Software Correction to Account for Dead Blocks in a Lead Glass Calorimeter

A senior thesis submitted in partial fulfillment of the requirement
for the degree of Bachelor of Science in
Physics from the College of William and Mary in Virginia,

by

Eric J. Koskinen

Advisor: Dr. David Armstrong

Dr. Keith Griffioen

Williamsburg, Virginia
May 2001

Abstract

In the Radiative Phi Decay Experiment (E-94-016) at the Thomas Jefferson National Accelerator Facility, a Lead Glass Calorimeter is used to detect high-energy photons that are produced as a result of the decay of the ϕ meson. The calorimeter is comprised of a grid of lead glass blocks which record the energy from the electromagnetic showers produced by these photons. A group of neighboring detector blocks are "clusterized" and combined so that the total energy of the high energy photon incident on the lead glass calorimeter can be determined. Since the overall detector is comprised of many smaller components, often the calorimeter remains in use despite malfunctioning components. We present an addition to the clusterizing software which estimates the amount of energy that an individual block would have measured if it was performing correctly, based on the readings of neighboring detectors and a statistical analysis of the electromagnetic showers.

Contents

List of Figures

1 Introduction and Motivation

1.1 Goal of the Experiment

In experiment E-94-016 at the Thomas Jefferson National Accelerator Facility [2], observing two rare decays of the ϕ meson, $\phi \to f_0(980)\gamma$ and $\phi \to a_0(980)\gamma$, is of primary interest. These decays will help in determining the structure of the $f_0(980)$ and $a_0(980)$ mesons [2]. The decay rates may help determine whether these mesons are comprised of the standard quark, anti-quark pair ($q\overline{q}$), or, instead, two quarks and two anti-quarks ($qq\overline{qq}$) or two mesons bound together ($K\overline{K}$) [3].

1.2 Physical Background

Currently, only two types of quark configurations have been definitively observed, the $q\overline{q}$ pair (meson) and the qqq triplet (baryon). Since these two configurations have the lowest amount of energy, they are frequently observed [3]. There are nine possible $q\overline{q}$ combinations of the lightest quarks, the up (u), down (d), and strange (s) quarks. However, the $f_0(980)$ and $a_0(980)$ particles have masses and widths, which make them appear to be too small to be any of these combinations. It has been shown [3] that measuring the branching ratios can help us determine whether the $f_0(980)$ and $a_0(980)$ particles are instead comprised of a $K\overline{K}$ pair or a four quark, $qq\overline{qq}$ configuration.

1.3 Jefferson Lab Experiment E94-016

1.3.1 Background

The Thomas Jefferson National Accelerator Facility supplies the experiment with a continuous 5.5 GeV electron beam. The experiment is located in Experimental Hall B, one of three end stations at Jefferson Lab. Hall B is equipped with a mechanism called a Tagger, which converts the electron beam to a maximum of 5.5 GeV photon beam. Electrons from the beam pass through a thin piece of material which acts as a radiator. The radiator's thickness is much less then one radiation length, thus producing photons for some fraction of the beam electrons, by means of a process called bremsstrahlung radiation. Electrons passing through the material are slowed down and, since an accelerating charge will radiate, the lost kinetic energy gives rise to photons. In the present experiment, this mechanism is used to produce the ϕ meson by the process $\gamma p \to \phi p$. Thus, the photon beam is directed at a solid Be target, which functions as a convenient source of the target protons.

1.3.2 Overall Apparatus

In this experiment, we are interested in decays of the ϕ meson. The production of the ϕ meson is only one of many possible processes that can result from a photon interacting with the Be target. In order to screen out events that do not involve the ϕ meson and events that involve other ϕ-decay modes besides the ones of interest, the experimental apparatus is somewhat involved. There are six main detectors used in the Radiative Phi Decay experiment [4] illustrated in Figure 1. The Photon tagger is the first detector and is responsible for identifying photons as they are radiated from the incident electron beam. Secondly, the Upstream Particle Veto (UPV) detects charged particles before they reach the target. Since we are only interested in photons interacting with the target, we use this detector to eliminate events that involve a charged particle in the beam. The third detector is the Barrel Scintillator detector (BSD), and is responsible for detecting the recoil proton, after it leaves the target. The Barrel Gamma Veto (BGV) detects photons which are produced at wide angles. Downstream of the target, the fifth detector, called the Charged

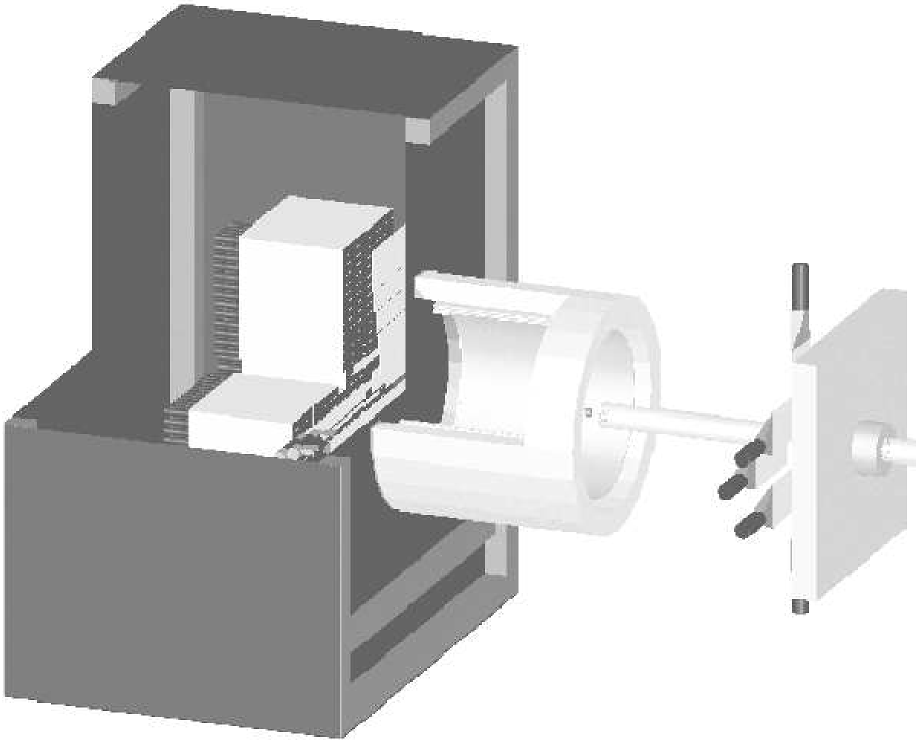


Figure 1: Detector overview (beam-line is from right to left).

Particle Veto (CPV), is used to eliminate events in which the ϕ meson radiates a charged particle. Finally, the Lead Glass Detector (LGD) is used to detect the photons that result from the decay of a ϕ meson and forms the core of the experiment. The present thesis is focused on improving the performance of this final detector.

1.4 The Lead Glass Calorimeter

1.4.1 Description, Specifications and Details

The Lead Glass Detector (LGD) is comprised of 620 individual blocks [7]. Each individual block has a 4 cm by 4 cm front face and extends 45 cm back. There is an 8 cm by 8 cm hole in the LGD face in the area directly surrounding the beam-line, which allows the portion of the beam that does not interact with the target to pass through. Figure 2 illustrates the

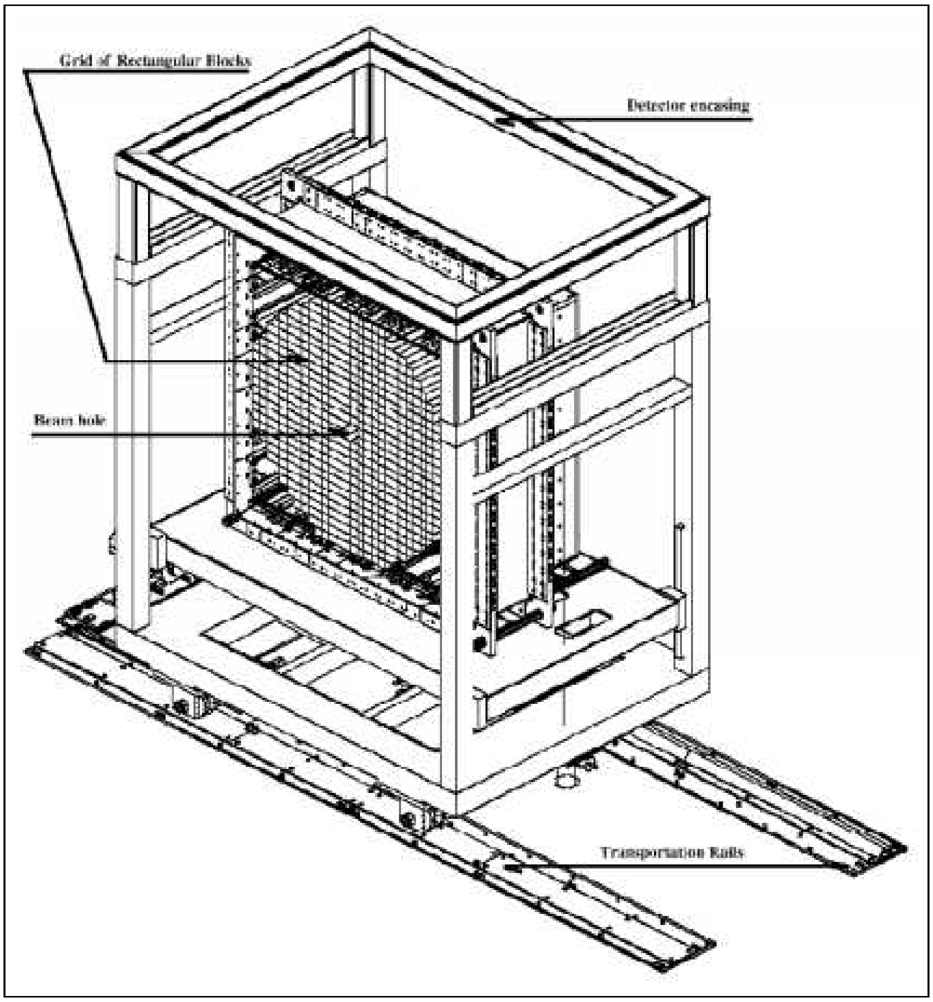


Figure 2: The Lead Glass Detector.

lead glass detector in its encasing.

