;
    return;
   }
}
```

C.3 Table Functions

table.h

```
#ifndef _TABLE_
#define _TABLE_

#include "track.h"

class Table
{
  int _num_tracks;
  Track* _track_array;

public:
  Table();
  ~Table();
  void process_pixel(int true_found, int found_tid, int true_tid);
  void unparse(int true_found);
};

#endif
```

table.cc

```
#include <iostream.h>
#include "diagnostic.h"
#include "table.h"

Table::Table()
{
  _num_tracks = 0;
  _track_array = NULL;
}

Table::~Table()
{
  if (_track_array != NULL)
    delete [] _track_array;
}

void Table::process_pixel(int true_found, int found_tid, int true_tid)
{
  // first check to see if track is already in the table:

  if(true_found == TRUE_TRACK_STATS)
    {
      for(int i = 0; i < _num_tracks; i++)
          if (true_tid == _track_array[i].get_tid())
            {
              _track_array[i].process_pixel(found_tid);
              return;
            }
    }
  else // true_found == FOUND_TRACK_STATS
    {
      for(int i = 0; i < _num_tracks; i++)
          if (found_tid == _track_array[i].get_tid())
            {
              _track_array[i].process_pixel(true_tid);
              return;
            }
    }

  // otherwise need to add another track to the table
```

```
  Track* temp_track_array = new Track[_num_tracks + 1];

  // copy the existing track array
  for(int i = 0; i < _num_tracks; i++)
    temp_track_array[i] = _track_array[i];

  if(_track_array != NULL)                                                    50
    delete [] _track_array;

  _track_array = temp_track_array;

  if(true_found == TRUE_TRACK_STATS)
    {
      // cout << "adding track (tid = " << true_tid << ")" << endl;
      _track_array[_num_tracks] = Track(true_tid);
      _track_array[_num_tracks].add_entry(found_tid); // process_pixel (?)
    }                                                                         60
  else // true_found == FOUND_TRACK_STATS
    {
      // cout << "adding track (tid = " << found_tid << ")" << endl;
      _track_array[_num_tracks] = Track(found_tid);
      _track_array[_num_tracks].add_entry(true_tid); // process_pixel (?)
    }

  _num_tracks++;
}                                                                             70

void Table::unparse(int true_found)
{
  for(int i = 0; i < _num_tracks; i++)
    _track_array[i].unparse(true_found);
}


                                                                              80
```

C.4 Table Entry Functions

entry.h

```
#ifndef _ENTRY_
#define _ENTRY_

class Entry
{
  int _tid;
  int _num_pixels;

public:
  Entry();
  Entry(int tid);
  ~Entry();
  void add_pixel();
  int get_tid();
  int get_num_pixels();
};

#endif
```

entry.cc

```
#include "diagnostic.h"
#include "entry.h"

Entry::Entry()
{
  _tid = 0;
  _num_pixels = 0;
}

Entry::~Entry()
{
}

Entry::Entry(int tid)
{
  _tid = tid;
  _num_pixels = 1;
}

void Entry::add_pixel()
{
  _num_pixels++;
}

int Entry::get_tid()
{
  return _tid;
}

int Entry::get_num_pixels()
{
  return _num_pixels;
}
```

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1993.

[2] W. Eric L. Grimson. *Object Recognition by Computer: The Role of Geometric Constraints*. The MIT Press, 1991.

[3] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery. *Numerical Recipes in C, Second Edition*. Cambridge University Press, 1992.

[4] Saulys, Al. "STAR Home Page." *Solenoidal Tracker At RHIC (STAR) Experiment.* <http://www.rhic.bnl.gov/STAR/star.html> (28 May 1997).

[5] George F. Simmons. *Calculus with Analytic Geometry*. McGraw-Hill Book Company, 1985.

[6] Supercomputing Technologies Group. *Cilk-5.0 (Beta-1) Reference Manual*. MIT Laboratory for Computer Science, 1997.