The components of the LGD are used to detect the electromagnetic showers produced by the incident photons. In the LGD, high energy photons may produce electron-positron pairs. These charged particles passing near some nucleus in a material may undergo radiative energy loss called *bremsstrahlung*. As a result, more photons are produced which, in turn, may split into more electron-positron pairs. This cascade of photon and electron-positron pair production is called an electromagnetic "shower" [5].

As charged particles (such as electrons or positrons) pass through the lead glass at speeds

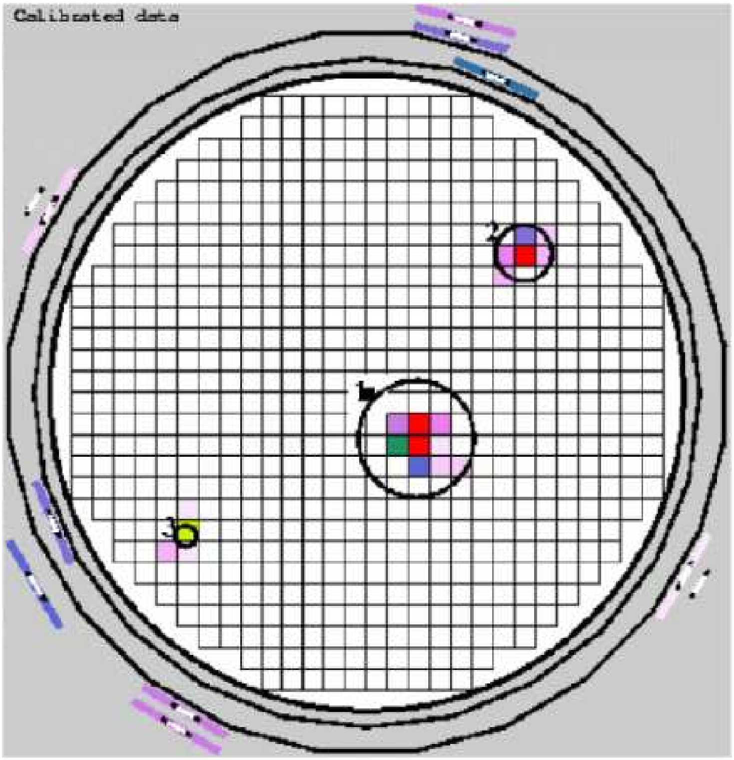


Figure 3: Clusters in the LGD from a typical event. (event 18, run 8209)

faster than the speed of light for the material, they may produce low-energy photons (visible or UV light) through a process known as Čerenkov Radiation. These low-energy photons can be detected by Photo Multiplier Tubes (PMTs) at the end of the blocks. The blocks in the LGD are designed to be more than several radiation lengths long (approximately 18 radiation lengths or 45 cm), so that the shower continues until most electrons and positrons are reduced to speeds slower than the speed of light. At this point, no more Čerenkov photons are produced [5].

At the downstream end of the LGD, PMTs convert the incident low-energy photons into a voltage which is then digitized using analog-to-digital converters (ADCs). The integrated area of the voltage pulse is proportional to the total energy of all low-energy photons detected by the PMT. Summing this energy with the energy of all neighboring PMT energies yields a value proportional to the original energy of the photon incident on the front edge of the LGD. Thus, the total energy of the photon that originally initiated the shower can be determined [4]. (See Figure 3.)

1.4.2 Dead Blocks

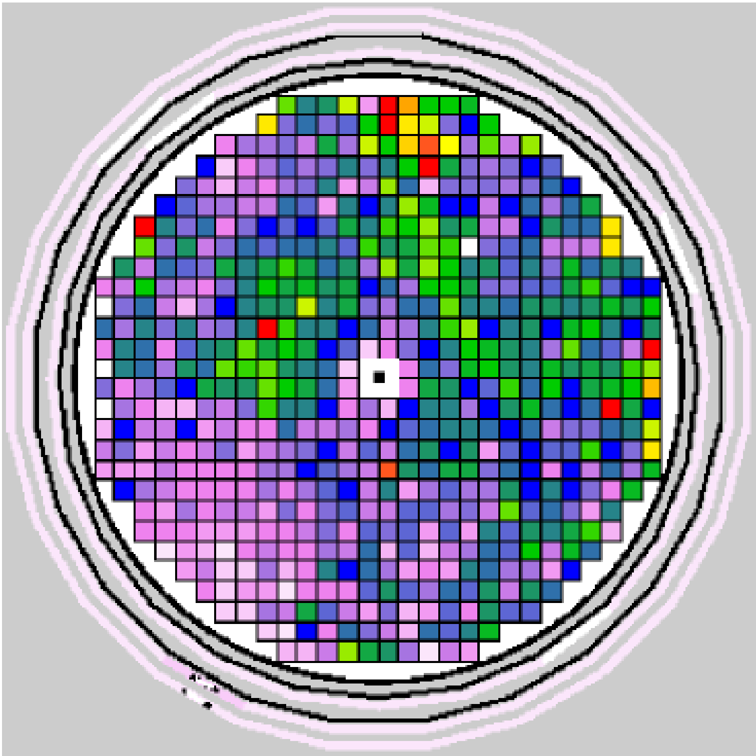


Figure 4: An LGD laser event test for run 8635.

Most of the relevant photon showers spread out across several lead glass blocks, in some cases, as many as fifteen blocks. Since the detector is comprised of many blocks, often one of them may malfunction during a run of the experiment. Before each experimental run and periodically thereafter, a test event called a "Laser Event" is generated. A laser pulse is disseminated across all of the LGD and each component reads the energy. While the laser event is not tuned finely enough to make sure that all blocks record the same energy, it at least indicates which blocks are not recording any energy. Figure 4 illustrates a test of all of the blocks of the LGD, in which several of the blocks were not responding; coloring indicates the pulse signal size and those without color did not record any energy.

However, the failure of one small portion of the LGD was not significant enough for us to consider terminating the run and spending valuable time fixing or replacing the faulty component. As a result, a large percentage of data were taken while at least a few blocks were dead. Since the shower is spread among many blocks, one is able to obtain much of the information about the shower despite the missing blocks. However, these missing blocks result in some slightly smaller estimates of total photon energy. Moreover, the missing block detracts from the ability to estimate the position of the photon as it struck the LGD since one has a somewhat inaccurate distribution of shower energy. The position is important to the experiment as it determines, along with the knowledge of the location of the target, the momentum vector of the incident photon. Both the energy (E) and the momentum ($\vec{p}$) are required to determine the photon's 4-vector, which is needed to determine the kinematics of the event.

1.5 Presented Work

In response to the inaccuracy in determining photon total energy, position, and momentum, which occurs as a result of malfunctioning lead glass blocks, we present a software correction. Based on the information we know about the surrounding blocks, within the photon shower "cluster," we attempt to estimate the energy that would have been measured by a non-faulty lead glass block.

With an effective estimation of the energy that would have been detected, we can calculate more accurately the total energy, position, and the momentum of the incident photon.

A more accurate account of the photon will help us determine more precisely the kinematics of the event.

1.6 Organization

Section 2 discusses the clusterizing software and how the correction is included. The estimation techniques are described in Section 3, and their effectiveness is analyzed in Section 4.

2 Implementation

2.1 Software Background

During experimental runs, all information produced by the detectors is stored in compacted, binary format files. We use a software program, entitled RODD [6], to unpack the compacted data, interpret the data, and produce files which can be read by histogram display software [1]. It is during this process that the values recorded by hardware detectors are interpreted.

The first phase of the process involves building a list of "hits", or blocks that recorded a non-zero energy. RODD dynamically builds a data structure called the `hitList`, which stores the lead glass block ID and the amount of energy recorded in the block. To accomplish this, RODD looks at the raw energy readings that are recorded in each block. However, the varying quality of materials comprising the blocks, and the varying environmental conditions incident on the block prevents the raw energy values from being consistent across all blocks for a given true energy value. To account for this, *calibration constants* have been calculated which are multiplied by the raw energy values (after subtraction of an offset, or *pedestal* to produce a more accurate value for the energy recorded as a result of the event.

Once the calibration constant has been applied to the raw energy reading, a cut is made, depending on whether the energy is above a minimum. Many of the blocks record energy values above zero for each event. However, in most of these uses this small amount of energy represents electronic noise or background radiation. This minimum value allows for all of the blocks which read nonzero but insignificant values to be restricted from the `hitList`.

2.2 Clusterizing

Once the `hitList` has been constructed to contain all blocks in which relevant energy was recorded, the *clusterizing* phase begins. The clusterizer applies a pattern recognition algorithm to group neighboring blocks with above pedestal energy readings into clusters. During this stage, energy hits stored in the `hitList` are combined into clusters in the `clusters` data structure. Ultimately, the total energy and position of the cluster reflects the incident high-energy photon's 4-momentum.

Clusterizing is divided into a three levels as described in the following portions of Section 2. These three phases are `levelOneSearch`, `levelTwoSearch`, and `levelThreeSearch`. In between each search, the function, `getClusterPositions` calculates the total energy, cluster center, and cluster width. The following portions of this Section 2 explain the algorithms used in each phase.

2.2.1 Level One Search

In the first phase of the clusterizer, an algorithm is used which looks through all of the individual LGD blocks to find energy hits greater than a specified minimum. Each of these high-energy block hits is used to build a unique entry in the `clusters` data structure.

2.2.2 Level Two Search

After all high energy hits have been assigned to clusters, the level two search adds additional hits that are close to the high energy hits into the same cluster. The algorithm searches through all unused hits, and if the block is close to one of the clusters defined in the level one search, it is added to that cluster. In some cases, a block is locate between two clusters and is close to each. These blocks are marked as *shared* and the energy will ultimately be shared between the two blocks. At the end of the level two search, all blocks that are not close to existing clusters are recorded in the structure, `isolatedHits`.

2.2.3 Level Three Search

The third level of the clusterizer is the new software component developed in the present work. This level is used to estimate the energy in dead blocks. The algorithm looks for blocks that are flagged as "dead" and uses the techniques described in Section 3 to estimate the amount of energy that would have been recorded.

2.2.4 Determining Cluster Positions

The `getClusterPositions` function is used to determine the physical location, total energy and width of the cluster shower, which corresponds to the momentum of the incident high energy photon. The pixelized nature of the LGD allows is fundamental in determining the location of the incident photon. Using the distribution of energy stored in individual block hits, the center of the cluster is determined [5] using a weighted mean formula

$$x_c = \frac{\sum_{j=1}^{N} w_j(E_j)x_j}{\sum_{j=1}^{N} w_j(E_j)},$$

where the sum is over all blocks in the cluster, E_j and w_j are the energy measured and weight used for block j, respectively. A similar form is used to determine y_c. One linear and two logarithmic weighting algorithms were evaluated [5] and it has been shown that the second logarithmic weighting,

$$w_j = \mathrm{Max}(0, a_0 + \ln \frac{E_j}{E_{tot}}),$$

is the least biased towards positioning the cluster in the middle of an individual block. Once the position of the photon is located, the momentum $\vec{p}$ of the photon is determined [4] as

$$\vec{p} = \frac{\vec{r_c}}{|\vec{r_c}|} E_c,$$

where r_c and E_c are the position and energy (respectively) of the cluster. A small depth correction must be applied since the incident photon travels some distance through the lead glass before the first pair production occurs [4].

2.3 Overall Structure

In practice, the level one search is used two times. The first time, the minimum energy is set to a high value so that all high-energy clusters are located. Once these blocks and their neighbors have been associated into clusters, a second level one search with a lower minimum energy locates low-energy clusters using blocks that have not yet been associated. Figure 3 illustrates individual block hits, notated by coloring, and their associated clusters, notated by black circles. In summary, the search levels are used in the following order:

$$\boxed{\texttt{levelOneSearch}} \longrightarrow \boxed{\texttt{getClusterPositions}} \longrightarrow \boxed{\texttt{levelTwoSearch}} \longrightarrow \boxed{\texttt{levelOneSearch}} \longrightarrow$$
$$\boxed{\texttt{getClusterPositions}} \longrightarrow \boxed{\texttt{levelThreeSearch}} \longrightarrow \boxed{\texttt{getClusterPositions}}$$

2.4 Presented Software Additions

We present an addition to the software in both the `levelThreeSearch` as well as during the generation of the `hitList`, to incorporate dead blocks. After the `hitList` has been built, dead blocks are appended to the list. These blocks, however, are marked as dead and assigned zero for the energy reading. The clusterizer is designed such that it assumes anything in the `hitList` has an above-threshold energy reading. Thus, while dead blocks are not recognized as high-energy cluster centers, they will automatically be incorporated into the cluster that they are neighbors of.

Once the clusters have been isolated, the estimation technique described in Section 3 is applied to all blocks marked as dead. Finally, the `getClusterPositions` function is used one more time. After this phase is complete, the total energy, position, and momentum of the incident photons are identified and the analysis of the source ϕ meson proceeds.

3 Estimating Dead Block Energy

The crucial step in accounting for dead blocks is estimating the energy that would have been recorded in the block, had it not been faulty. In this section, we describe several estimation techniques that were explored and how they are used within the context of the software. Each of the estimation techniques were evaluated by temporarily "turning off" properly functioning blocks and comparing the energy estimation with the actual value.

3.1 Simple Average Method

The most basic method used was the simple average method. To estimate the energy in the block, an average of all neighboring block energies is used. Thus, the energy estimated for a dead block i is calculated using all N neighbors as

$$E_i = \frac{\sum_{j=1}^{N} E_j}{N}.$$

This technique ensures that the estimated energy will be on the same order as all neighbors. Figure 5 illustrates a typical cluster in which this method is used (from event three in run number 8209). Unfortunately, although the estimated value of 253 MeV is on the same order as the actual energy value of 160 MeV, this method is not as accurate as the following methods.

3.2 Weighted Average Method

The next method of estimating dead block energy involves determining a weighted average of neighboring blocks. Since the clusters seem to exhibit a large fraction of the total energy in the most central blocks and smaller fractions for those blocks further from the center,

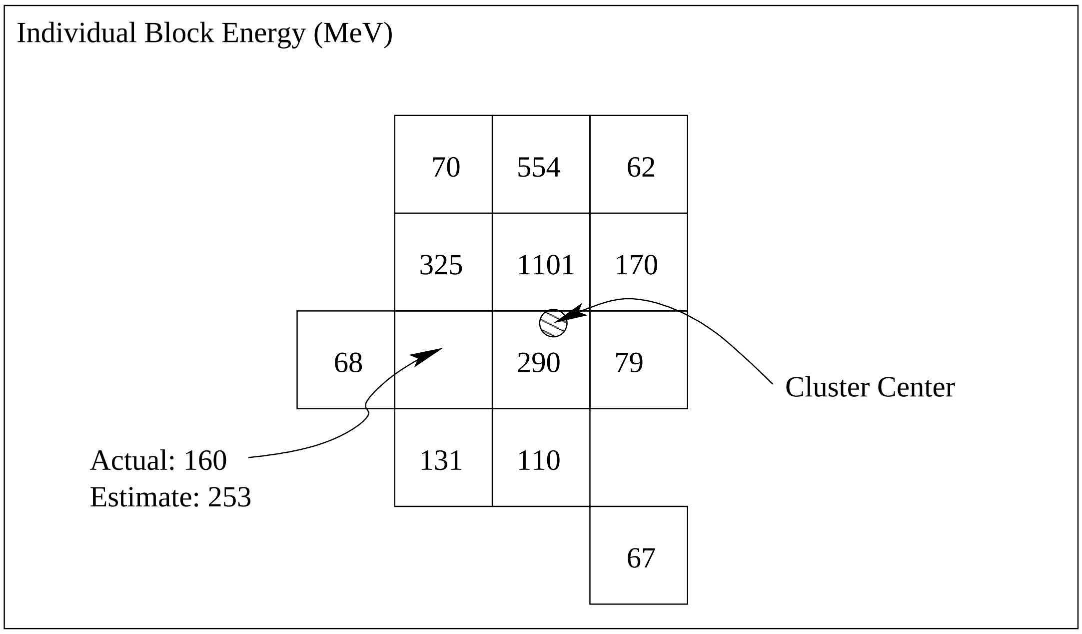


Figure 5: Sample effect of the simple average method.

our average gives more weight to those blocks which are at similar distances from the center of the cluster. Conversely, those blocks which are at dissimilar distances from the center of the cluster are given less weight in the overall average of the blocks.

To accomplish this, we begin by computing the differences in distances from the cluster center, $\hat{C}$, between the dead block $\hat{A}$, and each neighboring block, $\hat{B}_j$. These differences as calculated as

$$d_i = |\|\hat{C} - \hat{A}\| - \|\hat{C} - \hat{B}_j\||.$$

The sum of these distances is, $d = \sum_{j=1}^{N} d_i$. Thus, we compute a fractional distance difference for each neighbor as $f_i = d_i/d$.

Each of these fractions is, however, incorrectly weighted; those blocks at similar distances as the dead block are given less weight then those at more dissimilar distances. To resolve this, we take the complement of each fractional distance, $g_i = 1 - f_i$. To normalize the sum of the weights back to one, each complement is then divided by the sum of the complements as follows to determine h_i:

$$h_i = \frac{g_i}{\sum_{j=1}^{N} g_j}.$$

The set of weights, defined as $\{h_1, \ldots, h_N\}$, is then used to scale individual block energies when computing the dead block energy estimate. Thus the estimated energy E_i is calculated as

$$E_i = \sum_{j=1}^{N} h_j \cdot E_j,$$

where E_j is the energy recorded in neighboring block, j.

3.3 Fitting Method

A third method we investigated involves fitting the cluster energy distribution to a shower shape. Given a function that accurately models a typical shower, we extrapolate an energy estimate for a block at a particular position from the cluster center. In order to develop a shower shape model, we performed a study on the shapes of typical photon showers.

We began by developing histograms which illustrate the relationship between the energy in a single block of the cluster, e_i verses the distance of that block from the center of the cluster, r_i. Since showers have a great deal of variance in total energy, we did not want the curve to depend on the total energy. Moreover, the cluster width is much greater for higher energy showers. So, instead we looked for a relationship between the fractional

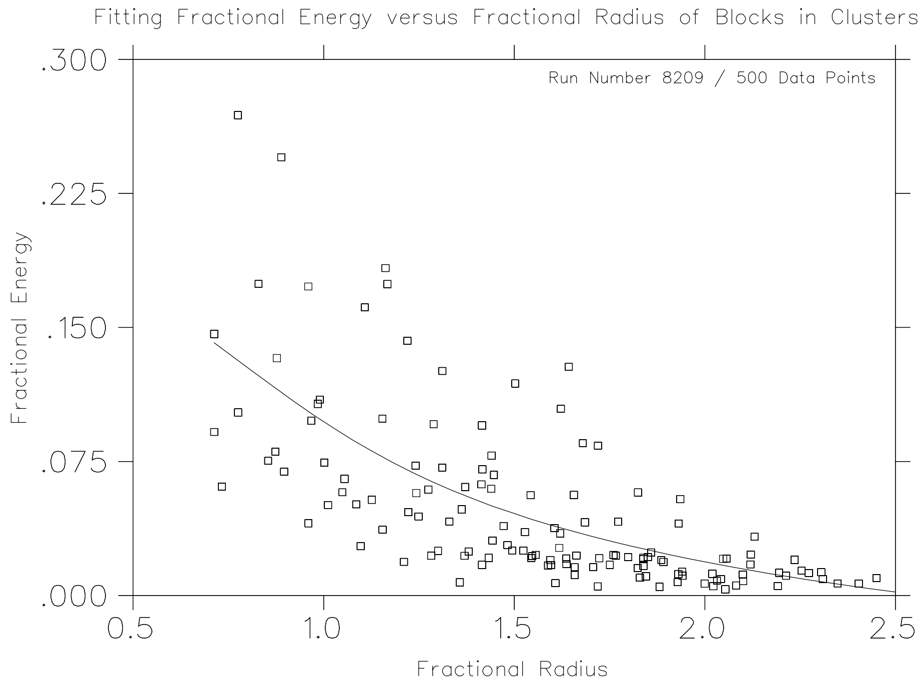


Figure 6: Fitting e_{fi} versus r_{fi} for blocks in various clusters.

energy in the block (or, the block energy divided by total cluster energy), e_{fi}, and the fractional radius of the block from the center of the cluster (or, distance of the block from the center divided by the width of the cluster), r_{fi}. The width we used is calculated by the `getClusterPositions` function which determines the standard deviation of the shower distribution. Figure 6 reveals a strong relationship between e_{fi} and r_{fi}. We discuss the functional forms used to model this in the final portion of Section 3.

Figure 6 illustrates how individual data points are represented with a global fit. Although the curve passes through the middle of the data points, there is a large degree of error involved in reducing the distribution to a single curve. In order to reduce the error, the curve must be parameterized, so that the curve is customized to an individual shower. In the following portions of this section, we study how the shower shape depends on several important parameters. For example, if the shower shape is quite different depending on the

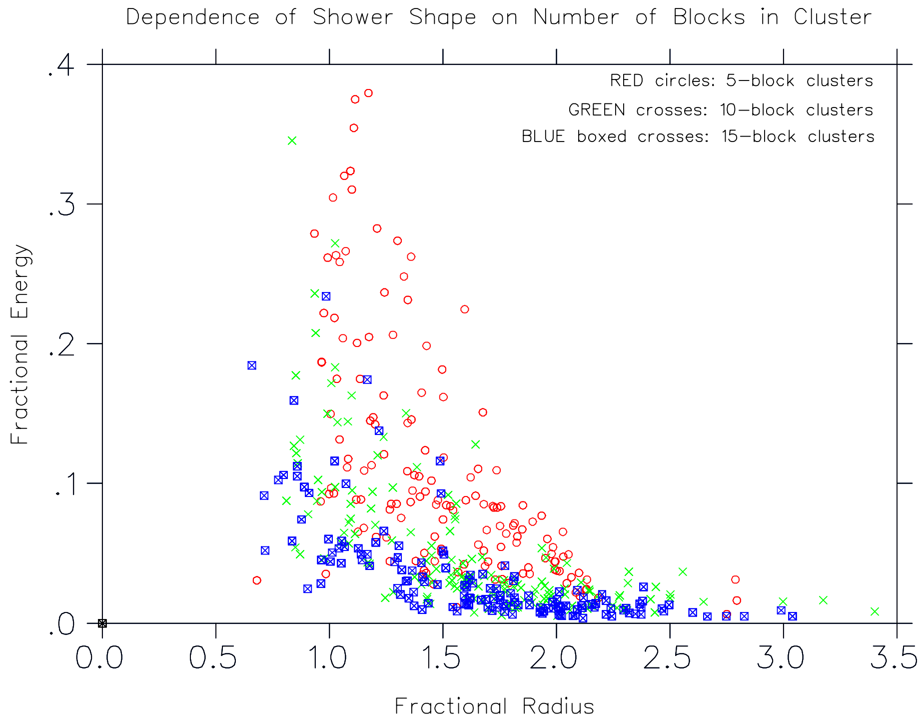


Figure 7: Dependence of Shower Shape on Number of Blocks in the Cluster.

number of blocks in the cluster, then we need to include a parameter in the curve to adjust the shape of the curve based on the number of blocks in the cluster.

3.3.1 Number of Blocks in the Cluster

We examine how the relationship between e_{fi} and r_{fi} depends on the total number of blocks present in the cluster. To determine whether or not this dependence exists, we examine the relationship between e_{fi} and r_{fi} for varying number of blocks in the cluster. Figure 7 illustrates the dependence of shower shape on the number of blocks in the cluster. Note that the absence of data points where $r_{fi} < 0.6$ is a result of the granularity of the LGD. Section 3.5.1 discusses these limitations.

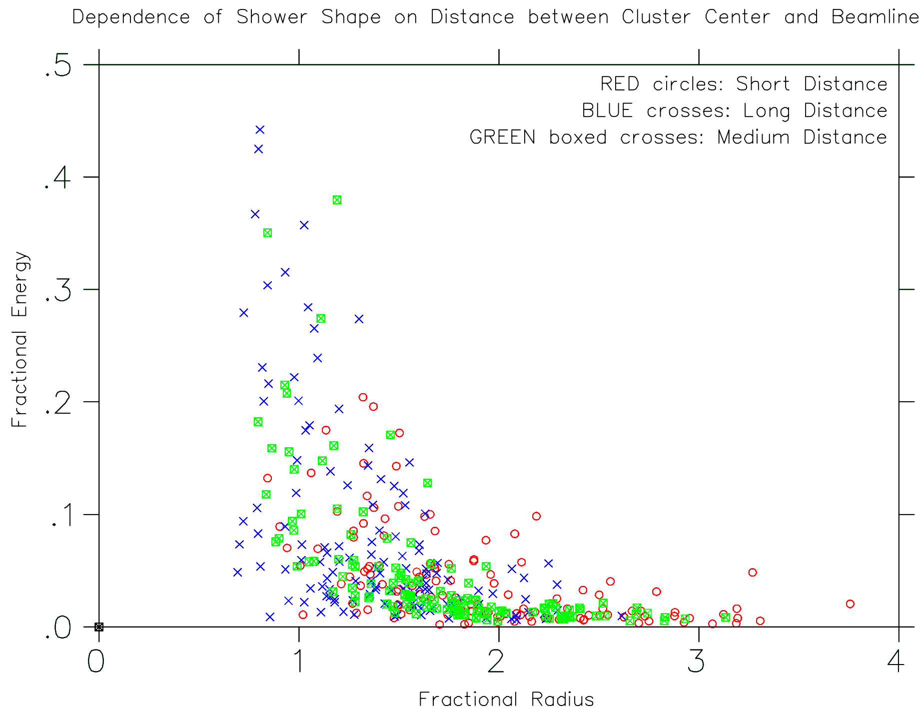


Figure 8: Dependence of shower shape on distance of cluster from beam-line.

3.3.2 Distance Between Cluster Center and Beam-line

We examine how the relationship between e_{fi} and r_{fi} depends on the distance of the cluster center from the center of the beam-line. Since the geometry of the apparatus includes a point target and a single-plane lead glass detector, the shape of the shower may be different for photons that are produced at a wider angle. In order to find this dependence, we examine plots which illustrate the shower shape for several distances from the beam-line. Figure 8 demonstrates only a small dependence of shower shape on the distance of the cluster center from the beam-line. Thus we have decided not to base the functional form on a parameter corresponding to distance from the beam-line.

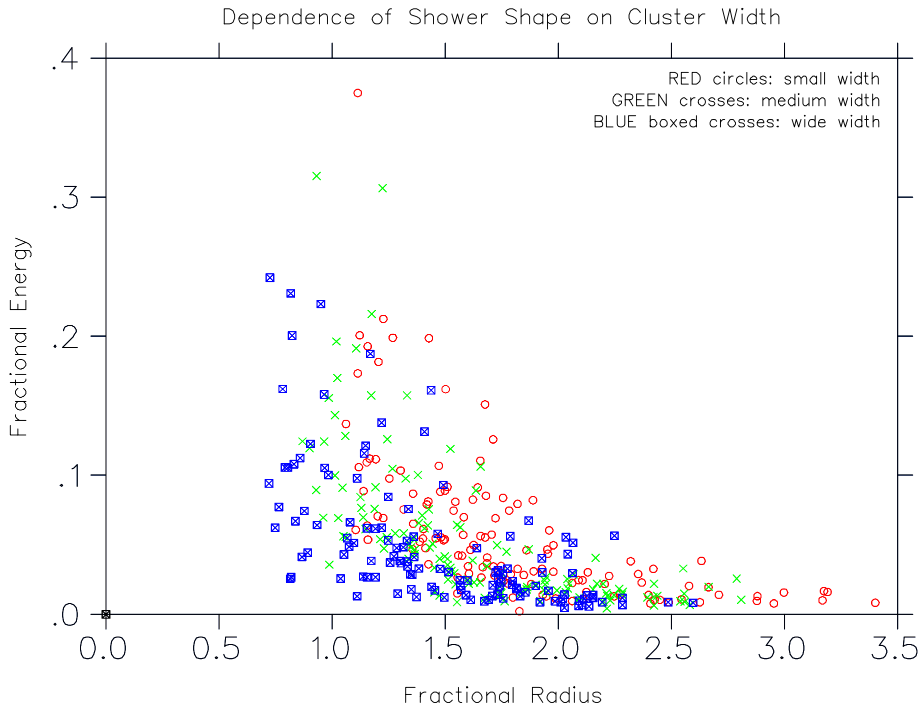


Figure 9: Dependence of shower shape on cluster width.

3.3.3 Cluster Width

A final parameter we examine for curve dependence is the width, or standard deviation of the distribution. Figure 9 illustrates the strong dependence of e_{fi} versus r_{fi} on the cluster width. The cluster width is one of the more significant parameters since it is computed directly based on the shower shape.

3.4 Curve-Fit Equation

3.4.1 Functional Forms

In order to fit the shower shape distribution to a curve, we compare the effectiveness of the following empirical models:

$$e_{fi} = (a/r_{fi}) + (b/r_{fi}^2) + c \tag{1}$$

$$e_{fi} = (a * r_{fi}) + (b * r_{fi}^2) + (c * r_{fi}^3) + d \qquad (2)$$

where $a, b, c,$ and d are variable parameters. Figures 10 and 11 illustrates how data points are approximated using a fit to each of these forms. We chose to use Eqn. (1) as it more accurately captures the shape of typical showers, particularly since it approaces zero at a large fractional radius.

3.4.2 Parameter Dependence

In order to accommodate shower shape differences that arise from variables such as the cluster width into the curve equation, we determine how each parameter, $a, b, c, \ldots$, depends on the number of blocks, distance from the beam-line, and cluster width. Figure 12 illustrates how each parameter depends on the cluster width. The crosses in the figure illustrate how the fit approximates parameter values.

While we examined how each parameter varies depending on the number of blocks and on the distance from the beam line, the cluster width presented the most striking correlation. As a result, we chose to vary the functional form based on the cluster width. Figure 13 illustrates how our functional form varies with the width of cluster based on these parameters.

3.4.3 Fitting Method Implementation

In order to estimate the energy in a dead block using the fitting method, the following algorithm is used.

1. Determine the distance, d_i between the dead block and the cluster center
2. Calculate $r_{fi} = d_i/\sigma$, where σ is the cluster width.
3. Calculate the parameters for the fit based on the cluster width as: $a = 0.406*\sigma - 1.493$, $b = -0.363 * \sigma + 1.467$, and $c = -0.121 * \sigma + 0.431$.

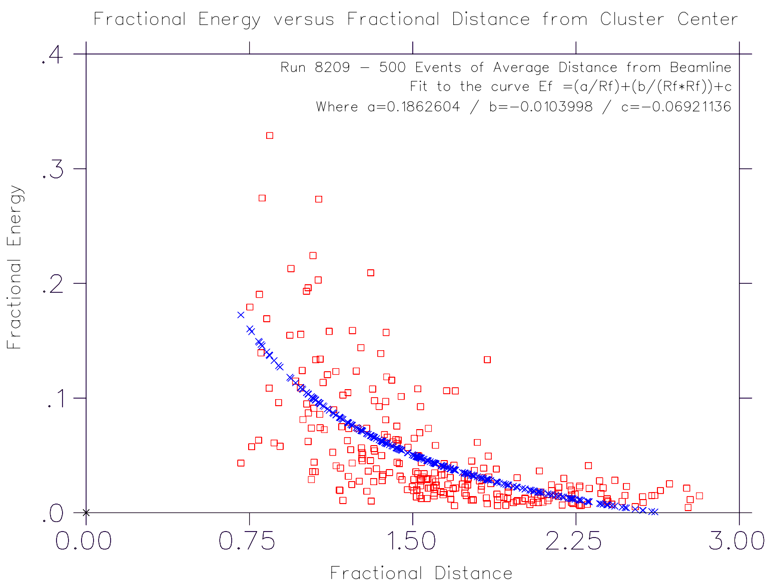


Figure 10: Fitting e_{fi} versus r_{fi} to Eqn (1).

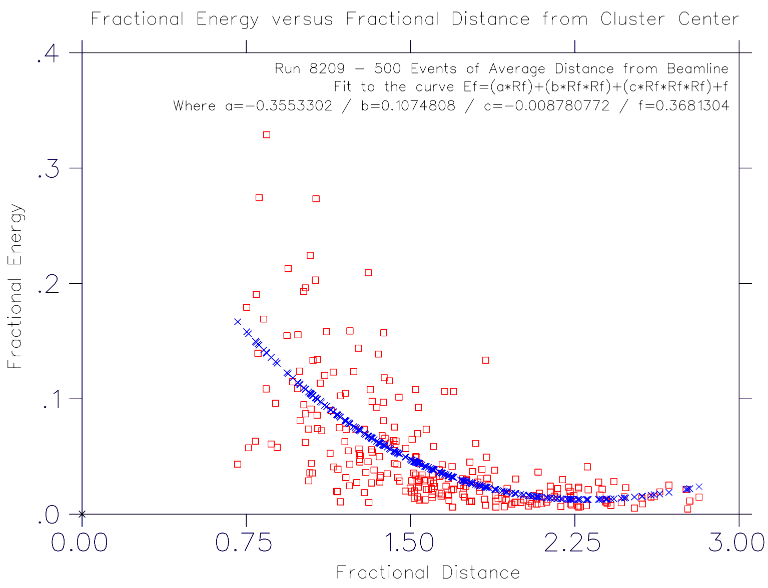


Figure 11: Fitting e_{fi} versus r_{fi} to Eqn (2).

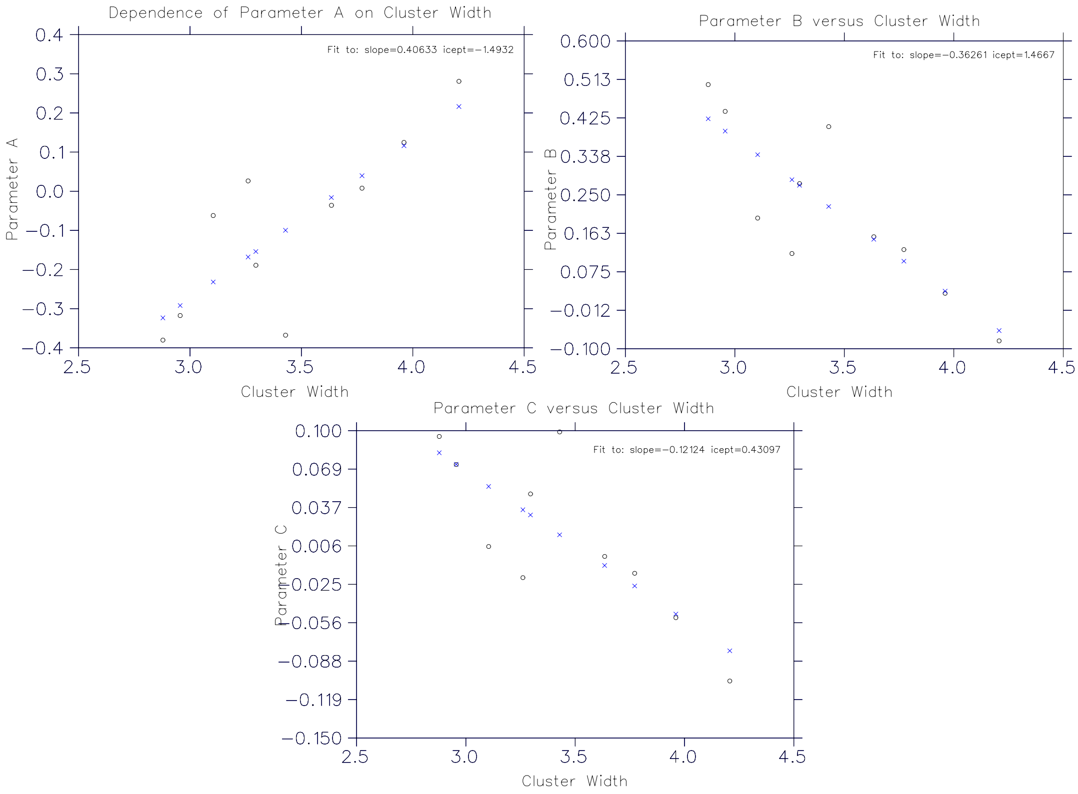


Figure 12: Individual parameter values versus cluster width (cm).

4. Determine $e_{fi} = a/r_{fi} + b/r_{fi}^2 + c$.
5. Calculate the amount energy that should have been recorded as $e_i = E_t * e_{fi}$, where E_t is the total energy of the cluster.

3.5 Special Considerations

3.5.1 Few-Block Clusters

Most of the methods that we present require a well behaved energy distribution in the cluster in order to be effective. We have seen that for cases where there are less than three blocks in the cluster, it is very unlikely that we will be able to establish a reasonable block energy prediction. In these cases, the dead block may be the block that would have been the center of the cluster. Since these clusters are narrow, we cannot develop an accurate

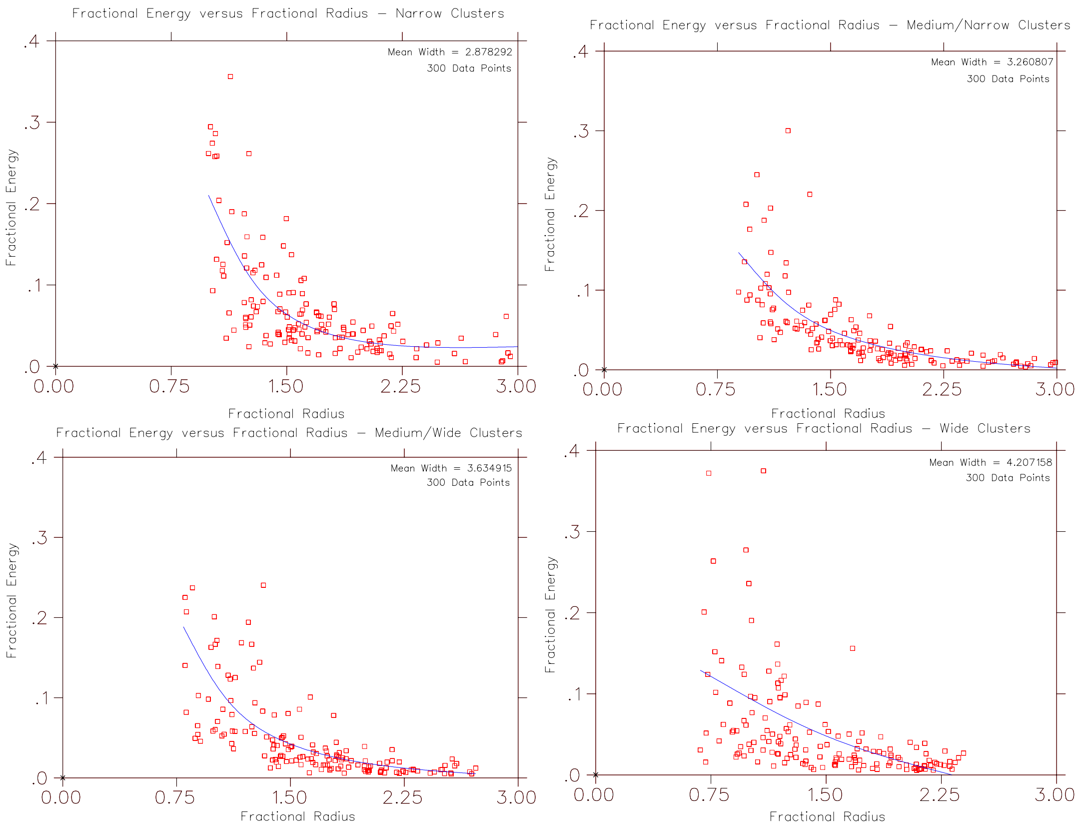


Figure 13: Functional form variations based on cluster width.

account of the shape of the cluster and therefore, the peak energy of the cluster. Thus, we only apply our method to clusters in which the total number of blocks is greater than three.

3.5.2 Detector limitations

We are also limited by the size of the overall detector. Clusters along the edge often "leak over" beyond the outer lead glass blocks, preventing us from determining the portion of the energy that was lost. Similarly, the inner ring of the detector has a hole through which the beam passes. High-energy photons may strike the LGD along the rim of the hole. In both of these regions, it is difficult to find a curve to represent the clusters, and, therefore, the dead block energy predictions will suffer as well. Since much of the general data error results from these clusters and there is little we can do to improve these clusters, we accept them as they are without trying to improve the data.

3.5.3 Block Sharing

A third consideration involves overlapping clusters. Often two photons that are incident on the LGD have not had enough distance to spread apart sufficiently. Thus, many clusters overlap neighboring clusters. The first hindering effect of this phenomenon is that the energy measure in the blocks of the individual clusters do not represent the energy of the individual incident photon. Therefore, the entire cluster is somewhat faulty to begin with. Moreover, if a dead block resides between the two clusters, two energy predictions must be made, one for each cluster, and then combined for the block. This becomes even more nebulous when three or more clusters overlap. In most cases, the estimation technique resolves the block sharing issue. In our technique, the energy in a shared block is estimated twice, once for one cluster, and then erased and estimated again for the other cluster.

4 Results

4.1 Testing Effectiveness of Techniques

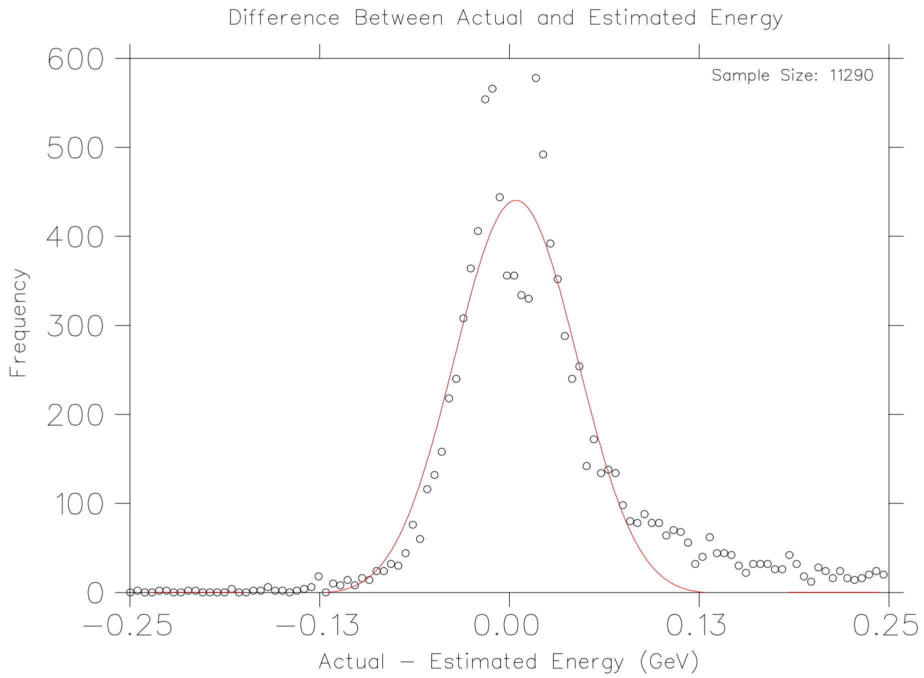
In order to test the effectiveness of this method, we need to "invent" dead blocks. We consider a normal run in which there are no dead blocks and arbitrarily chose several blocks throughout the lead glass detector to ascribe as "dead." We compare the effectiveness of each estimation technique in the following two manners.

4.1.1 Estimation Comparisons

We examined the effectiveness of the estimation technique by comparing the estimated and the actual energy for a number of dead blocks. Based on $10,000$ events from run 8209, we histogram the difference between the actual and the estimated block energy, as illustrated in Figure 14. As expected, the mean difference $\langle E_{act} - E_{est} \rangle$ is nearly zero and the standard deviation σ is rather small. These statistics signify that on average, our technique provided an appropriate estimate.

4.1.2 Improvement in Invariant Mass Determination

Since a large percentage of events involve the process $\pi^0 \rightarrow \gamma\gamma$, we will test the effectiveness of the methods by comparing histograms of the invariant mass two photon combinations. Figure 15 illustrates a histogram of two photon masses ($M_{\gamma\gamma}$) for an analysis in which no dead block estimation techniques were used. Each technique has a different effect on the results, and the most effective technique would ideally produce a more narrow peak. Moreover, the center of the peak reflects the mass of the source π^0 whose mass is known. Thus, if our methods are effective, we will improve the invariant mass plots, by shifting the centroid of the π^0 histogram closer to the accepted mass of a π^0.



$$\begin{aligned} \sigma &= 53.4878 \text{ MeV} \\ \langle E_{act} - E_{est} \rangle &= 0.004246 \text{ MeV} \\ Area &= 61,164 \end{aligned}$$

Figure 14: Difference between actual and estimated energy.

To make these statistical comparisons, we use the same source data (run number 8209) and examine the shape of the histogram for each technique. We compare these histograms with those produced from the same run for a situation in which no dead blocks were invented and a situation in which all dead blocks are left at zero, with no estimation applied. Thus, we compare the statistics for the following five different software configurations:

1. **Ideal**: No invented dead blocks.
2. **Dead**: All dead blocks left with zero energy.
3. **Simple Average**: Dead blocks given energy estimated by the simple average method.
4. **Weighted Average**: Dead blocks given energy estimated by the weighted average method.

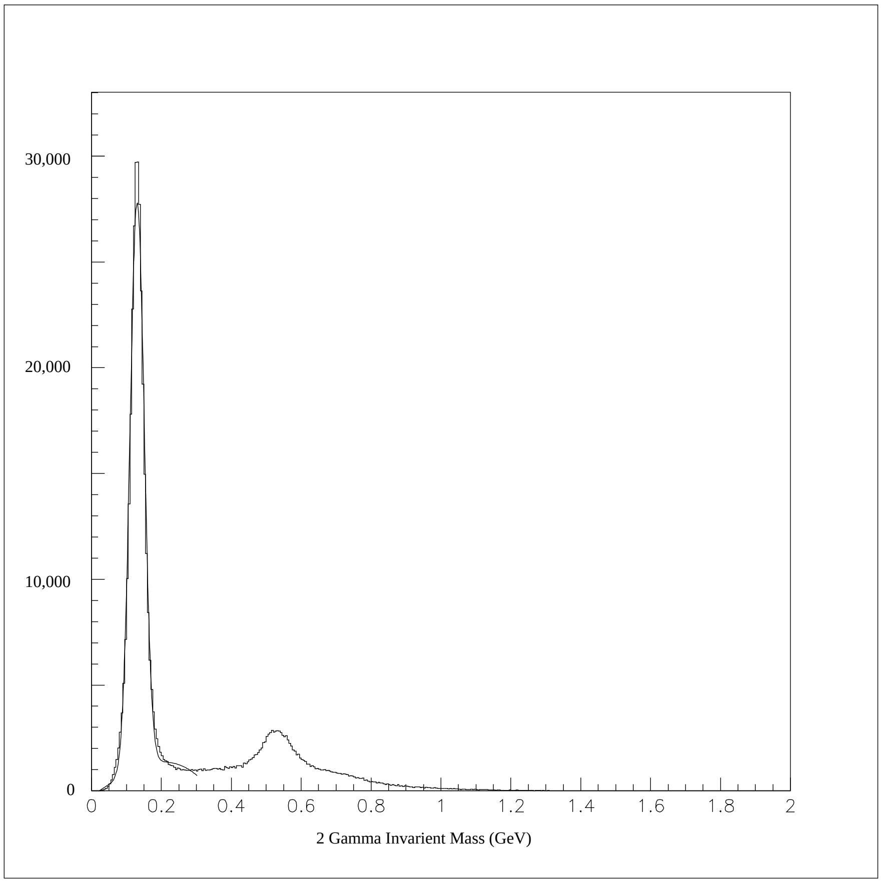


Figure 15: Invariant mass $M_{\gamma\gamma}$ histogram for run 8209.

5. **Curve Fit**: Dead blocks given energy estimated by the curve fitting method.

Figure 16 illustrates the histogram properties for each of the above configurations based on a sample of $1,150,000$ events from run number 8209. The first two configurations provide a framework for comparing the methods. The statistics for a perfect estimation technique would be identical to those gathered for the ideal configuration. All estimations for dead blocks would be precisely what was actually recorded and, thus, there would be no difference between the ideal configuration and the perfect method configuration.

Comparing the Ideal to the Dead configuration, the statistics are significantly skewed. Clusters with dead blocks in them receive no estimation for the dead block and the total

Configuration	Events in Peak	Area of Peak	Centroid (GeV)	Width (GeV)
Ideal	26,746	1,360.02	0.13065	0.20286E-01
Dead	25,707	1,343.98	0.12877	0.20857E-01
Simple Average	25,302	1,339.11	0.13056	0.21114E-01
Weighted Average	25,681	1,344.42	0.12895	0.20885E-01
Curve Fit	26,331	1,353.83	0.13010	0.20512E-01

Figure 16: π^0 statistics for each software configuration.

energy sum is less than the total energy of the incident photon. Often this causes a cluster to “fall out” below the peak, decrementing the number of events within the peak. Those that do not fall out of the peak, help move the centroid to the left, further from the true value. Finally, the width of the peak in the dead configuration is greater, representing more variance in the statistics.

Neither averaging technique is effective at reviving the ideal statistics. In both cases, the width is larger that that of the ideal Gaussian, and the centroid is less than the ideal. While the simple average method does estimate the centroid pretty well, this indicates that individual estimates are generally either too high or too low, but equally so. Unlike the averaging techniques, the curve fitting method seems to be the most effective. Of all of the techniques, it is the only one in which the area and the width of the Gaussian are better than when all dead blocks are left at zero. Thus, we have chosen to incorporate the curve fitting method into the software to estimate dead block energy.

5 Conclusions

The functionality of the lead glass detector is essential in developing the kinematics of incident high-energy photons. The momentum and energy of the photon is derived from the location and total energy of clusters. Each individual block directly contributes to the distribution of the cluster and, ultimately, the kinematics of the incident photon. As we have seen, the parameterized fitting method is the most effective at predicting dead block energy when compared with the simple average and weighted average techniques. We have shown that providing dead block energy estimates in the clusterizing software improves the overall experimental results.

References

[1] D. S. Armstrong. *RODD for Radiative Phi* `http://www.jlab.org/ radphi/Software/RODD.html`.

[2] H. C. *et al.* CEBAF proposol e-94-016 proposal to PAC-8. *(unpublished)*, 1994.

[3] S. K. F. Close, N. Isgur. B389. *Nuclear Physics*, 1993.

[4] L. J. Kaufman. Recoil proton detection for radiative decays of the ϕ meson. Senior Thesis, College of William and Mary.

[5] R. A. Lindenbusch. *An Analysis of* $\pi^- p \to \eta\pi^0 n$ *at* 18 GeV/c. PhD thesis, Indiana University, Department of Physics, 1996.

[6] T. O'Connor. Off-line data analysis software for TJNAF e94-016. Master's thesis, April 1997.

[7] E. Scott. *The Lead Glass Detector* `http://www.jlab.org/ radphi/LGD/LGD.html`.

A Presented Software

A.1 Revised MakeHits

makehits/make_lgd_hits.c

```
/*     make_lgd_hits.c     */
/*     D.S. Armstrong    March 9 2000  */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
#include <itypes.h>
#include <eventType.h>
#include <ntypes.h>
#include <disData.h>
#include <lgdCluster.h>
#include <lgdUtil.h>
#include <lgdGeom.h>
#include <pedestals.h>
#include <calibration.h>
#include <makeHits.h>

/*
 * This routine will calculate the hits in the LGD, where a hit requires
 * an ADC value sufficiently over threshold, and the hit value is in
 * energy units (GeV), i.e. a calibration has been applied.  The group
 * GROUP_LGD_HITS is then created.  The routine setup_makeHits must be
 * called some time before this routine is invoked, to load geometry and
 * calibrations appropriate for the particular run number.
 */

int make_lgd_hits(itape_header_t *event,
                  lgd_hits_t *p_lgd, int maxhits)
{

  adc_values_t *lgd=NULL;
  lgd_hits_t *hits=NULL;
  int index, index2, flag, found, idx;

  int LGD_db_nChannels, LGD_db[20]; /* deadblocks */

  /* These dead channels will be appended */
  LGD_db_nChannels = 14;
  LGD_db[0] = 459; LGD_db[1] = 611;
  LGD_db[2] = 329; LGD_db[3] = 321;
  LGD_db[4] = 90;  LGD_db[5] = 629;
  LGD_db[6] = 277; LGD_db[7] = 314;
  LGD_db[8] = 303; LGD_db[9] = 311;
  LGD_db[10] = 661; LGD_db[11] = 120;
  LGD_db[12] = 582; LGD_db[13] = 72;
```

```
if(p_lgd==NULL){
  hits = malloc(sizeof_lgd_hits_t(maxhits+LGD_db_nChannels));
}
else{
  hits = p_lgd;
}

hits->nhits = 0;

if( (lgd = data_getGroup(event,GROUP_LGD_ADCS,0)) == NULL){
  static int first_time=1;
  if(p_lgd==NULL)
    free(hits);
  if (first_time && !monte_carlo) {
    fprintf(stderr,"Warning in make_lgd_hits: No LGD ADC's\n");
    first_time=0;
    return(0);
  }
  else return(0);
}
for(index=0;index<lgd->nadc;index++){

    if((((float)lgd->adc[index].value) -
        lgd_ped[lgd->adc[index].channel])>8.0 ){
      if(hits->nhits < maxhits){
        hits->hits[hits->nhits].channel = lgd->adc[index].channel;
        hits->hits[hits->nhits].energy = (lgd->adc[index].value -
              lgd_ped[lgd->adc[index].channel])*
          lgd_cc[lgd->adc[index].channel];
        hits->hits[hits->nhits].flags = 0;
        (hits->nhits)++;
      }else{
        fprintf(stderr,"Warning in make_lgd_hits: ");
        fprintf(stderr," buffer full at %d hits, truncating.\n",
                maxhits+1);
        break;
      }
    }

}

if(LGD_DB_METHOD != LGD_DB_NONE) {
  /* add dead blocks to the hit list */
  for(index=0; index<LGD_db_nChannels; index++) {
    if(hits->nhits < maxhits){
      /* see if we already have one */
      found = 0;
      for(idx=0;((idx<hits->nhits)&&(found==0));idx++) {
        if(hits->hits[idx].channel == LGD_db[index])
          found = idx;
      }
      /* clear it out if its already there */
      if(found) {
```

```
          hits->hits[found].energy  = 0;
          hits->hits[found].flags   |= LGD_F_DEADBLOCK;
        } else {
          /* add a new hit */
          hits->hits[hits->nhits].channel = LGD_db[index];
          hits->hits[hits->nhits].energy  = 0;
          hits->hits[hits->nhits].flags   |= LGD_F_DEADBLOCK;
          (hits->nhits)++;
        }
      } else {
        fprintf(stderr,"Warning in make_lgd_hits: ");
        fprintf(stderr," buffer full at %d hits, truncating.\n",
                maxhits+1);
        break;
      }
    }
  }

  if(p_lgd==NULL) {
    int size = sizeof_lgd_hits_t(hits->nhits);
    lgd_hits_t *tmp = data_addGroup(event,BUFSIZE,
                                    GROUP_LGD_HITS,0,size);
    memcpy(tmp,hits,size);
    free(hits);
  }

  return(0);
}
```

A.2 Additional Level Three Search

libLGD/lgdClusterIU.c:

```
void levelThreeSearch_DeadBlocks(lgd_hits_t *hits,
                   lgd_clusters_t *clusters,
                   float fractionClusterized,
                   lgd_hit_t **hitList,
                   lgd_hits_t *isolatedHits,
                   float scaling)

  /* Search through all clusters for blocks that are marked dead
     and estimate the energy that would have been recorded in them. */
{

  int         clus_idx, block_idx, k, p, q;
  float       guessE, totaldist, Rfrac, Efrac, dist;
  int         neighbor[8], nNeighbors;
  vector3_t   ithblock, jthblock;
  FILE       *f;

  if(clusters->nClusters == 0)     return;
  if(LGD_DB_METHOD == LGD_DB_NONE) return;

  /* Look through all blocks in all clusters for dead blocks ... */

  for(clus_idx=0; clus_idx<clusters->nClusters; clus_idx++) {
    for(block_idx=1;block_idx<clusters->cluster[clus_idx].nBlocks;block_idx++) {

      /* if the block is dead */
      if(hitList[clus_idx][block_idx].flags == 205) {

          /* flag the cluster as modified */
          clusters->cluster[clus_idx].flags |= LGD_F_DEADBLOCK;
          guessE = 0.0;

          switch(LGD_DB_METHOD) {


          case LGD_DB_BLANK: /* === Dead Blocks - Don't put in energy */
            guessE = 0.0;
            break;

          case LGD_DB_FIT1:  /* === Dead Blocks - ORIGINAL Fitting Method (1) */
            if(lgdLocalCoord(hitList[clus_idx][block_idx].channel, &ithblock)
               != LGDGEOM_OK)
              printf("Channel Error!\n");
            totaldist =
              sqrt(pow(ithblock.x-clusters->cluster[clus_idx].space.x,2)+
                   pow(ithblock.y-clusters->cluster[clus_idx].space.y,2));
            guessE = 0.75*(1/pow(totaldist,2)) + 0.25*(1/(totaldist));
            break;
```

```
        case LGD_DB_FIT2:  /* Dead Blocks - New Fitting Method (2)*/
          if(lgdLocalCoord(hitList[clus_idx][block_idx].channel, &ithblock)
             != LGDGEOM_OK)
            printf("Channel Error!\n");
          if(clusters->cluster[clus_idx].nBlocks <= 5) break;
          totaldist =
            sqrt(pow(ithblock.x-clusters->cluster[clus_idx].space.x,2)+
                 pow(ithblock.y-clusters->cluster[clus_idx].space.y,2));
          Rfrac = totaldist / clusters->cluster[clus_idx].width;
          if(Rfrac<0.9 || Rfrac > 2.25) break;
          Efrac = DBFit2(Rfrac, clusters->cluster[clus_idx].width);
          guessE = clusters->cluster[clus_idx].energy * Efrac;
          break;

        case LGD_DB_WEIGHTAVG: /* === Dead Blocks - Weighted Average Method */
          if(lgdGeomGetNeighbors(hitList[clus_idx][block_idx].channel,
                                 &neighbor[0], &nNeighbors)
             == LGDGEOM_CHANNELERROR) {
            printf("Channel Error!\n");
          }
          if(clusters->cluster[clus_idx].nBlocks <= 5) break;
          for(p=0;p<nNeighbors;p++) {
            for(q=0;q<hits->nhits;q++) {
              if(hits->hits[q].channel == neighbor[p]) {
                if(lgdLocalCoord(hits->hits[q].channel, &ithblock)
                   != LGDGEOM_OK)
                  printf("Channel Error!\n");
                totaldist +=
                  sqrt(pow(ithblock.x-clusters->cluster[clus_idx].space.x,2)+
                       pow(ithblock.y-clusters->cluster[clus_idx].space.y,2));
              }
            }
          }
          for(p=0;p<nNeighbors;p++) {
            for(q=0;q<hits->nhits;q++) {
              if(hits->hits[q].channel == neighbor[p]) {
                if(lgdLocalCoord(hits->hits[q].channel, &ithblock)
                   != LGDGEOM_OK)
                  printf("Channel Error!\n");
                dist =
   sqrt(pow(ithblock.x-clusters->cluster[clus_idx].space.x,2)+
pow(ithblock.y-clusters->cluster[clus_idx].space.y,2));
                guessE += (hits->hits[q].energy * (dist/totaldist));
              }
            }
          }
          break;

        case LGD_DB_SIMPLEAVG: /* === Dead Blocks - Simple Average Method === */
          if(lgdGeomGetNeighbors(hitList[clus_idx][block_idx].channel,
                                 &neighbor[0], &nNeighbors)
             == LGDGEOM_CHANNELERROR) {
            printf("Channel Error!\n");
```

```
          }
          if(clusters->cluster[clus_idx].nBlocks <= 5) break;
          for(p=0;p<nNeighbors;p++) {
            for(q=0;q<hits->nhits;q++) {
              if(hits->hits[q].channel == neighbor[p]) {
                guessE += hits->hits[q].energy;
              }
            }
          }
          guessE = guessE / nNeighbors;
          break;

        }

        /* never guess more than 30% of total cluster energy */
        if(guessE / clusters->cluster[clus_idx].old_e > 0.3)
           guessE = 0.3 * clusters->cluster[clus_idx].old_e;

        /* update data structures with estimation */
        hitList[clus_idx][block_idx].energy  = scaling*guessE;
        clusters->cluster[clus_idx].energy   =
           clusters->cluster[clus_idx].old_e + scaling*guessE;

    } /* endif */
  } /* endfor */
 } /* endfor */
}
```

A.3 Functional Form Fitting

libLGD/lgdClusterIU.c:

```
float DBFit2(float Rf, float w) {
  return ((DB_ParamA(w)/Rf)+(DB_ParamB(w)/(Rf*Rf))+(DB_ParamC(w)));
}

float DB_ParamA(float width) {
        return (((0.40633)*width)-1.4932);
}

float DB_ParamB(float width) {
        return (((-0.36261)*width)+1.4667);
}

float DB_ParamC(float width) {
        return (((-0.12124)*width)+0.43097);
}
```